Anne Benoit, Yves Robert, Frédéric Vivien

A guide to algorithm design

Paradigms, methods and complexity analysis

CRC PRESS
Boca Raton London New York Washington, D.C.

Chapter 1

Introduction to complexity

This chapter revisits basic notions on the cost of an algorithm and on the complexity of a problem. To illustrate these notions, in Section 1.1, we study the problem of computing x^n, given x and n (where n is a positive integer). Then, in Section 1.2, we recall the classical asymptotic notations O, o, Θ and Ω. Finally, exercises are proposed in Section 1.3, with their solutions in Section 1.4.

1.1 On the complexity to compute x^n

We study the problem of computing x^n, given x and n (where n is a positive integer). Note that x is not necessarily a number, it can be a matrix, a polynomial with several unknowns, or any mathematical object for which the multiplication is defined.

We let $y_0 = x$, and we use the following "rule of the game": If I have already computed $y_1, y_2, \ldots, y_{i-1}$, then I can compute y_i as a product of any of two previous temporary results: $y_i = y_j \times y_k$, with $0 \leqslant j, k \leqslant i-1$. The goal is to reach x^n as soon as possible, i.e., to minimize the *cost* of the algorithm, expressed in the number of multiplications. The cost is the first index m such that $y_m = x^n$.

We define $Opt(n)$ as the minimum index m such that $y_m = x^n$, where the minimum is taken over all algorithms, i.e., all possible sequences of y_i. The cost of an algorithm, therefore, is always greater than or equal to $Opt(n)$. Formally,

$$Opt(n) = \min \left\{ m \;\middle|\; \begin{array}{l} \exists y_0 = x,\ y_1, y_2, \ldots, y_{m-1},\ y_m = x^n, \\ \forall i \in [1, m],\ \exists j, k \in [0, i-1],\ y_i = y_j \times y_k \end{array} \right\}.$$

In the following, we present four methods to compute x^n, and we compare their costs. Then we end the section with some complexity results that aim at providing bounds on $Opt(n)$.

1.1.1 Naive method

Let us consider the following naive algorithm: $y_i = y_0 \times y_{i-1}$. We have $y_{n-1} = x^n$, and thus a cost of $n - 1$.

1.1.2 Binary method

We can easily find a method more efficient than the naive algorithm:

$$x^n = \begin{cases} x^{n/2} \times x^{n/2} & \text{if } n \text{ is even,} \\ x^{\lfloor n/2 \rfloor} \times x^{\lfloor n/2 \rfloor} \times x & \text{if } n \text{ is odd.} \end{cases}$$

This algorithm can be formulated as follows: We write n in binary, and then we replace each "1" by SX and each "0" by S, and we remove the first SX. The word that we obtain gives a method to compute x^n. The i-th letter indicates how to compute y_i; letter S corresponds to a *squaring* operation ($y_i = y_{i-1} \times y_{i-1}$), while letter X corresponds to a *multiplying by x* operation ($y_i = y_{i-1} \times y_0$).

For instance, for $n = 23$ (n=10111), we obtain SX S SX SX SX, and after removing the first SX, we obtain the word SSXSXSX. Therefore, we compute, in order, $y_1 = y_0 \times y_0 = x^2$, $y_2 = y_1 \times y_1 = x^4$, $y_3 = y_2 \times y_0 = x^5$, $y_4 = y_3 \times y_3 = x^{10}$, $y_5 = y_4 \times y_0 = x^{11}$, $y_6 = y_5 \times y_5 = x^{22}$, and finally $y_7 = y_6 \times y_0 = x^{23}$.

The correction of the algorithm is easy to justify from the properties of binary decomposition. The cost is $\lfloor \log(n) \rfloor + \nu(n) - 1$, where $\nu(n)$ is the number of 1s in the binary writing of n. $\nu(n) - 1$ is thus the number of Xs, and $\lfloor \log(n) \rfloor$ is the number of Ss in the word. Logarithms are taken in base 2 here, and this will be the case throughout the book unless specified otherwise. In the example $n = 23$, there are four Ss and three Xs, and the cost is, therefore, 7. This value is also obtained with the formula.

Note that this binary method is not optimal; for instance, with $n = 15$, we get the word SXSXSX, leading to six multiplications, while one could notice that $15 = 3 \times 5$, that we need two multiplications to compute $z = x^3$ ($z = (x \times x) \times x$), and then three additional ones to compute $x^{15} = z^5$ (with the binary method: z^2, z^4, z^5).

1.1.3 Factorization method

This method is based on the factorization of n, that is applied recursively when $n \geqslant 2$:

$$x^n = \begin{cases} (x^p)^q & \text{if } p \text{ is the smallest prime factor of } n \ (n = p \times q), \\ x^{n-1} \times x & \text{if } n \text{ is a prime number.} \end{cases}$$

For instance, with this method, for $n = 15$, we obtain the computation described above, i.e., $x^{15} = (x^3)^5 = x^3 \times (x^3)^4$, leading to five multiplications: $y_1 = y_0 \times y_0 = x^2$, $y_2 = y_1 \times y_0 = x^3$, $y_3 = y_2 \times y_2 = (x^3)^2$, $y_4 = y_3 \times y_3 = (x^3)^4$, $y_5 = y_4 \times y_2 = (x^3)^5 = x^{15}$.

Note that if n is a power of 2, this method is identical to the binary method. Also, this factorization method is not optimal. For instance, with $n = 33$, we have seven multiplications ($x^{33} = (x^3)^{11} = x^3 \times (x^3)^{10} = x^3 \times ((x^3)^2)^5 = x^3 \times z \times z^4$, with $z = (x^3)^2$), while the binary method requires only six multiplications ($x^{33} = x \times x^{2^5}$). Note also that there is an infinity of numbers for which the factorization method is better than the binary method ($n = 15 \times 2^k$), and reciprocally ($n = 33 \times 2^k$).

However, we need to emphasize the fact that the cost of decomposing n into prime numbers is not accounted in this formulation, while this would be necessary to correctly quantify the cost of the factorization method. The problem is that we do not know, as of today, how to decompose n in polynomial time. This problem is indeed still open.

1.1.4 Knuth's tree method

The last method that we detail consists in using *Knuth's tree* [64], illustrated on Figure 1.1. The path from the root of the tree to n indicates a sequence of exponents from which we can compute efficiently x^n.

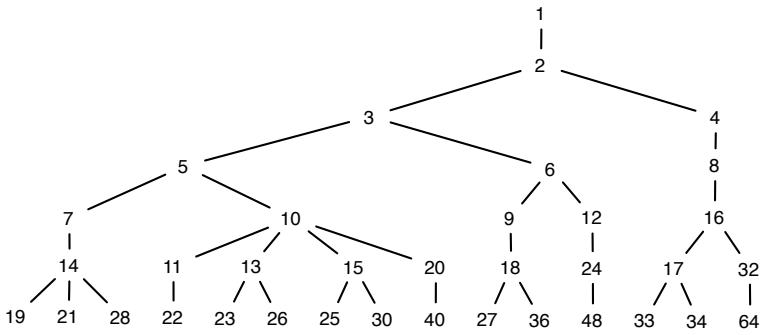


FIGURE 1.1: The first seven levels of Knuth's tree.

Building the tree. The root of the tree is 1. The tree is then built by induction. The $(k+1)$-th level of the tree is defined from the first k levels as follows. Consider each node j of the k-th level from the left to the right, and create nodes $j+1, j+a_1, j+a_2, \ldots, j+a_{k-1} = 2j$ at level $k+1$, as children of node j, in this order from left to right, where $1, a_1, \ldots, a_{k-1} = j$ is the path from the root to j. We do not add a node in the tree if there is already a node with the same value.

The algorithm. The algorithm simply consists in finding n in the tree (it appears only once by construction), and extracting nodes on the path from the root to n: $1, a_1, \ldots, n$. At each step of the algorithm, we compute $y_i = x^{a_i}$ as a product of two previous temporary results, which is possible by construction of the tree. The number of products to be done, i.e., the cost of the algorithm, is equal to the length of the path.

Statistics. Some interesting statistics are extracted from Knuth's book [64]. The smallest numbers for which the tree method is not optimal are $n = 77$, $n = 154$, and $n = 233$. The smallest number for which the tree method is better both to the binary and the factorization methods is $n = 23$. The smallest number for which the tree method is worse than the factorization method is $n = 19,879 = 103 \times 193$, and such cases are rare; for $n \leqslant 100,000$, the tree method is better than the factorization method $88,803$ times, it is equivalent $11,191$ times, and it is worse than the factorization method only 6 times.

At this point, we have several algorithms, but we do not know anything on the value of $Opt(n)$ yet. To assess the complexity of the problem, we have to provide bounds or asymptotic estimates for $Opt(n)$.

1.1.5 Complexity results

THEOREM 1.1. *For all integer $n \geqslant 1$, $Opt(n) \geqslant \lceil \log(n) \rceil$.*

Proof. Let us consider an algorithm that computes x^n in m steps. Recall that y_i is the intermediate result at step i of the algorithm and thus $y_m = x^n$. Let $\alpha(i)$ be the integer such that $y_i = x^{\alpha(i)}$, for $1 \leqslant i \leqslant m$. Then we prove by induction that $\alpha(i) \leqslant 2^i$.

Initially, we have $y_0 = x$, and thus $\alpha(0) = 1 \leqslant 1 = 2^0$.

For $1 \leqslant i \leqslant m$, there exist j and k $(0 \leqslant j, k < i)$ such that $y_i = y_j \times y_k$, by definition of the algorithm. Therefore, we have $\alpha(i) = \alpha(j) + \alpha(k)$, and we can apply the induction hypothesis on j and k, leading to $\alpha(j) \leqslant 2^j \leqslant 2^{i-1}$, and $\alpha(k) \leqslant 2^k \leqslant 2^{i-1}$. Finally, we have $\alpha(i) \leqslant 2^{i-1} + 2^{i-1} = 2^i$, which concludes the proof. □

Intuitively, the proof expresses the fact that we cannot do better at each step than doubling the exponent. Thanks to this theorem and to the study of the binary method, whose number of steps is bounded by $2\lfloor \log(n) \rfloor$ (recall that $\log(n)$ denotes $\log_2(n)$), we have the following result for all $n \geqslant 2$:

$$1 \leqslant \frac{Opt(n)}{\lceil \log(n) \rceil} \leqslant 2.$$

THEOREM 1.2. $\lim_{n \to \infty} \dfrac{Opt(n)}{\log(n)} = 1.$

Proof. The idea is to improve the binary method by applying it in base b. We let $b = 2^k$, where the value of k will be fixed later, and we write n in base b: $n = \alpha_0 b^t + \alpha_1 b^{t-1} + \cdots + \alpha_t$, where $t = \lfloor \log_b(n) \rfloor$, and $0 \leqslant \alpha_i \leqslant b-1$ (for $0 \leqslant i \leqslant t$). Then, we compute all x^d, for $1 \leqslant d \leqslant b-1$, with the naive method, in $b-2$ multiplications. Note that we do not necessarily need all these values (only the ones corresponding to the α_is), but they are computed on the fly and we can compute them without significant additional cost.

Then we successfully compute:

$$\begin{aligned} y_0 &= x^{\alpha_0}, \\ y_1 &= (y_0)^b \times x^{\alpha_1} &&= x^{\alpha_0 b + \alpha_1}, \\ y_2 &= (y_1)^b \times x^{\alpha_2} &&= x^{(\alpha_0 b + \alpha_1) b + \alpha_2}, \\ &\vdots \\ y_t &= (y_{t-1})^b \times x^{\alpha_t} = x^n. \end{aligned}$$

At each step i (for $1 \leqslant i \leqslant t$), we need $k+1$ computations (k squaring to compute $(y_{i-1})^b$, and one multiplication by x^{α_i}), and, therefore, we have a total cost of

$$\begin{aligned} t \times (k+1) + (b-2) &= \lfloor \log_b(n) \rfloor (k+1) + 2^k - 2 \\ &\leqslant (\log_b(n))\,(k+1) + 2^k = (\log(n))\,\frac{k+1}{k} + 2^k \end{aligned}$$

(recall that $\log_b(a) = \log_x(a)/\log_x(b)$).

We want k to be a function of n tending to infinity when n tends to infinity, so that we have $(k+1)/k$ tending to 1, and such that $2^k = o(\log(n))$ (see Section 1.2 for a definition of the o-notation). For instance, with $k = \lfloor \frac{1}{2} \log(\log(n)) \rfloor$, we have $2^k \leqslant \sqrt{\log(n)}$. (As we are only interested in the asymptotic behavior, we assume that $n > 16$; then $k \geqslant 1$ and $b \geqslant 2$.)

Therefore, we have $Opt(n) \leqslant (\log(n))\frac{k+1}{k} + \sqrt{\log(n)}$ and $\frac{k+1}{k} + \frac{1}{\sqrt{\log(n)}}$ tends to 1 when n tends to infinity. □

Note that this method is somewhat complicated, only to gain a factor 2 by comparison to the binary method.

Finally, we point out that the complexity of the problem of computing x^n is still open, i.e., we do not know whether there exists a polynomial-time method that performs the exact minimum number of operations. Formally, the underlying problem is that of *addition chains*. Starting with $a_0 = 1$, and given $a_0, a_1, \ldots, a_i$ for $i \geqslant 0$, we build a_{i+1} as $a_{i+1} = a_j + a_k$ where $0 \leqslant j \leqslant k \leqslant i$. The length of the chain is the smallest integer $\ell(n)$, if it exists, such that $a_{\ell(n)} = n$. Clearly, the a_is represent the exponents of the values x^{a_i}s that we compute to derive x^n. Given n, what is the complexity to derive an addition chain of minimal length?

An optimal method is easily derived from Knuth's tree. If we keep all possibilities in Knuth's tree, i.e., we add a node in the tree even if it already exists

somewhere else, then we have an exhaustive method that always performs the minimum number of operations. However, this method clearly takes an exponential amount of time, and thus is not satisfying. In fact, to the best of our knowledge, the complexity of the problem is still open. There is a common misbelief that the problem of determining whether there exists an addition chain whose length does not exceed some bound is NP-complete. In fact, the result is known to be NP-complete only for a sequence of integers $n_1, n_2, \ldots, n_m$, but not for a single value n [33].

1.2 Asymptotic notations: O, o, Θ and Ω

Let $f(n)$ be a function, where n is an integer. The asymptotic notations describe the complexity of the function for large values of n.

We say that $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that for all $n \geqslant n_0$, $0 \leqslant f(n) \leqslant c\, g(n)$. The O-notation allows us to give an upper bound on the function, up to within a constant factor.

The o-notation expresses the fact that the upper bound is not asymptotically tight: $f(n) = o(g(n))$ if for any positive constant c, there exists a positive constant n_0 such that for all $n \geqslant n_0$, $0 \leqslant f(n) < c\, g(n)$.

The Ω-notation provides an asymptotic lower bound on the function: $f(n) = \Omega(g(n))$ if there exist positive constants c and n_0 such that for all $n \geqslant n_0$, $0 \leqslant c\, g(n) \leqslant f(n)$.

The Θ-notation is more accurate, since it bounds the function both from below and above: $f(n) = \Theta(g(n))$ if there exist positive constants c_1, c_2 and n_0 such that for all $n \geqslant n_0$, $0 \leqslant c_1\ g(n) \leqslant f(n) \leqslant c_2\ g(n)$. In other words, $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

1.3 Exercises

Exercise 1.1: Longest balanced section (solution p. 14)

Let F be an array of size $n \geqslant 1$ whose elements are 0 or 1. A section $[i..j]$ of consecutive elements of F, with $1 \leqslant i < j \leqslant n$, is *balanced* if it contains as many 0 as 1 elements:

$$\text{card}\{k \mid F[k] = 0,\ i \leqslant k \leqslant j\} = \text{card}\{k \mid F[k] = 1,\ i \leqslant k \leqslant j\}.$$

The length of a balanced section $[i..j]$ is its number of elements $j - i + 1$. The goal of this exercise is to find the longest balanced section of F.

1. Provide a solution whose complexity is $O(n^2)$.
2. Provide a solution whose complexity is $O(n)$.

The reader may want to think for a while before reading the following hint for linear-time complexity. Introduce an array $Q[-n..n]$ of size $2n+1$ and let $Q[b]$ be the first index j such that the imbalance of section $[1..j]$ in F is equal to b. Here the imbalance $\text{imbal}(i,j)$ of section $[i..j]$ is defined as

$$\text{imbal}(i,j) = \text{card}\{k \mid F[k] = 1,\ i \leqslant k \leqslant j\} - \text{card}\{k \mid F[k] = 0,\ i \leqslant k \leqslant j\}.$$

Exercise 1.2: Find the star (solution p. 15)

In a group of n persons (numbered from 1 to n), a *star* is someone who does not know anybody else, but who is known by all other persons. Our goal is to identify a star, if one exists, in the group. The only action that can be taken is to ask a question to any person i: "Do you know person j?" We assume that everybody tells the truth.

1. How many stars can exist in the group?
2. Design an algorithm to find the star (if any) that requires $O(n)$ questions.
3. Provide a lower bound on the complexity (in terms of number of questions) of any algorithm solving the problem. Prove that the best lower bound for this problem is $3n - \lfloor \log(n) \rfloor - 3$.

Exercise 1.3: Breaking boxes (solution p. 16)

The problem consists in finding the lowest floor of a building from which a box would break when dropping it. The building has n floors, numbered from 1 to n, and we have k boxes. There is only one way to know whether dropping a box from a given floor will break it or not. Go to that floor and throw a box from the window of the building. If the box does not break, it can be collected at the bottom of the building and reused.

The goal is to design an algorithm that returns the index of the lowest floor from which dropping a box will break it. The algorithm returns $n+1$ if a box does not break when thrown from the n-th floor. The cost of the algorithm, to be kept minimal, is expressed as the number of boxes that are thrown (note that re-use is allowed).

1. For $k \geqslant \lceil \log(n) \rceil$, design an algorithm with $O(\log(n))$ boxes thrown.
2. For $k < \lceil \log(n) \rceil$, design an algorithm with $O\left(k + \frac{n}{2^{k-1}}\right)$ boxes thrown.
3. For $k = 2$, design an algorithm with $O(\sqrt{n})$ boxes thrown.

Exercise 1.4: Maximum of n integers (solution p. 17)

The goal is to compute the maximum of n integers, and we study the complexity of the algorithms in terms of number of comparisons and number of assignments.

1. Write a naive algorithm to solve the problem. What is its complexity in the worst and best cases?

2. Is this algorithm optimal for the number of comparisons in the worst case?

3. What is its complexity in the average number of comparisons or assignments? To compute the average number of assignments, you may use the following reasoning. Let $P_{n,k}$ be the number of permutations σ of $\{1, \ldots, n\}$ such that on $T[1] = \sigma(1)$, $\ldots$, $T[n] = \sigma(n)$, the algorithm performs k assignments. Give a recurrence relation for $P_{n,k}$. Let $G_n(z) = \sum P_{n,k} z^k$. Prove that $G_n(z) = z(z+1)\cdots(z+n-1)$, and give a conclusion.

Exercise 1.5: Maximum and minimum of n integers (solution p. 20)

The goal is to compute simultaneously the maximum and the minimum of n integers, and we study the complexity of the algorithms in terms of number of comparisons in the worst case.

1. Design a naive algorithm and give its complexity.

2. One idea to improve the algorithm is to group elements by pairs, in order to decrease the number of comparisons that must be done. Design an algorithm based on this idea, and analyze its complexity.

3. Prove the optimality of such an algorithm, by providing a lower bound on the number of comparisons. The idea is to use the *adversary* method. Let A be an algorithm that finds the maximum and minimum. For a given input, when the algorithm is executed, a *novice* is an element that has never been compared, a *winner* has been compared at least once and has always been superior in comparisons, a *loser* has been compared at least once and has always been inferior in comparisons, and the remaining elements are called *average* elements. The number of such elements is represented by a quadruplet of integers (i, j, k, l), with, of course, $i + j + k + l = n$. Give the value of this quadruplet at the beginning and at the end of the algorithm. Provide a strategy for the adversary, so as to maximize the duration of the execution of the algorithm. Conclude with a lower bound on the number of comparisons.

Exercise 1.6: Maximum and second maximum of n integers (solution p. 23)

The goal is to compute simultaneously the maximum and the second maximum of n integers, and we study the complexity of the algorithms in terms of the number of comparisons in the worst case.

1. Design a naive algorithm and give its complexity.
2. One idea to improve the algorithm is to compute the maximum following a tournament (as, for instance, a tennis tournament). If there are $n = 2^k$ numbers taking part into the tournament, how do we find the maximum and the second maximum, once the tournament is over? What is the complexity of this algorithm? In the general case, how can we adapt the algorithm for any value of n?
3. Prove the optimality of this algorithm, by providing a lower bound on the number of comparisons. The idea is to use *decision trees*. The decision tree of an algorithm is a tree that represents all the possible executions of the algorithm, on every possible input of size n. The internal nodes correspond to tests. In our case, the test is a comparison, if the answer is "yes" we move to the left child, otherwise to the right child, hence having a binary tree. The leaves correspond to the results of the different executions (several leaves may correspond to the same result). Each branch of the tree corresponds to an execution of the algorithm, and the number of comparisons is the height of the branch. The number of comparisons in the worst case is then obtained as the height of the tree.
   (a) Prove that any decision tree that computes the maximum of n integers has at least 2^{n-1} leaves.
   (b) Prove that any binary tree of height h and with f leaves is such that $2^h \geqslant f$.
   (c) Let A be a decision tree solving the problem. Give a lower bound on its number of leaves. Conclude with a lower bound on the number of comparisons in the worst case.

Exercise 1.7: Merging two sorted sets (solution p. 25)

The goal is to merge two sorted sets, a set A of size m, and a set B of size n. The $m+n$ numbers to merge are all different and such that $A_1 < A_2 < \cdots < A_m$ and $B_1 < B_2 < \cdots < B_n$.

1. Prove that we need at least $\left\lceil \log \binom{m+n}{n} \right\rceil$ comparisons for the merge (recall that logarithms are taken in base 2).

2. Deduce that for $n = m$, there is a constant k such that, when n is sufficiently large, we need at least $2n - \frac{1}{2}\log(n) - k$ comparisons for the merge.

3. Recall briefly the usual merging algorithm and give its complexity.

4. Prove that for $n = m$, we cannot do better than the usual algorithm. Therefore, the lower bound of Question 2 cannot be matched.

Exercise 1.8: The toolbox (solution p. 26)

In a toolbox, there are n nuts, all of different sizes, and n corresponding bolts. However, everything is mixed up, and you wish to associate each nut with the corresponding bolt. The size differences are so small that it is not possible to decide if a nut (or a bolt) is larger than another one just by looking at them. The only way to proceed consists in trying one nut with one bolt, and each operation can lead to three possible answers: (i) the nut is strictly larger than the bolt; (ii) the bolt is strictly larger than the nut; and (iii) they correspond to each other.

1. Design a simple algorithm with $O(n^2)$ operations that associates each nut with the corresponding bolt.

2. Prove that the problem of finding the smallest nut and the corresponding bolt can be solved with no more than $2n - 2$ operations.

3. Prove that any algorithm solving the initial problem (i.e., associate each nut with the corresponding bolt) requires at least $\Omega(n \log(n))$ operations in the worst case.

Exercise 1.9: Sorting a small number of objects (solution p. 29)

This exercise investigates the complexity of sorting a small number of objects, when the only possible operation is the comparison of two objects. For n elements, we know that the number of comparisons is at least $\lceil \log(n!) \rceil$ (see, for instance, Section 10.2 page 243). We ask whether this bound can be reached. Asymptotically, this is true because, for instance, the merge sort has a complexity in $O(n \log(n))$ in the worst case. We check if the bound can be exactly reached in terms of number of comparisons. In the following table, for $2 \leqslant n \leqslant 12$ objects, we indicate the lower bound on the number of comparisons ($\lceil \log(n!) \rceil$), the number of comparisons done by a merge sort algorithm (merge-sort(n)), and the optimal number of comparisons (opt(n)).

n	2	3	4	5	6	7	8	9	10	11	12
$\lceil \log(n!) \rceil$	1	3	5	7	10	13	16	19	22	26	29
merge-sort(n)	1	3	5	8	11	14	17	21	25	29	33
opt(n)	1	3	5	7	10	13	16	19	22	26	30

Therefore, merge-sort is not reaching the bound as soon as $n \geqslant 5$. The goal of this exercise is to design ad hoc sorting algorithms for each value of n ($2 \leqslant n \leqslant 12$) that perform the optimal number of comparisons $\mathrm{opt}(n)$.

Several techniques can be used:

- *Binary-search insertion*: If we want to insert an element in a sorted set of k elements, the cost is of r comparisons in the worst case if $2^{r-1} \leqslant k \leqslant 2^r - 1$. Therefore, it is less costly to insert an element in a set of 3 elements than in a set of 2 elements (2 comparisons in both cases), and in a set of 7 elements rather than between 4 and 6 elements (3 comparisons), because the cost in the worst case is the same. In other words, insertion is the most cost-effective when $k = 2^r - 1$.

- *Incremental sort* of n elements: We first sort $n - 1$ elements, and then we insert the last one with a binary-search insertion.

- *Divide-and-conquer*: To sort four elements, we create two pairs of two elements $(a \to b)$ and $(c \to d)$, where $(a \to b)$ means that $a \leqslant b$, and then we compare the two largest elements to obtain, for instance, $(a \to b \to d)$. Finally, we insert c with a binary search. The following figure illustrates this technique:

$$\begin{array}{ccc} a & \longrightarrow & b \\ & & \searrow \\ & c & \longrightarrow \quad d \end{array}$$

For instance, for $n = 3$, we compare two elements, hence obtaining $(a \to b)$ with one comparison, and then we compare the third element to a and b with two more comparisons, obtaining $3 = \mathrm{opt}(3)$ comparisons. For $n = 4$, we can use the incremental sort to obtain $(a \to b \to c)$ with three comparisons, and then we insert the last element with a binary search, with two comparisons, hence a total of $3 + 2 = 5 = \mathrm{opt}(4)$ comparisons.

1. Provide another technique for $n = 4$, based on divide-and-conquer.

2. Following the previous ideas, provide algorithms for any value $5 \leqslant n \leqslant 11$ that perform $\mathrm{opt}(n)$ comparisons.

3. For $n = 12$, provide a method with 30 comparisons. Indeed, it is impossible to succeed with $\lceil \log(12!) \rceil = 29$ comparisons; researchers have tested all possible algorithms with the brute force method, it took two hours of computation in 1990 (and it was a real challenge at that time!).

Chapter 2

Divide-and-conquer

This chapter revisits the divide-and-conquer paradigms and explains how to solve recurrences, in particular, with the use of the "master theorem." We first illustrate the concept with Strassen's matrix multiplication algorithm (Section 2.1) before explaining the master theorem (Section 2.2), and finally providing techniques to solve recurrences (Section 2.3). These techniques are further illustrated in the exercises of Section 2.4, with solutions found in Section 2.5.

2.1 Strassen's algorithm

The classical matrix multiplication algorithm computes the product of two matrices of size $n \times n$ with $Add(n) = n^2(n-1)$ additions and $Mult(n) = n^3$ multiplications. Indeed, there are n^2 coefficients to compute, each of them corresponding to a scalar product of size n, thus with n multiplications, $n-1$ additions, and one affectation. Can we do better than this?

Note that the question was raised at a time when it was mainly interesting to decrease the number of multiplications, even though this would imply computing more additions. The pipelined architecture of today's processors allows us to perform, in steady-state mode, one addition or one multiplication per cycle time.

Strassen introduced a new method in his seminal paper [102]. Let us compute the product of two 2×2 matrices:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

We first compute seven intermediate products

$$\begin{aligned} p_1 &= a(f-h) \\ p_2 &= (a+b)h \\ p_3 &= (c+d)e \\ p_4 &= d(g-e) \\ p_5 &= (a+d)(e+h) \\ p_6 &= (b-d)(g+h) \\ p_7 &= (a-c)(e+f) \end{aligned}$$

and then we can write

$$\begin{aligned} r &= p_5 + p_4 - p_2 + p_6 \\ s &= p_1 + p_2 \\ t &= p_3 + p_4 \\ u &= p_5 + p_1 - p_3 - p_7 \end{aligned}$$

If we count operations for each method, we obtain the following:

Classic	*Strassen*
$Mult(2) = 8$	$Mult(2) = 7$
$Add(2) = 4$	$Add(2) = 18$

Strassen's method gains one multiplication, but at the price of 14 extra additions, thus being worse on modern processors than the classical method for 2×2 matrices. However, it is remarkable that the new method does not require the commutativity of multiplication, and, therefore, it can be used, for instance, with matrices instead of numbers. We can readily use it with matrices of even size n, say $n = 2m$. We consider that $a, b, c, d, e, f, g, h, r, s, t$, and u are matrices of size $m \times m$. So, let $n = 2m$, and use the previous approach with submatrices of size $m \times m$. To compute each p_i ($1 \leqslant i \leqslant 7$) with the classic matrix multiplication algorithm, we need m^3 multiplications, thus a total $Mult(n) = 7m^3 = 7n^3/8$. For the additions, we need to add the additions performed in the seven matrix multiplications to form the intermediate products p_i, namely $7m^2(m-1)$, with the number of additions required to form the auxiliary matrices, namely $18m^2$. Indeed, there are 10 matrix additions to compute the p_is, and then 8 other matrix additions to obtain r, s, t, and u. Therefore, we have a total of $Add(n) = 7m^3 + 11m^2 = 7n^3/8 + 11n^2/4$.

Asymptotically, the dominant term is in $\frac{7}{8}n^3$ for $Mult(n)$ as for $Add(n)$, and the new method is interesting for n large enough. The intuition is the following: Multiplying two matrices of size $n \times n$ requires $O(n^3)$ operations (both for pointwise multiplications and additions), while adding two matrices of size $n \times n$ requires only $O(n^2)$ operations. For n large enough, matrix additions have a negligible cost in comparison to matrix multiplications (and the main source of pointwise additions is within these matrix multiplications). That was not the case for real numbers, hence, the inefficiency of the method for 2×2 matrices.

Strassen's algorithm is the recursive use of the decomposition explained above. We consider the case in which n is a power of 2, i.e., $n = 2^s$. Otherwise, we can extend all matrices with zeroes so that they have a size that is the first power of 2 greater than n, and replace in the following $\log(n)$ by $\lceil \log(n) \rceil$:

$$(X) \longrightarrow \begin{pmatrix} X & 0 \\ 0 & 0 \end{pmatrix} .$$

Let us consider matrices of size $n \times n$, where $n = 2^s$. We proceed by induction. We use the method recursively to compute each of the matrix products p_i, for $1 \leqslant i \leqslant 7$. We stop when matrices are of size 1 or, better, when Strassen's method is more costly than the classical method, for matrix sizes below a "crossover point." In practice, this crossover point is highly system dependent. By ignoring cache effects, we can obtain crossover points as low as $n = 8$ [50], while [30] determines the crossover points by benchmarking on various systems, and it ranges from $n = 400$ to $n = 2150$.

In the following, we stop the recursion when $n = 1$, and:

- $M(n)$ is the number of multiplications done by Strassen's algorithm to multiply two matrices of size $n \times n$;
- $A(n)$ is the number of additions done by Strassen's algorithm to multiply two matrices of size $n \times n$.

For the multiplications, we have:

$$\begin{cases} M(1) = 1 \\ M(n) = 7 \times M(n/2) \end{cases} \Longrightarrow M(n) = 7^s = 7^{\log(n)} = n^{\log(7)}.$$

As before, additions come from two different sources: the additions that are done in the 7 matrix multiplications (recursive call), and the 18 matrix additions (construction of the p_i's and of r, s, t, and u). We finally have:

$$\begin{cases} A(1) = 0 \\ A(n) = 7 \times A(n/2) + 18 \times (n/2)^2 \end{cases} \Longrightarrow A(n) = 6 \times (n^{\log(7)} - n^2) \quad (2.1)$$

We explain in Section 2.3 how this recurrence can be solved. Note that the recursive approach has improved the order of magnitude of the total computation cost, not just only the constant (previously, we only had $\frac{7}{8}n^3$ instead of n^3). The new order of magnitude is $O(n^{\log(7)})$ and $\log(7) \approx 2.81$.

Finally, we conclude by saying that Strassen's algorithm is not widely used, because it introduces some numerical instability. Also, there are some algorithms with a better complexity. At the time of this writing, the best algorithm is the Coppersmith–Winograd algorithm, in $O(n^{2.376})$ [27]. The problem of establishing the complexity of matrix product is still open. The only known lower bound is a disappointing $O(n^2)$; we need to touch each coefficient at least once.

Strassen's algorithm provides, however, an excellent illustration of the divide-and-conquer paradigm, that we formalize in the next section through the master theorem.

2.2 Master theorem

Before formulating the master theorem, we need to formalize the divide-and-conquer paradigm that was illustrated in the previous section through the Strassen's algorithm.

DEFINITION 2.1 (Divide-and-conquer)**.** Consider a problem of size n. In order to solve the problem, divide it into a subproblems of size n/b that will allow us to find the solution. The cost of this divide-and-conquer algorithm is then

$$S(n) = a \times S\left(\frac{n}{b}\right) + R(n) \tag{2.2}$$

where $R(n)$ is the cost to reconstruct the solution of the problem of size n from the solutions of the subproblems; it is often equal to $R(n) = c \times n^\alpha$, for some constants c and α. Initially, we often have $S(1) = 1$ (or equal to another constant value).

For instance, with Strassen's algorithm, if we consider the number of additions to be executed in a matrix product, we have $a = 7$, $b = 2$, $\alpha = 2$, and $c = \frac{18}{4}$. Indeed, the product of two matrices of size $n \times n$ is performed by first computing 7 products of matrices of size $n/2 \times n/2$, and reconstructing the solution through 18 additions of matrices of size $n/2 \times n/2$, therefore, $R(n) = 18(n/2)^2 = \frac{18}{4}n^2$. In this case, the initial cost is $S(1) = 0$.

Let us assume that there exists $k \in \mathbb{N}$ such that $n = b^k$, thus $k = \log_b(n)$ and $a^k = n^{\log_b(a)}$. If we develop the formula in equation (2.2), we obtain the following:

$$\begin{aligned} S(n) &= a \times S(\tfrac{n}{b}) \quad + R(n) \\ &= a^2 \times S(\tfrac{n}{b^2}) + a \times R(\tfrac{n}{b}) + R(n) \\ &= \cdots \\ &= a^k \times S(1) \quad + \textstyle\sum_{i=0}^{k-1} a^i \times R(\tfrac{n}{b^i}). \end{aligned}$$

We consider the most usual case in which $R(n) = c \times n^\alpha$, and, therefore, we have $\sigma = \sum_{i=0}^{k-1} a^i \times R(\frac{n}{b^i}) = c \times n^\alpha \sum_{i=0}^{k-1} (a/b^\alpha)^i$.

We then distinguish several cases:

1. $(a > b^\alpha)$: $\sigma = \Theta(n^\alpha \times (\frac{a}{b^\alpha})^k) = \Theta(a^k) \Longrightarrow S(n) = \Theta(n^{\log_b(a)})$;
2. $(a = b^\alpha)$: $\sigma = \Theta\left(k \times n^\alpha\right) \Longrightarrow S(n) = \Theta\left(n^\alpha \times \log(n)\right)$;
3. $(a < b^\alpha)$: $\sigma = \Theta\left(n^\alpha \times \frac{1}{1-\frac{a}{b^\alpha}}\right) \Longrightarrow S(n) = \Theta(n^\alpha)$.

We have proved the following theorem:

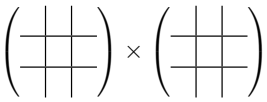
THEOREM 2.1 (Master theorem)**.** *The cost of a divide-and-conquer algorithm such that $S(n) = a \times S(\frac{n}{b}) + c \times n^\alpha$ is the following:*

(i) if $a > b^\alpha$, *then* $S(n) = \Theta(n^{\log_b(a)})$;

(ii) if $a = b^\alpha$, *then* $S(n) = \Theta(n^\alpha \times \log(n))$;

(iii) if $a < b^\alpha$, *then* $S(n) = \Theta(n^\alpha)$.

A fully detailed proof of Theorem 2.1 is given in [28]. Let us come back to Strassen's algorithm. We divided the matrices into four blocs of size $n/2$, and we would like to investigate a solution in which we would rather divide matrices into nine blocs of size $n/3$:



We would then have $b = 3$ and $\alpha = 2$ (the reconstruction cost is still in n^2). Let us assume that we are in case (i) of the master theorem. Then, this new algorithm would become better than Strassen's if and only if:

$$\begin{array}{rr} & \log_3(a) < \log(7) \\ \iff & \log(a) < \log(7) \times \log(3) \\ \iff & a < 7^{\log(3)} \approx 21.8. \end{array}$$

This is an open problem; one knows a method with $a = 23$ subproblems [71], but not with $a = 21$!

2.3 Solving recurrences

In this section, we detail how to solve recurrences that occur in the cost analysis of divide-and-conquer algorithms, but that are slightly more complex than in the application case of the master theorem. We start with homogeneous recurrences, and then consider the most general case of recurrences with a second member.

2.3.1 Solving homogeneous recurrences

A homogeneous linear recurrence with constant coefficients has the form $p_0 \times s_n + p_1 \times s_{n-1} + \cdots + p_k \times s_{n-k} = 0$, where each p_i is a constant and $(s_i)_{i \geqslant 0}$ is an unknown sequence. It is said to be homogeneous because the second member is null, i.e., the linear combination is set equal to zero. Solving such recurrences requires finding all the roots of the polynomial $P = \sum_{i=0}^{k} p_i \times X^{k-i}$, together with their multiplicity order. However, we see that P is a polynomial of degree k, and no algebraic method can find the roots of arbitrary polynomials

of degree 5 or higher. Therefore, we need additional information, such as trivial roots, for high-degree recurrences.

Let us assume that we can find the k roots of P, $r_1, \ldots, r_k$. If these roots are distinct, then the general form of the solution is $s_n = \sum_{i=1}^{k} c_i \times {r_i}^n$, where the c_is are some constants that depend upon the first values of the sequence. Otherwise, let q_i be the order of multiplicity of root r_i, for $1 \leqslant i \leqslant \ell$ (with $\ell < k$ distinct roots). Then we have $s_n = \sum_{i=1}^{\ell} P_i(n) \times {r_i}^n$, where $P_i(n)$ is a polynomial of degree $q_i - 1$. Here again, the coefficients of the $P_i(n)$s are computed using the initial values of the recurrence.

2.3.2 Solving nonhomogeneous recurrences

In the general case, the recurrence may have a nonzero right-hand side, for instance, $s_n - 2s_{n-1} = 2^{n+1}$. Such recurrences are called *nonhomogeneous*. To explain how to solve them, we start by introducing a few notations. A sequence is represented by writing down its n-th element formula in brackets, for instance $\{3^n\}$ represents the sequence $1, 3, 9, 27, \ldots$ (starting at $n = 0$).

Then we introduce E, an operator that transforms a sequence by shifting it and leaving out its first element. In our example, $E\{3^n\} = 3, 9, 27, 81, \ldots = \{3^{n+1}\}$. More generally, $E\{s_n\} = \{s_{n+1}\}$.

We then define the following operations on sequences:

$$\begin{aligned} c\{s_n\} &= \{cs_n\}, \\ (E_1 + E_2)\{s_n\} &= E_1\{s_n\} + E_2\{s_n\}, \\ (E_1 E_2)\{s_n\} &= E_1(E_2\{s_n\}). \end{aligned}$$

For instance, $(E - 3)\{s_n\} = \{s_{n+1} - 3s_n\}$, and $(2 + E^2)\{s_n\} = \{2s_n + s_{n+2}\}$.

We are looking for annihilators of the sequences. That is, we are looking for operators $P(E)$ such that $P(E)\{s_n\} = \{0\}$. For our example, $(E - 3)\{3^n\} = \{3^{n+1} - 3 \times 3^n\} = \{0\}$. We provide a few more examples, where $Q_k(n)$ is a polynomial in n of degree k:

sequence	annihilator
$\{c\}$	$E - 1$
$\{Q_k(n)\}$	$(E - 1)^{k+1}$
$\{c^n\}$	$E - c$
$\{c^n \times Q_k(n)\}$	$(E - c)^{k+1}$

The first three lines are special cases of the fourth line, therefore, we only need to prove the last relation. We prove it by induction on k. We start with $k = 0$, writing $Q_0(n) = q$:

$$(E - c)\{c^n \times Q_0(n)\} = qE\{c^n)\} - c\{qc^n\} = \{qc^{n+1}\} - \{qc^{n+1}\} = \{0\}.$$

Now by induction for $k \geqslant 1$, writing $Q_k(n) = a_0 n^k + Q_{k-1}(n)$:

$$\begin{aligned}(E-c)^{k+1}\{c^n \times Q_k(n)\} &= (E-c)^{k+1}\{c^n \times (a_0 n^k + Q_{k-1}(n))\} \\ &= (E-c)^k[(E-c)\{c^n(a_0 n^k + Q_{k-1}(n))\}] \\ &= (E-c)^k\{c^{n+1}(a_0(n+1)^k + Q_{k-1}(n+1)) \\ &\qquad - \ c^{n+1}(a_0 n^k + Q_{k-1}(n))\} \\ &= (E-c)^k[c^{n+1} \times R_{k-1}(n)],\end{aligned}$$

where $R_{k-1}(n)$ is a polynomial in n of degree $k-1$, because both $(n+1)^k - n^k$ and $Q_{k-1}(n+1) - Q_{k-1}(n)$ are polynomials of degree $k-1$. With the induction hypothesis, we obtain the result: $(E-c)^{k+1}\{c^n \times Q_k(n)\} = \{0\}$.

2.3.3 Solving the recurrence for Strassen's algorithm

We focus on the recurrence for the number of additions (see equation (2.1)):

$$A(n) = 7 \times A\left(\frac{n}{2}\right) + \frac{18}{4} \times n^2.$$

We have $n = 2^s$, and we consider the sequence $\{A_s\}$ such that $A_s = A(2^s)$. Thus, we have $A_{s+1} = 7 \times A_s + \frac{18}{4} \times (2^{s+1})^2 = 7 \times A_s + 18 \times 4^s$, and the annihilator is $(E-4)(E-7)$:

$$(E-4)(E-7)\{A_s\} = (E-4)\{A_{s+1} - 7A_s\} = (E-4)\{18 \times 4^s\} = \{0\}.$$

We have found an annihilator for the sequence, namely $(E-4)(E-7)$, and, even better, it is in decomposed form, so we immediately have its two distinct roots, 4 and 7. From the previous result, we know the general form of the solution, namely

$$A_s = k_1 \times 7^s + k_2 \times 4^s \ .$$

From the initial conditions $A_0 = 0$ and $A_1 = 18$, we obtain the values $k_1 = 6$ and $k_2 = -6$, and, finally,

$$A(n) = 6 \times 7^s - 6 \times 4^s \ .$$

2.4 Exercises

Exercise 2.1: Product of two polynomials (solution p. 42)

The goal of this exercise is to multiply two polynomials efficiently. An n-polynomial is a polynomial with a degree strictly less than n, thus with n coefficients.

Let $P = \sum_{i=0}^{n-1} a_i X^i$ and $Q = \sum_{i=0}^{n-1} b_i X^i$ be two n-polynomials. Their product $R = P \times Q$ is a $(2n-1)$-polynomial. We denote by $M(n)$ (resp. $A(n)$) the number of multiplications (resp. number of additions) done by an algorithm to multiply two n-polynomials.

1. Compute $M(n)$ and $A(n)$ for the usual algorithm to multiply two n-polynomials.
2. We assume that n is even, $n = 2 \times m$. We can then write $P = P_1 + X^m \times P_2$ and $Q = Q_1 + X^m \times Q_2$. What is the degree of the polynomials P_1, P_2, Q_1, and Q_2?
3. Let $R_1 = P_1 \times Q_1$, $R_2 = P_2 \times Q_2$, and $R_3 = (P_1 + P_2) \times (Q_1 + Q_2)$. Can you express $R = P \times Q$ as a function of R_1, R_2, and R_3? What is the degree of these three new polynomials? Compute $M(n)$ and $A(n)$, assuming that we use the classical multiplication algorithm to compute R_1, R_2, and R_3.
4. We assume now that $n = 2^s$ and we apply recursively the previous algorithm. Compute $M(n)$ and $A(n)$ for this algorithm.

Exercise 2.2: Toeplitz matrices (solution p. 44)

A *Toeplitz matrix*, or diagonal-constant matrix, named after Otto Toeplitz, is an $n \times n$ matrix with $(a_{i,j})$ coefficients $(1 \leqslant i, j \leqslant n)$, and such that $a_{i,j} = a_{i-1,j-1}$ for $2 \leqslant i, j \leqslant n$.

1. Let A and B be two Toeplitz matrices. Is the sum $A + B$ a Toeplitz matrix? And the product $A \times B$?
2. Give an algorithm to add two Toeplitz matrices in $O(n)$.
3. We assume here that $n = 2^k$. How can we compute the product of an $n \times n$ Toeplitz matrix M by a vector T of length n? What is the complexity of the algorithm?

   Hint: Decompose M as a matrix of blocks of size 2^{k-1}, decompose T accordingly:

   $$M = \begin{pmatrix} A & B \\ C & A \end{pmatrix} \quad \text{and} \quad T = \begin{pmatrix} X \\ Y \end{pmatrix}$$

   and consider the three matrices $U = (C + A)X$, $V = A(Y - X)$, and $W = (B + A)Y$.

Exercise 2.3: Maximum sum (solution p. 45)

Let T be a table of n relative integers. We want to find the maximum sum of contiguous elements, namely, two indices i and j $(1 \leqslant i \leqslant j \leqslant n)$ that maximize $\sum_{k=i}^{j} T[k]$.

1. If the values in the table are $T[1] = 2$, $T[2] = 18$, $T[3] = -22$, $T[4] = 20$, $T[5] = 8$, $T[6] = -6$, $T[7] = 10$, $T[8] = -24$, $T[9] = 13$, and $T[10] = 3$, can you return the two indices and the corresponding optimal sum?

2. Design an algorithm that returns the maximum sum of contiguous elements with a divide-and-conquer algorithm.

3. Design a linear-time algorithm that solves the problem through a single scan of the array.

Exercise 2.4: Boolean matrices: The Four-Russians algorithm
(solution p. 49)

The goal in this exercise is to multiply two $n \times n$ Boolean matrices, A and B. All matrix elements are either 0 or 1, and the sum and product correspond respectively to the *or* and *and* operations on Booleans.

1. Can we easily apply Strassen's algorithm to compute the product $A \times B$?

2. Apart from the classical multiplication algorithm, another way to view the product consists in multiplying columns of A with rows of B. Give an expression of $A \times B$, using $A_c[\ell]$, the ℓ-th column of A, and $B_r[\ell]$, the ℓ-th row of B.

3. To optimize the matrix product, the idea is to partition the columns of A and the rows of B into n/k equal-sized groups of size k (we can assume, for simplicity, that k divides n; otherwise, the last group is smaller). Therefore, for $1 \leqslant i \leqslant n/k$, A_i is a $n \times k$ matrix with k columns of A ($A_c[(i-1) \times k + 1], \ldots, A_c[(i-1) \times k + k]$), and, similarly, B_i is a $k \times n$ matrix with k rows of B ($B_r[(i-1) \times k + 1], \ldots, B_r[(i-1) \times k + k]$). Give an expression of $A \times B$, using the matrices A_i and B_i.

4. Provide a method to compute $C_i = A_i \times B_i$ in time $O(n^2)$, for all $1 \leqslant i \leqslant n/k$. (*Hint:* Show that each row of C_i can take only 2^k different values, precompute all possible values and store them in a table. What is the size of the table, i.e., the additional space required to run the algorithm? What is the time required to build the table?)

5. Building upon the previous method, provide an algorithm to compute $A \times B$, and give its complexity, in terms of k and n.

6. Which value of k would be most suited for this algorithm? What is the complexity of this matrix product algorithm? Compare with Strassen's algorithm.

Note that this algorithm is known as the *Four-Russians algorithm*, and it is due to Arlazarov et al. [3, 77].

Exercise 2.5: Matrix multiplication and inversion (solution p. 50)

Let $M(n)$ be the complexity of multiplying two square matrices of size n and $I(n)$ be the complexity of inverting a (square) matrix of size n. The functions $M(n)$ and $I(n)$ are not known, but the goal of this exercise is to show the following: If we assume that $M(n) = \Theta(n^\alpha)$ and $I(n) = \Theta(n^\beta)$, then $\alpha = \beta$. In other words, both operations have same order complexity under our hypothesis.

1. Prove that $2 \leqslant \alpha, \beta \leqslant 3$.
2. Prove that $\alpha \leqslant \beta$; matrix multiplication is not more complex than matrix inversion (which is intuitive).
3. Prove that $\beta \leqslant \alpha$; reciprocally, matrix inversion is not more complex than matrix multiplication (which is less intuitive).
   (*Hint*: Show that we can reduce the problem to inverting symmetric and positive definite matrices A whose size is an exact power of 2, and use the Schur complement $S = D - CB^{-1}C^T$ to recursively compute the inverse of $A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix}$. Note that B and D are symmetric and positive definite, too.)

2.5 Solutions to exercises

Solution to Exercise 2.1: Product of two polynomials

1. With the usual algorithm to multiply n-polynomials:

$$M(n) = n^2 \text{ and } A(n) = n^2 - \underbrace{(2n-1)}_{assignments} = (n-1)^2.$$

   Indeed, we multiply each of the n coefficients of P with each of the n coefficients of Q. Then, the number of additions is equal to the number of multiplication results minus the number of results computed and there are $2n-1$ coefficients in the computed polynomial.

2. P_1, P_2, Q_1, and Q_2 are m-polynomials and of degree $m-1$.
3. We have $R = R_1 + (R_3 - R_2 - R_1) \times X^m + R_2 \times X^{2m}$. R_1, R_2, and R_3 are polynomials of degree $2m-2 = n-2$, and thus $(n-1)$-polynomials.

   Following this computation scheme, $M(n) = 3M(\frac{n}{2}) = \frac{3n^2}{4}$, as the computation of R_1, R_2, and R_3 each requires $M(m) = M(\frac{n}{2})$ multiplications. There are four types of additions: (1) those involved in the

Chapter 3

Greedy algorithms

This chapter explains the reasoning in finding optimal greedy algorithms. The main feature of a greedy algorithm is that it builds the solution step by step, and, at each step, it makes a decision that is locally optimal. Throughout Sections 3.1 to 3.3, we illustrate this principle with several examples, and also outline situations where greedy algorithms are not optimal; taking a good local decision may prove a bad choice in the end! In Section 3.4, we also cover matroids, a (mostly theoretical) framework to prove the optimality of greedy algorithms. All of these techniques are then illustrated with a set of exercises in Section 3.5, with solutions found in Section 3.6.

3.1 Motivating example: The sports hall

Problem. Let us consider a sports hall in which several events should be scheduled. The goal is to have as many events as possible, given that two events cannot occur simultaneously (only one hall). Each event i is characterized by its starting time s_i and its ending time e_i. Two events are compatible if their time intervals do not overlap. We would like to solve the problem, i.e., find the maximum number of events that can fit in the sports hall, with a greedy algorithm.

A first greedy algorithm. The first idea consists in sorting events by increasing durations $e_i - d_i$. At each step, we schedule an event into the sports hall if it fits, i.e., if it is compatible with events that have already been scheduled. The idea is that we will be able to accommodate more shorter events than longer ones. However, we make local decisions at each step of the algorithm (this is a greedy algorithm!), and it turns out that we can make decisions that do not lead to the optimal solution. For instance, in the example of Figure 3.1, the greedy algorithm schedules only the shortest event i, while the two compatible events j and k would lead to a better solution.

A second greedy algorithm. In order to avoid the problem encountered in the previous example, we design a new algorithm that sorts events by starting

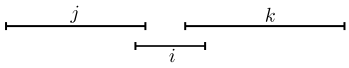


FIGURE 3.1: The first greedy algorithm is not optimal.

times s_i, and then proceeds similarly to the first greedy algorithm. In the example of Figure 3.1, this greedy algorithm returns the optimal solution. However, the local decisions that are made may not be the optimal ones, as shown in the example of Figure 3.2. Indeed, the algorithm schedules event i at the first step, and then no other event can be scheduled, while it would be possible to have eight compatible events. Note that the first greedy algorithm would return the optimal solution for this example.

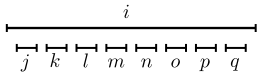


FIGURE 3.2: The second greedy algorithm is not optimal.

A third greedy algorithm. Building upon the first two algorithms, we observe that it is always a good idea to first select events that do not intersect with many other events. In the first example, events j and k intersect with only one other event, while event i intersects with two events and is chosen later; therefore, the new algorithm finds the optimal solution. Similarly in the second example, event i intersects eight other events and it is the only event not to be scheduled. However, this greedy algorithm is still not optimal. We can build an example in which we force the algorithm to make a bad local decision. In the example of Figure 3.3, event i is the first to be chosen because it has the smallest number of intersecting events. However, if we schedule i, we can have only three compatible events, while we could have a solution with four compatible events, j, k, l, and m.

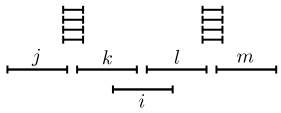


FIGURE 3.3: The third greedy algorithm is not optimal.

An optimal greedy algorithm. Even though many greedy choices do not lead to an optimal solution, as observed with the preceding algorithms, there is a greedy algorithm that solves the sports hall problem in polynomial time. The idea is to sort the events by increasing ending times e_i, and then to greedily schedule the events. This way, at each step we fit the maximum number of events up to a given time, and we never make a bad decision. We now prove the optimality of this algorithm.

Let f_1 be the event with the smallest ending time. We prove first that there exists an optimal solution that schedules this event. Let us consider an optimal solution $O = \{f_{i_1}, f_{i_2}, \ldots, f_{i_k}\}$, where k is the maximum number of events that can be scheduled in the sports hall, and where events are sorted by nondecreasing ending times. There are two possible cases: either (i) $f_{i_1} = f_1$, the optimal solution schedules f_1, and nothing needs to be done, or (ii) $f_{i_1} \neq f_1$. In this second case, we replace f_{i_1} with f_1 in solution O. We have $e_1 \leqslant e_{i_1}$ by definition of event f_1, and $e_{i_1} \leqslant s_{i_2}$ because O is a solution to the problem (f_{i_1} and f_{i_2} are compatible). Therefore, $e_1 \leqslant s_{i_2}$ and, thus, f_{i_2} is compatible with f_1. The new solution is still optimal (the number of events remain unchanged), and event f_1 is scheduled.

The proof works by induction, following the previous reasoning. Once f_1 is scheduled, we consider only events that do not intersect with f_1, and we iterate the reasoning on the remaining events to conclude the proof.

Finally, we emphasize that there can be many optimal solutions, and not all of them will include the first event f_1 selected by the greedy algorithm, namely the event with the smallest end time. However, schedules that select f_1 are *dominant*, meaning that there exists an optimal solution that includes f_1.

3.2 Designing greedy algorithms

The example of the sports hall gives a good introduction to the design principles of greedy algorithms. Actually, the binary method to compute x^n in Section 1.1.2 also is a greedy algorithm, in which we decide at each step which computation to perform. We can formalize the reasoning to find greedy algorithms as follows:

1. Decide on a greedy choice that allows us to locally optimize the problem;

2. Search for a counterexample that shows that the algorithm is not optimal (and go back to step 1 if a counterexample is found), or prove its optimality through steps 3 and 4;

3. Show that there is always an optimal solution that performs the greedy choice of step 1;

4. Show that if we combine the greedy choice with an optimal solution of the subproblem that we still need to solve, then we obtain an optimal solution.

We say that a greedy algorithm is a *top-down* algorithm, because at each step we make a local choice, and we then have a single subproblem to solve, given this choice. On the contrary, we will see in Section 4 that dynamic programming algorithms are *bottom-up*; we will need results of multiple subproblems to make a choice.

3.3 Graph coloring

In this section, we further illustrate the principle of greedy algorithms through the example of graph coloring. The problem consists in coloring all vertices of a graph using the minimum number of colors, while enforcing that two vertices, which are connected with an edge, are not of the same color. Formally, let $G = (V, E)$ be a graph and $c : V \to \{1..K\}$ be a K-coloring such that $(x, y) \in E \Rightarrow c(x) \neq c(y)$. The objective is to minimize K, the number of colors.

3.3.1 On coloring bipartite graphs

We start with a small theorem that allows us to define a bipartite graph, defined as a graph that can be colored with only two colors.

THEOREM 3.1. *A graph can be colored with two colors if and only if all its cycles are of even length.*

Proof. Let us first consider a graph G that can be colored with two colors. Let $c(v) \in \{1, 2\}$ be the color of vertex v. We prove by contradiction that all cycles are of even length. Indeed, if G has a cycle of length $2k + 1$, $v_1, v_2, \ldots, v_{2k+1}$, then we have $c(v_1) = 1$, say, which implies that $c(v_2) = 2$, $c(v_3) = 1$, until $c(v_{2k+1}) = 1$. However, since it is a cycle, there is an edge between v_1 and v_{2k+1}, so they cannot be of the same color, which leads to the contradiction.

Now, if all the cycles of the graph G are of even length, we search for a 2-coloring of this graph. We assume that G is connected (the problem is independent from one connected component to another). The idea consists in performing a breadth-first traversal of G.

Le $x_0 \in G$, $X_0 = \{x_0\}$ and $X_{n+1} = \bigcup_{y \in X_n} N(y)$, where $N(y)$ is the set of nodes connected to y, but not yet included in a set X_k, for $k \leqslant n$. Each vertex appears in one single set, and we color with color 1 the elements from sets X_{2k}, and with color 2 the elements from sets X_{2k+1}.

This 2-coloring is valid if and only if two vertices connected by an edge are of different colors. If there is an edge between $y \in X_i$ and $z \in X_j$, where i and j are both either even or odd, then we have a cycle $x_0, \ldots, y, z, \ldots, x_0$ of length $i + j + 1$, and this value is even, leading to a contradiction. The coloring, therefore, is valid, which concludes the proof. □

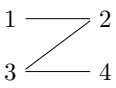
In a bipartite graph, if we partition vertices into two sets according to the colors, all edges go from one set to the other. We retrieve here the usual definition of bipartite graphs, namely graphs whose vertices are partitioned into two sets and with no edge inside these sets. We now consider colorings of general graphs, and we propose a few greedy algorithms to solve the problem.

3.3.2 Greedy algorithms to color general graphs

The first greedy algorithm takes the vertices in a random order, and, for each vertex v, it colors it with the smallest color number that has not been yet given to a neighbor of v, i.e., a node connected to v.

Let $K_{greedy1}$ be the total number of colors needed by this greedy algorithm. Then we have $K_{greedy1} \leqslant \Delta(G) + 1$, where $\Delta(G)$ is the maximal degree of a vertex (number of edges of the vertex). Indeed, at any step of the algorithm, when we color vertex v, it has at most $\Delta(G)$ neighboring vertices and, therefore, the greedy algorithm never needs to use more than $\Delta(G)+1$ colors.

Note that this algorithm is optimal for a fully connected graph (a clique), since we need $\Delta(G)+1$ colors to connect such a graph (one color per vertex). However, this algorithm is not optimal in general; on the following bipartite graph, if the order of coloring is 1 and then 4, we need three colors, while the optimal coloring uses only two colors.

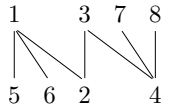


In order to improve the previous algorithm, one idea consists in ordering vertices in a smart way, and then in proceeding as before, i.e., color each vertex in turn with the smallest possible color.

Let $n = |V|$ be the number of vertices, and d_i be the degree of vertex v_i. We have $K_{greedy2} \leqslant \max_{1 \leqslant i \leqslant n} \min(d_i + 1, i)$. Indeed, when we color vertex v_i, it has at most $\min(d_i, i - 1)$ neighbors that have already been colored, and thus its own color is at most $1 + \min(d_i, i - 1) = \min(d_i + 1, i)$. To obtain the result, we take the maximum of these values on all vertices.

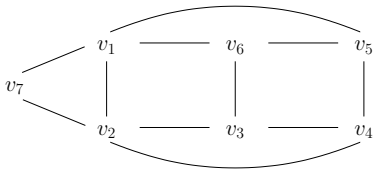
This result suggests that it would be smart to first color vertices with a high degree, so that we have $\min(d_i + 1, i) = i$. Therefore, the second greedy algorithm sorts the vertices by nonincreasing degrees.

Once again, the algorithm is not optimal. On the following bipartite graph, we choose to color vertex 1, then vertex 4, which imposes the use of three colors instead of the two required ones.



Based on these ideas, several greedy algorithms can be designed. In particular, a rather intuitive idea consists in giving priority to coloring vertices that have already many colored neighbors. We define the color-degree of a vertex as the number of its neighbors that are already colored. Initially, the color-degree of each vertex is set to 0, and then it is updated at each step of the greedy algorithm.

The following greedy algorithm is called the *Dsatur* algorithm in [20]. The ordering is done by (color-degree, degree); we choose a vertex v with maximum color-degree, and such that its degree is the largest among the vertices with maximum color-degree. This vertex v is then colored with the smallest possible color, and the color-degrees of the neighbors of v are updated before proceeding to the next step of the algorithm. We illustrate this algorithm on the following example:

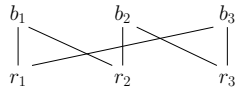


We first choose a vertex with maximum degree, for instance v_1, and it is colored with color 1. The color-degree of v_2, v_5, v_6, and v_7 becomes 1, and we choose v_2, which has the maximum degree (between these four vertices); it is assigned color 2. Now, v_7 is the only vertex with color-degree 2; it is given the color 3. All remaining noncolored vertices have the same color-degree 1 and the same degree 3, we arbitrarily choose v_3 and color it with 1. Then, v_4, with color-degree 2, receives color 3. Finally, v_5 is colored in 2 and v_6 in 3; the graph is 3-colored, and it is an optimal coloring.

The name *Dsatur* comes from the fact that maximum color-degree vertices are saturated first. We prove below that *Dsatur* always returns an optimal coloring on bipartite graphs; however, it may use more colors than needed on arbitrary graphs.

THEOREM 3.2. *The* Dsatur *algorithm is optimal on bipartite graphs, i.e., it always succeeds to color them with two colors.*

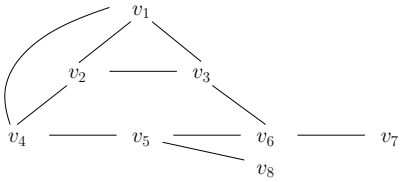
Proof. Consider a connected bipartite graph $G = (V, E)$, where $V = B \cup R$ and each edge in E is connecting a vertex in B (color 1 is blue) and a vertex in R (color 2 is red). Note first that the first two greedy algorithms may fail. Let G be such that $B = \{b_1, b_2, b_3\}$, $R = \{r_1, r_2, r_3\}$, and $E = \{(b_1, r_2), (b_2, r_3), (b_3, r_1), (b_i, r_i) | 1 \leqslant i \leqslant 3\}$, as illustrated below.



All vertices have a degree 2. If we start by coloring a vertex of B, for instance b_1, and then a nonconnected vertex of R, r_3, with the same color 1, it is not possible to complete the coloring with only two colors. The use of the color-degree prevents us from such a mistake, since once b_1 has been colored, we need to color either r_1 or r_2 with the color 2, and finish the coloring optimally.

In the general case, with *Dsatur*, we first color a vertex, for instance from B, with color 1 (blue). Then we have to color a vertex of color-degree 1, that is, a neighboring vertex. This neighboring vertex belongs necessarily to R. It is colored with color 2 (red). We prove by induction that at any step of the algorithm, all colored vertices of B are colored in blue, and all vertices of R are colored in red. Indeed, if the coloring satisfies this property at a given step of the algorithm, we choose next a vertex v with nonnul color-degree. Because the graph is bipartite, all its neighbors are in the same set and have the same color: red if $v \in B$, or blue if $v \in R$. Vertex v, therefore, is colored in red if it is in R, or in blue if it is in B. □

We exhibit a counterexample to show that *Dsatur* is not optimal on arbitrary graphs.



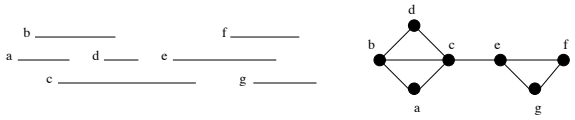
Dsatur can choose v_4 first, because it has the maximum degree 3; it is colored with 1. Between the vertices with color-degree 1, the algorithm can (arbitrarily) choose v_5, which is colored with 2. Then the algorithm can choose to color v_6, using color 1. Then, v_1 is chosen between vertices of color-degree 1 and degree 3, and it is colored with 2. We finally need to use colors 3 and 4 for v_2 and v_3, while this graph could have been colored with only three colors (v_1, v_5, v_7 with color 1, v_2, v_6, v_8 with color 2, and v_3, v_4 with color 3).

To build this counterexample, we force *Dsatur* to make a wrong decision, by coloring both v_4 and v_6 with color 1, and v_1 with color 2, which forces four colors because of v_2 and v_3. Note that it would be easy to built an example without any tie (thereby avoiding random choices) by increasing the degree of some vertices (for instance, in the example, v_7 and v_8 are just there to increase the degrees of v_5 and v_6).

The problem of coloring general graphs is NP-complete, as will be shown in Chapter 7. However, for a particular class of graphs, a smart greedy algorithm can return the optimal solution, as we detail below.

3.3.3 Coloring interval graphs

We focus now on interval graphs. Given a set of intervals, we define a graph whose vertices are intervals, and whose edges connect intersecting intervals. The following example shows such a graph, obtained with a set of seven intervals.



The problem of coloring such a graph is quite similar to the sports hall problem. Indeed, one can see each interval as representing an event, with its starting and ending time, and the color as representing a sports hall. Then, only compatible events will be colored with the same color, and we could use one sports hall per set of compatible events. If we minimize the number of colors, we minimize the number of sports halls that are needed to organize all events.

Graphs that are obtained from a set of intervals are called interval graphs. We define the following greedy algorithm: intervals (i.e., vertices) are sorted by nondecreasing starting times (or left extremity). In the example, the order is a, b, c, d, e, f, g. Then, the greedy coloring is done as before; for each chosen vertex, we color it with the smallest compatible color. On the example, we obtain the coloring $1, 2, 3, 1, 1, 2, 3$, which is optimal, as the graph contains a cycle of length 3.

We prove now that this greedy algorithm is optimal for any interval graph. Let G be such a graph, and let d_v be the starting time of interval v corresponding to vertex v. We execute the greedy algorithm; it uses k colors. If vertex v receives color k, then this means that $k-1$ intervals that start no later than d_v intersect this interval and had all been colored with colors 1 to $k-1$; otherwise, v would be colored with a color $c \leqslant k-1$. All of these intervals are thus intersecting, because they all contain the point d_v; therefore, graph G contains a clique of size k. Since all vertices of a clique must be colored with distinct colors, we cannot color the graph with less than k colors. The greedy algorithm, therefore, is optimal.

Once again, we point out that the order chosen by the greedy algorithm is vital, since we could force the greedy algorithm to make a wrong decision, even on a bipartite graph as below, if we would not proceed from left to right. We could first color a, then d, leading to the use of three colors instead of two.

3.4 Theory of matroids

In this section are elementary results on matroids, a framework that allows us to guarantee the optimality of a generic greedy algorithm in some situations. Unfortunately, it is not easy to characterize which problems can be captured as matroid instances. Still, the theory is beautiful, and we outline its main ideas.

Matroids. The term *matroid* was introduced in 1935 by H. Whitney [108], while working on the linear independence of the vector columns of a matrix. We define it below and illustrate the concept through a canonical example.

DEFINITION 3.1. $(S, \mathcal{I})$ is a matroid if S is a set of n elements, and $\mathcal{I}$ is a collection of subsets of S, with the following properties:

i. $X \in \mathcal{I} \Rightarrow (\forall\ Y \subset X,\ Y \in \mathcal{I})$ (*hereditary property*), and
ii. $(A, B \in \mathcal{I},\ |A| < |B|) \Rightarrow \exists x \in B \backslash A$ s.t. $A \cup \{x\} \in \mathcal{I}$ (*exchange property*).
If $X \in \mathcal{I}$, X is said to be an *independent set*.

Readers familiar with linear algebra will immediately see that linearly independent subsets of a given vector set form a matroid. The canonical computer science example follows.

Example of matroid: Forests of a graph. Let $G = (V, E)$ be a (nondirected) graph. We define a matroid with $S = E$ (the elements are the edges of the graph), and $\mathcal{I} = \{A \subset E \mid A \text{ has no cycle}\}$. Therefore, a set of edges is an independent set if and only if this set of edges is a forest of the graph, i.e., a set of trees (a tree is a connected graph with no cycle). We check that this matroid satisfies both properties.

(i) The hereditary property. It is pretty obvious that a subset of a forest is a forest; if we remove edges from a forest, we cannot create a cycle, thus we still have a forest.

(ii) The exchange property. Let A and B be two forests of G (i.e., $A, B \in \mathcal{I}$) such that $|A| < |B|$. $|A|$ is the number of edges in forest A, and every vertex is part of a tree (an isolated vertex with no edges is a tree made of a single vertex). Then A (resp. B) contains $|V| - |A|$ (resp. $|V| - |B|$) trees. Indeed, each time an edge is added to the independent set, two trees are connected, therefore decrementing the number of trees by one. Thus, B contains less trees than A, and there exists a tree T of B that is not included in a tree of A, i.e., two vertices u and v of tree T are not in the same tree of A. On the path from u to v in T, there are two vertices, connected by an edge (x, y), that are not in the same tree of A. Then, if we add this edge to the forest A, we still have a forest, i.e., $A \cup \{(x, y)\} \in \mathcal{I}$, which concludes the proof.

DEFINITION 3.2. Let $F \in \mathcal{I}$; $x \notin F$ is an *extension* of F if $F \cup \{x\} \in \mathcal{I}$, i.e., $F \cup \{x\}$ is an independent set. An independent set is *maximal* if it has no extensions.

In our running example, any edge connecting two distinct trees of a forest is an extension. A forest is maximal if adding any edge to it would create a cycle. A maximal independent set in the example of the forest is a spanning tree (or spanning forest if G is not connected).

LEMMA 3.1. *All maximal independent sets are of same cardinal.*

Proof. If this lemma was not true, we could find an extension to the independent set of smaller cardinal thanks to the exchange property, which would mean that it was not maximal. □

We introduce a last definition: We add weights to the elements of the matroid and, therefore, obtain a *weighted matroid*.

DEFINITION 3.3. In a *weighted matroid*, each element of S has a weight: $x \in S \mapsto w(x) \in \mathbb{N}$. The weight of a subset $X \subset S$ is defined as the sum of the weights of its elements: $w(X) = \sum_{x \in X} w(x)$.

Greedy algorithms on a weighted matroid. The problem is to find an independent set of maximum weight. The idea of the greedy algorithm is to sort elements of S by nonincreasing weights. We start with the empty set, which always is an independent set because of the hereditary property. Then we add elements into this set, as long as we keep an independent set. This generic algorithm is formalized in Algorithm 3.1.

```
1 Sort elements of S = {s_1, ..., s_n} by nonincreasing weight:
  w(s_1) ⩾ w(s_2) ⩾ ··· ⩾ w(s_n)
2 A ← ∅
3 for i = 1 to n do
4 |   if A ∪ {s_i} ∈ I then
5 |   |   A ← A ∪ {s_i}
```

ALGORITHM 3.1: Independent set of maximum weight.

THEOREM 3.3. *Algorithm 3.1 returns an optimal solution to the problem of finding an independent set of maximum weight in the weighted matroid.*

Proof. Let s_k be the first independent element of S, i.e., the first index i of the algorithm such that $\{s_i\} \subset \mathcal{I}$. We first prove that there exists an optimal solution that contains s_k.

Let B be an optimal solution, i.e., an independent set of maximum weight. If $s_k \in B$, we are done. Otherwise, let $A = \{s_k\} \in \mathcal{I}$. While $|B| > |A|$, we apply the exchange property to add an element of B to the independent set A. We obtain the independent set with $|B|$ elements, $A = \{s_k\} \cup B \backslash \{s_j\}$, where $\{s_j\}$ is the one element of B that has not been chosen for the extension (there is already element s_k in A, and at the end, $|A| = |B|$, therefore all elements of B but one are extensions of A).

We now compare the weights. We have $w(A) = w(B) - w(s_j) + w(s_k)$. Moreover, $w(s_k) \geqslant w(s_j)$, because s_j is independent (by hereditary property), and $j > k$ (by definition of s_k). Finally, $w(A) \geqslant w(B)$, and since B is an optimal solution, $w(A) = w(B)$. The independent set A is of maximal weight, and it contains s_k, which proves the result.

To prove the theorem, we show by induction that the greedy algorithm returns the optimal solution; we restrict the search to a solution that contains s_k, and we start the reasoning again with $S' = S \backslash \{s_k\}$, and $\mathcal{I}' = \{X \subset S' \mid X \cup \{s_k\} \in \mathcal{I}\}$. □

Back to the running example. Theorem 3.3 proves the optimality of Kruskal's algorithm to build a minimum weight spanning tree [69]. Edges are sorted by nondecreasing weight, and we choose greedily the next edge that does not add a cycle when added to the current set of edges. Of course, we should discuss a suitable data structure so that we can easily check the condition "no cycle has been created." With a simple array, we can check the condition in $O(n^2)$, and it is possible to achieve a better complexity with other data structures [28]. In any case, the complexity of the greedy algorithm remains polynomial.

Example: A semimatching problem. In this very simple example, we are given a directed weighted graph. The problem is to find a maximum weight subset of the edges so that no two starting points are the same. A natural greedy algorithm would sort all edges according to their weight in nonincreasing order, then consider all edges in this order, selecting an edge (i, j) if and only if no edge (i, j') had been selected earlier. In fact, this greedy algorithm selects for every node the outgoing edge that has maximum weight, hence, it can be easily implemented in time $O(n + m)$, where n is the number of nodes, and m the number of edges, of the directed graph. While the optimality of this greedy algorithm is not difficult to prove directly, we prove it using matroid theory.

The problem can be cast in terms of a matrix W with nonnegative entries, with the goal to select a set of entries whose sum is maximal, subject to the constraint that no two entries are from the same row of the matrix. There are n rows in W, one per node in the graph. Let W_{ij} be the entry in row i and column j of the matrix W, and let $x_{ij} \in \{0, 1\}$ be the indicator of

whether W_{ij} is selected. We aim at maximizing $\sum_{i,j} W_{ij}x_{ij}$ subject to the set of constraints $\sum_j x_{ij} \leqslant 1$ for each row i. The greedy algorithm chooses entries one at a time in order of weight, largest first (and breaking ties arbitrarily), rejecting an entry only if an entry in the same row has already been chosen. Here is an example, where chosen entries are underlined:

$$W = \begin{pmatrix} \underline{12} & 7 & 10 & 11 \\ 8 & 6 & 4 & \underline{16} \\ 3 & \underline{5} & 2 & 1 \\ 14 & 13 & 9 & \underline{15} \end{pmatrix}$$

To prove that the greedy algorithm is optimal, we exhibit the matroid; independent sets are sets of entries such that no two of them are from the same row of the matrix. We show that both properties hold. The hereditary property is obvious. Indeed, when removing entries from an independent set, we cannot create a row with two entries or more. The exchange property is not difficult either. Let A and B be two independent sets with $|A| < |B|$. There is at most one element per row in A and B, so there must be a row that contains an element of B and no element of A. Adding this element to A preserves its independence. This concludes the proof of optimality of the greedy algorithm.

As mentioned before, it is not easy to exhibit matroid structures for which interesting and efficient greedy algorithms can be derived. A more complicated example that involves scheduling tasks with deadlines is studied in Exercise 3.5. We refer the reader to [72, 95] for much more material on matroids and greedoids.

3.5 Exercises

Exercise 3.1: Interval cover (solution p. 68)

We are given a set $X = \{x_1, \ldots, x_n\}$ of n points on a line.

1. Design a greedy algorithm that determines the smallest set of closed intervals of length 1 that contains all the points.
2. Prove the optimality of the algorithm and give its complexity.
3. Could you use the theory of matroids to prove the optimality of the algorithm?

Exercise 3.2: Memory usage (solution p. 69)

Given a memory of size L, we want to store a set of n files $P = (P_1, \ldots, P_n)$. File P_i $(1 \leqslant i \leqslant n)$ is of size a_i, where a_i is an integer. If $\sum_{i=1}^{n} a_i > L$, we

cannot store all files. We need to select a subset $Q \subseteq P$ of files to store, such that $\sum_{P_i \in Q} a_i \leqslant L$. We sort the files P_i by nondecreasing sizes ($a_1 \leqslant \cdots \leqslant a_n$).

1. Write a greedy algorithm that maximizes the number of files in Q. The output must be a Boolean table S such that $S[i] = 1$ if $P_i \in Q$, and $S[i] = 0$ otherwise. What is the complexity of this algorithm, in number of comparisons and number of arithmetic operations?

2. Prove that this strategy always returns a maximal subset Q. We define the utilization ratio as $\frac{\sum_{P_i \in Q} a_i}{L}$. How small can it be with our strategy?

3. We now want to maximize the utilization ratio, i.e., fill the memory as much as possible. Design a greedy algorithm for this new objective function.

4. Is the latter greedy algorithm optimal? How small can the utilization ratio be with this algorithm? Prove the result.

Exercise 3.3: Scheduling dependent tasks on several machines
(solution p. 71)

Let $G = (V, E)$ be a directed acyclic graph (DAG). Here G is a task graph. In other words, each node $v \in V$ represents a task, and each edge $e \in E$ represents a precedence constraint, i.e., if $e = (v_1, v_2) \in E$, then the execution of v_2 cannot start before the end of the execution of v_1. We need to schedule the tasks on an unlimited number of processors. Moreover, the execution time of task $v \in V$ is $w(v)$. The problem is to find a valid schedule, i.e., a start time $\sigma(v)$ for each task v such that no precedence constraints are violated, and that minimizes the total execution time. The reader may refer to Section 6.4.4, p. 140, for more background on scheduling.

1. Define formally (by induction) the top level $tl(v)$ of a task $v \in V$, which is the earliest possible starting time of task v.

2. Propose a greedy schedule of the tasks, based on the top levels, and prove its optimality. This schedule is called σ_{free}.

3. We define the bottom level $bl(v)$ of a task as the largest weight of a path from v to an output task, i.e., a task with no successor. The weight of the path includes the weight of v. Define bottom levels formally, and propose a schedule of the tasks, based on the bottom levels, that is called σ_{late}.

4. Show that any optimal schedule σ satisfies:

$$\forall v \in V,\ \sigma_{free}(v) \leqslant \sigma(v) \leqslant \sigma_{late}(v).$$

5. Give an example of a DAG that has at least three different optimal schedules.

Exercise 3.4: Scheduling independent tasks with priorities (solution p. 72)

We need to schedule n independent tasks, T_1, T_2, ..., T_n, on a single processor. Each task T_i has an execution time w_i and a priority p_i. Because we execute the tasks sequentially on a single processor and as we target an optimal schedule, we can focus on schedules that execute tasks as soon as possible. A schedule is then fully defined by the order followed to execute the tasks. In other words, here, a schedule of tasks $T_1, \ldots, T_n$ is a permutation $T_{\sigma(1)}$, $T_{\sigma(2)}$, ..., $T_{\sigma(n)}$, specifying the order in which tasks are executed. We assume that the first task to be executed is processed from time 0 on. The cost of a schedule is defined as $\sum_{i=1}^{n} p_i C_i$, where C_i is the completion time of task T_i, i.e., the date at which its processing was completed. We look for a schedule that minimizes this cost.

1. Consider any schedule and two tasks T_i and T_j that are executed consecutively under this schedule. Which task should be executed first in order to minimize the cost?

2. Design an optimal greedy algorithm. What is its complexity?

Exercise 3.5: Scheduling independent tasks with deadlines (solution p. 73)

The goal here is to exhibit a matroid to prove the optimality of a greedy algorithm. We need to schedule n independent tasks, T_1, T_2, ..., T_n, on a single processor. Each task T_i is executed in one time unit, but it has a deadline d_i that should not be exceeded. If a task does not complete its execution before its deadline, there is a cost w_i to pay. The objective here is to find a schedule that minimizes the sum of the costs of the tasks that are completed after their deadlines. A schedule, in this exercise, will be a function, $\sigma : \mathcal{T} \to \mathbb{N}$, that associates to each task its execution time, such that two tasks cannot be scheduled at the same time, i.e., for all $1 \leqslant i, j \leqslant n$, $\sigma(T_i) \neq \sigma(T_j)$. The first task can be executed at time 0.

We say that a task is *on time* if it finishes its execution before its deadline, and that it is *late* otherwise. Note that minimizing the cost of late tasks is equivalent to maximizing the cost of on-time tasks. A *canonical* schedule is such that (i) on-time tasks are scheduled before late tasks, and (ii) on-time tasks are ordered by nondecreasing deadlines.

1. Prove that there is always an optimal schedule that is canonical, i.e., we can restrict the search to canonical schedules.

2. Design a greedy scheduling algorithm to solve the problem. What is its complexity?

3. Illustrate the greedy algorithm on the following example with seven tasks; the tasks are sorted by nonincreasing w_i ($w_i = 8-i$, for $1 \leqslant i \leqslant 7$), and their deadlines are as follows: $d_1 = 4$, $d_2 = 2$, $d_3 = 4$, $d_4 = 3$, $d_5 = 1$, $d_6 = 4$, and $d_7 = 6$.

4. Prove the optimality of the algorithm by exhibiting a matroid.

While Exercises 3.4 and 3.5 deal with simple uniprocessor scheduling problems for which the greedy algorithm is optimal, there are many more complex scheduling problems [21]. These include, for instance, scheduling problems with tasks with different execution times, several machines, precedence constraints between tasks, and so on. More scheduling problems are described in Section 6.4.4, p. 140.

Exercise 3.6: Edge matroids (solution p. 74)

This exercise aims at illustrating the matroid theory. The goal here is to exhibit a weighted matroid, to design the corresponding greedy algorithm, and prove its optimality.

This exercise is a generalization of the semimatching algorithm presented in Section 3.4. We are given a directed graph $G = (V, E)$ whose edges have integer weights. Let $w(e)$ be the weight of edge $e \in E$. We also are given a constraint $f(u) \geqslant 0$ on the out-degree of each node $u \in V$. The goal is to find a subset of edges of maximal weight, and whose out-degree at any node satisfy the constraint. We see that if $f(u) = 1$ for all nodes, we retrieve the semimatching algorithm.

1. Define independent sets and prove you have a matroid.

2. What is the cardinal of maximal independent sets?

3. What is the complexity of the (optimal) greedy algorithm?

Exercise 3.7: Huffman code (solution p. 75)

Let Σ be a finite alphabet with at least two elements. A binary code is an injective application from Σ to the set of finite suites of 0 and 1 (i.e., a binary word, also called *code word*). The code can be naturally extended by concatenation to a mapping defined on the set Σ^* of words using the alphabet Σ. A code is said to be *of fixed length* if all the letters in Σ are coded by binary words of same size. A code is said to be *prefix* if no code word is a prefix of another code word. Given the code of a word in Σ^*, the decoding operation consists in finding the original word.

1. Prove that the decoding operation has a unique solution, both for a code of fixed length, and for a prefix code.

2. Represent a prefix code by a binary tree, where leaves are the letters of the alphabet Σ.

3. Consider a text in which each letter $c \in \Sigma$ appears with a frequency $f(c) \neq 0$. To each prefix code of this text, represented by a tree T, is associated a cost, defined by $B(T) = \sum_{c \in \Sigma} f(c) \times l_T(c)$, where $l_T(c)$ is the size of the code word of c. If $f(c)$ is exactly the number of occurrences of c in the text, then $B(T)$ is the number of bits in the encoded text. A prefix code T is *optimal* if, for this text, $B(T)$ is minimum. Prove that for any optimal prefix code there is a corresponding binary tree with $|\Sigma|$ leaves and $|\Sigma| - 1$ internal nodes.

4. Prove that there is an optimal prefix code such that two letters of smallest frequencies are siblings in the tree (i.e., their code words have the same size, and differ only by the last bit).

   Hint: Prove also that these two letters are leaves of maximal depths.

5. Given x and y, two letters of smallest frequencies, we consider the alphabet $\Sigma' = (\Sigma \setminus \{x, y\}) \cup \{z\}$, where z is a new letter with frequency $f(z) = f(x) + f(y)$. Let T' be the tree of an optimal code for Σ'. Prove that the tree T obtained from T' by replacing the leave associated to z by an internal node with two leaves x and y is an optimal code for Σ.

6. Using both previous questions, design an algorithm that returns an optimal code, and give its complexity. Illustrate the algorithm on the following problem instance: $\Sigma = \{a, b, c, d, e, g\}$, $f(a) = 45$, $f(b) = 13$, $f(c) = 12$, $f(d) = 16$, $f(e) = 9$, and $f(g) = 5$.

3.6 Solutions to exercises

Solution to Exercise 3.1: Interval cover

1. Algorithm 3.2 is a greedy algorithm to solve the interval cover problem. It builds a maximal length interval starting at the first point on the line, remove all points included in that interval, and then iterates.

2. Sorting the points cost $O(n \log(n))$ when the execution of the while loop costs $O(n)$. Therefore, the algorithm runs in $O(n \log(n))$.

   We prove the optimality of Algorithm 3.2 by induction on the number n of points. If $n = 1$, there is a single point, the algorithm returns a single interval and thus is optimal. Now, assume we have proved the

Chapter 4

Dynamic programming

In this chapter, we focus on how to find optimal dynamic-programming algorithms. A particular attention is paid to problem size in order to avoid exponential-cost algorithms. The chapter is illustrated with two classical examples: the coin changing problem and the knapsack problem. These techniques are then further illustrated with a set of exercises in Section 4.4, with solutions found in Section 4.5.

4.1 The coin changing problem

The problem is the following: If we want to make change for S cents, and we have infinite supply of each coin in the set $Coins = \{v_1, v_2, \ldots, v_n\}$, where v_i is the value of the i-th coin, what is the minimum number of coins required to reach the value S?

Greedy algorithm. We propose a greedy algorithm to solve the problem. First, we sort coins by nonincreasing values, then for each coin value we take as many coins as possible. The algorithm is formalized as Algorithm 4.1.

```
1 Sort elements of Coins = {v_1, ..., v_n} by nonincreasing values:
  v_1 ⩾ v_2 ⩾ ··· ⩾ v_n
2 R ← S        { R is the remaining sum to reach; it is initially S }
3 for i = 1 to n do
4 |   c_i = ⌊R / v_i⌋        { c_i is the number of coins of value v_i that are taken }
5 |   R ← R − c_i × v_i      { R is updated }
```

ALGORITHM 4.1: Greedy algorithm for the coin changing problem.

We first assume that $Coins = \{10, 5, 2, 1\}$ (a typical European set of coins). In this case, we can prove that Algorithm 4.1 is optimal:

- An optimal solution returns, at most, one coin of value 5 (if there are two, it is better to use one single coin of value 10).
- An optimal solution returns, at most, one coin of value 1 (otherwise, we can use a coin of value 2).
- An optimal solution returns, at most, two coins of value 2 (otherwise, to obtain $6 = 2 + 2 + 2$, we would rather use $6 = 5 + 1$: one coin 5 and one coin 1).

Therefore, in the optimal solution, there cannot be more than four coins that are not of value 10, and $5 + 2 + 2 + 1 = 10$, so if there are four such coins, we would rather use a coin of 10. Thus, the optimal solution uses, at most, three coins that are not of value 10, and their total is at most 9. We can then conclude that the optimal number of coins of value 10 is $\lfloor \frac{S}{10} \rfloor$, which is the number selected by the greedy algorithm. It is then easy to conclude that the greedy algorithm always selects the optimal number of coins of each value.

Note, however, that the greedy algorithm is not optimal for any set of coins. For instance, if $\mathit{Coins} = \{6, 4, 1\}$ and $S = 8$, the greedy algorithm requires three coins $8 = 6 + 1 + 1$, while the optimal solution requires two coins of value 4. Still, U.S. readers will be pleased to know that the greedy algorithm is optimal for the set $\mathit{Coins} = \{25, 10, 5, 1\}$. The proof follows an ad hoc case analysis very similar to that conducted for European coins. Because the greedy algorithm is not always optimal, we explore another idea to solve the problem.

An optimal algorithm. The problem is to find the minimum number of coins required to reach sum S, with coins of value $\{v_1, \ldots, v_n\}$, which we denote as $z(S, n)$. Because the greedy algorithm may fail, we try to solve more subproblems so that we do not take a bad greedy choice as we did in the previous example. We also allow ourselves to come back to a choice already made and try another set of coins.

We investigate a way to solve the problem that is in appearance more complex than the initial problem. In other words, we artificially ask for more than requested, and aim at finding $z(T, i)$, the minimum number of coins required to reach sum $T \leqslant S$ with the first i coins, i.e., coins selected from the subset $\{v_1, \ldots, v_i\}$ (where $0 \leqslant i \leqslant n$). Instead of computing only $z(S, n)$, the original problem, we compute $S \times n$ values $z(T, i)$. But now, we have a recurrence relation to compute $z(T, i)$:

$$z(T, i) = \min \begin{cases} z(T, i-1) & i\text{-th coin not used;} \\ z(T - v_i, i) + 1 & i\text{-th coin used (at least) once.} \end{cases}$$

The recurrence must be properly initialized; values of i and T are decreasing, so we consider the cases $i = 0$ and $T \leqslant 0$:

- $z(T, 0) = +\infty$ for $T > 0$: There are no more coins, therefore, we cannot reach the sum $T > 0$ and this solution cannot be correct.
- $z(0, i) = 0$: We do not need any coin to reach the sum $T = 0$.

- $z(T, i) = +\infty$ for $T < 0$: We have exceeded the sum; this solution cannot be correct.

Thanks to the recurrence relation and the initialization conditions, we are now able to compute $z(S, n)$ and to solve the original problem. This kind of algorithm is called a *dynamic-programming* algorithm.

If the recurrence is applied without memoizing which values have already been computed, using a recursive algorithm, there will be an exponential number of computations. Note that the word *memoization* comes from "memo": the idea consists of memoizing the values so that we can look them up later.

However, we need to compute only $S \times n$ values of the function $z(T, i)$ ($1 \leqslant T \leqslant S$ and $1 \leqslant i \leqslant n$). This can be done either recursively, by memoizing the values that have already been computed, or iteratively, with, for instance, a loop with increasing i and then a loop with increasing T, so that we always have the values required to compute $z(T, i)$, i.e., $z(T, i-1)$ and $z(T - v_i, i)$, as shown in Algorithm 4.2. The precedence constraints are shown in Figure 4.1, and they are always enforced with this algorithm (for details about precedence constraints, see Section 6.4.4, p. 140). Note that we ensure that we never call the function with $T < 0$, and, therefore, we do not need the third initialization condition.

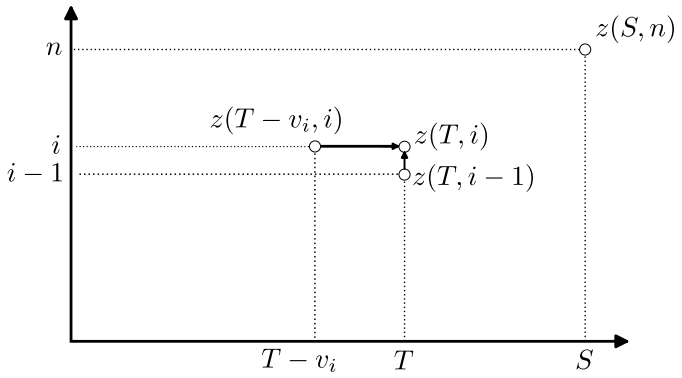


FIGURE 4.1: Precedence constraints for the coin-changing dynamic-programming algorithm.

The complexity of the dynamic-programming algorithm is $O(n \times S)$, while the greedy algorithm has a complexity in $O(n \log n)$ (the execution is linear, but sorting the coins requires a time in $O(n \log n)$).

Finally, note that characterizing the set of coins for which the greedy algorithm is optimal is still an open problem. It is easy to find sets that work. For instance, coins $\{1, B, B^2, B^3, \ldots\}$ with $B \geqslant 2$. However, the general case seems tricky. There are several variants of the coin changing problem, and

```
1 for T = 1 to S do
2 |  z(T,0) ← +∞      { Initialization: case i = 0 }
3 for i = 0 to n do
4 |  z(0,i) ← 0       { Initialization: case T = 0 }
5 for i = 1 to n do
6 |  for T = 1 to S do
7 |  |  z(T,i) ← z(T,i−1)
  |  |     { z(T,i−1) computed at previous iteration, or case i = 0 }
8 |  |  if T − v_i ⩾ 0 then
9 |  |  |  z(T,i) ← min(z(T,i), z(T − v_i, i))
  |  |  |     { z(T − v_i, i) computed earlier in this loop, or case T = 0 }
```

ALGORITHM 4.2: Dynamic-programming algorithm for the coin changing problem.

many dynamic-programming algorithms to solve them. The interested reader may refer to the following papers: [85, 98]. We move in the next section to another classical problem: The knapsack problem.

4.2 The knapsack problem

We have a set of items, each with a weight and a value, and we want to determine the items to include in the collection so that the total weight does not exceed a given limit, and the total value is as large as possible. Formally, there are n items $I_1, \ldots, I_n$, and item I_i has a weight w_i and a value c_i ($1 \leqslant i \leqslant n$). We are also given a maximum total weight W. The goal is to find a subset K of $\{1, \ldots, n\}$ that maximizes $\sum_{i \in K} c_i$, under the constraint $\sum_{i \in K} w_i \leqslant W$. The analogy with the problem of packing the best items for a well-deserved vacation should be clear.

Greedy algorithm. Here again, we start by designing a greedy algorithm to solve the problem. The idea consists in selecting first those items that have a good value per unit of weight, $\frac{c_i}{w_i}$. Therefore, we sort items by nonincreasing $\frac{c_i}{w_i}$, and then we greedily add them in the knapsack as long as the total weight is not exceeded.

However, the algorithm is not optimal because items are not divisible. We cannot take only a fraction of an item, i.e., either we take it or we discard it. A counterexample for the greedy algorithm can be designed as follows, with

three items. The first item with the greatest ratio c_1/w_1 is such that it fills up the knapsack by itself (no other item can fit in the knapsack once I_1 has been chosen, i.e., $w_1 + w_i > W$, for $i \geqslant 2$). Then, two more items are such that $w_2 + w_3 \leqslant W$ (they fit together in the knapsack), and $c_2 + c_3 > c_1$ (they have more value than the first item alone). If we are able to construct such an example, the greedy algorithm chooses the first item, while a better solution consists in choosing items 2 and 3. A possible set of items is the following, with $W = 10$: $(w_1 = 6, w_2 = 5, w_3 = 5)$ and $(c_1 = 7, c_2 = 5, c_3 = 5)$.

If we consider the problem of the fractional knapsack, in which it is possible to take only a fraction of an object, then the greedy algorithm is optimal. In the example, it would take the whole item 1, and then a fraction (4/5) of item 2 to fill the remaining space in the knapsack. The value would then be $c_1 + \frac{4}{5}c_2 = 7 + 4 = 11$, which is optimal. It is easy to prove the optimality of the greedy algorithm in this case. If an optimal solution is not making the greedy choice, we can always exchange a fraction of item of the optimal solution with the fraction of item of better value per weight unit that was not greedily chosen, and the total value can only increase.

Dynamic-programming algorithm. We come back to the integer knapsack problem, and since the greedy algorithm is not optimal, we try to solve a more complex problem, as in the coin changing problem, in order to be able to establish a recurrence. The two parameters are the total weight and the number of items considered. We want to compute $C(v, i)$, which is the maximum value that can be obtained when filling up a knapsack of maximum total weight v, using only some of the first i items $\{I_1, \ldots, I_i\}$. The original problem is the value $C(W, n)$: The knapsack is of maximum total weight W, and we have the n items at our disposal.

To write the recurrence, we have two choices: (1) either we have chosen the last object, or (2) we have not, therefore leading to:

$$C(v,i) = \max \begin{cases} C(v, i-1) & \text{last object not chosen;} \\ C(v - w_i, i-1) + c_i & \text{last object chosen;} \end{cases}$$

with the initialization conditions:

- $C(v, i) = 0$ for $v = 0$ or $i = 0$;
- $C(v, i) = -\infty$ if $v < 0$ (capacity exceeded).

The optimal solutions of all subproblems that we solve allow us to compute the optimal solution of the original problem. Similarly to the coin changing problem, we need to carefully respect the precedence constraints of the computations, and we want to never compute twice the same value of the function $C(v, i)$. The algorithm is formalized in Algorithm 4.3. The precedence constraints are shown in Figure 4.2. Because the computation is done row by row, these constraints are always respected.

The complexity of the greedy algorithm is in $O(n \log n)$, because the n items must be sorted. However, the complexity of the dynamic programming algo-

```
1 for i = 0 to n do
2 |  C(0,i) ← 0      { Initialization: case v = 0 }
3 for v = 1 to W do
4 |  C(v,0) ← 0      { Initialization: case i = 0 }
5 for i = 1 to n do
6 |  for v = 1 to W do
7 |  |  C(v,i) ← C(v,i−1)
8 |  |  if v − w_i ⩾ 0 then
9 |  |  |  C(v,i) ← max(C(v,i), C(v − w_i, i) + c_i)
```

ALGORITHM 4.3: dynamic-programming algorithm for the knapsack problem.

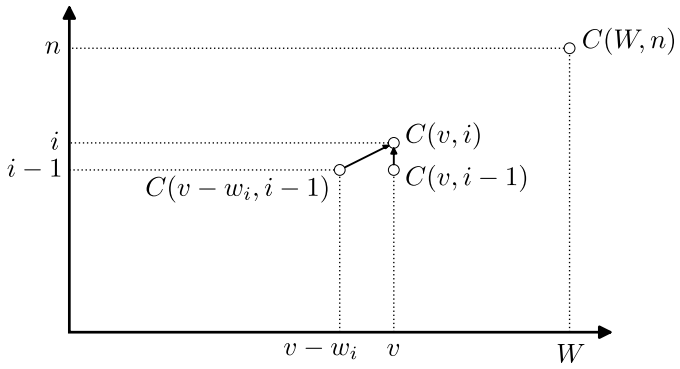


FIGURE 4.2: Precedence constraints for the knapsack dynamic-programming algorithm.

rithm is in $O(n \times W)$, because we need to compute $n \times W$ values of the function $C(v, i)$, and each computation takes constant time.

4.3 Designing dynamic-programming algorithms

In the previous two sections, we have given examples of dynamic-programming algorithms. The basic reasoning to obtain the optimal algorithm is similar in both cases:

1. Identify subproblems whose optimal solutions can be used to build an optimal solution to the original problem. Conversely, given an optimal

solution to the original problem, identify subparts of the solution that are optimal solutions for some subproblems. Usually, this step means that we identify a more complex problem derived from the original problem.

2. Write the recurrence.

3. Write the initial cases.

4. Write the algorithm, usually as an iterative algorithm, and taking care to enforce precedence constraints (use a figure to check that these constraints are indeed satisfied). A recursive algorithm may be used, but it requires tests to avoid redundant computations.

5. Study the complexity of the algorithm (usually straightforward from the iterative version of the algorithm).

Such an algorithm is *bottom-up*; we need results of the multiple subproblems to make a choice and compute the optimal solution, while the greedy algorithms were *top-down*, making a local choice at each step.

With dynamic-programming algorithms, one must be particularly cautious about the size of the data. It is not unusual to write nonpolynomial dynamic-programming algorithms. For instance, in the knapsack problem, the cost of the dynamic-programming algorithm is $O(nW)$. However, data can be encoded in $\sum_{i=1}^{n} \log w_i + \sum_{i=1}^{n} \log c_i \leqslant n(\log W + \log C)$, which means that W is in fact exponential in the problem size. This important encoding issue is related to weak NP-completeness and pseudo-polynomial algorithms, which we come back to in Section 6.6, p. 145.

4.4 Exercises

Exercise 4.1: Matrix chains (solution p. 90)

Consider n matrices $A_1, \ldots, A_n$, where A_i is of size $P_{i-1} \times P_i$ $(1 \leqslant i \leqslant n)$. We want to compute $A_1 \times A_2 \times \cdots \times A_n$. The problem is to decide in which order the multiplications should be done and, therefore, to add parentheses to the expression, in order to minimize the number of operations. Note that it costs $P_a \times P_b \times P_c$ to multiply a matrix of size $P_a \times P_b$ by a matrix of size $P_b \times P_c$.

Propose a dynamic-programming algorithm to solve the problem and give its complexity. Be careful to define the initial conditions and the recurrence.

Exercise 4.2: The library (solution p. 91)

The library is planning to move. It has a collection of n books $b_1, b_2, \ldots, b_n$. Book b_i has a width w_i and a height h_i. The books are stored on identical shelves of width L. Each shelf is used to store a set of books of consecutive indices. In other words, for each shelf, there exist two indices i and j such that the shelf exactly includes the books b_i, b_{i+1}, ..., b_{j-1}, b_j.

1. We assume first that all heights are identical: $h_i = h$, for $1 \leqslant i \leqslant n$, and we want to minimize the number of shelves that are used. Propose a greedy algorithm to solve the problem and prove that it is optimal.

2. Now, books have different heights, but we can adjust the distance between two shelves. The new objective criteria is the total space usage, defined as the sum of the heights of the higher book on each shelf. Give an example where the greedy algorithm of the previous question is no longer optimal, design an optimal algorithm to solve this problem, and give its complexity.

3. We come back to the problem with identical heights. Now, we want to place the n books on k shelves of same length L, and the objective is to minimize L, while k is fixed. In other words, we need to partition the n books into k sets, where the width of the widest set is as small as possible. Design an algorithm to solve the problem, and give its complexity in terms of n and k.

Exercise 4.3: Polygon triangulation (solution p. 93)

We consider planar convex polygons. A triangulation of a polygon is a set of lines that do not intersect inside the polygon and that divide the polygon into triangles. Here, the triangulation lines all pass through polygon vertices.

Let $P = \langle v_0, \ldots, v_n \rangle$ be a convex polygon, where $v_0, \ldots, v_n$ are the polygon vertices numbered in the direct order, and let w be a weight function defined on the triangles formed by the sides and the lines drawn in P. For instance, $w(i, j, k)$ can be the perimeter of the triangle defined by the vertices v_i, v_j, and v_k. The problem is to find a triangulation that minimizes the sum of the weight of the triangles induced by the triangulation.

1. For $1 \leqslant i < j \leqslant n$, we define $t(i, j)$ as the weight of an optimal triangulation of the polygon $\langle v_{i-1}, \ldots, v_j \rangle$, with $t(i, i) = 0$ for $1 \leqslant i \leqslant n$. Express a recurrence to compute t, derive an algorithm to solve the problem, and give its complexity.

2. If the weight function can be anything, how many values do we need to know for the function to be defined on all polygon triangles? Compare with the complexity of the algorithm.

3. If the weight of a triangle is equal to its surface, what can you say about the algorithm that you have designed?

Exercise 4.4: Square of ones (solution p. 96)

Given a matrix A of size $n \times m$ with coefficients in $\{0, 1\}$, we want to find the maximum width K of a square of ones in A, as well as the coordinates (I, J) of the top left corner of such a square. In other words, for all i, j such that $I \leqslant i \leqslant I + K - 1$ and $J \leqslant j \leqslant J + K - 1$, we have $A[i, j] = 1$.

1. Design a dynamic-programming algorithm to solve this problem.
2. What is the complexity of your algorithm?

(*Hint:* Consider $t[i, j]$, the width of the biggest square of ones whose top left corner is (i, j).)

Exercise 4.5: The wind band (solution p. 98)

In a wind band, there are n musicians of size $t_1, t_2, \ldots, t_n$. For concerts, the orchestra has m suits ($m \geqslant n$) of size $u_1, u_2, \ldots, u_m$. Every year, some musicians leave the band and they are replaced by new ones, and we need to give each musician a suit of appropriate size: $\alpha(i)$ is the index of the suit given to the musician of size t_i.

1. Yves, the drum player, believes that the objective is to minimize the average difference between the size of a musician and the size of their suit, i.e., minimize $\frac{1}{n}\sum_{i=1}^{n} |t_i - u_{\alpha(i)}|$. He proposes a greedy algorithm. We find i and j such that $|t_i - u_j|$ is minimum, we give the suit of size u_j to the musician of size t_i, and we iterate until everybody received a suit. Is this algorithm optimal?
2. Anne, the horn player, believes that it is more fair to minimize the average square of differences: $\frac{1}{n}\sum_{i=1}^{n}(t_i - u_{\alpha(i)})^2$. Show on an example the advantage of this objective function, compared to Yves's. Is the greedy algorithm optimal for this objective function?
3. If there are as many suits as musicians (i.e., $n = m$), then design an optimal algorithm for Anne's objective function.
4. Design an optimal algorithm for the general case $m \geqslant n$ (and Anne's objective function).

Exercise 4.6: Ski rental (solution p. 98)

The problem is to distribute m pairs of skis of lengths $s_1, \ldots, s_m$ to n persons of size $h_1, \ldots, h_n$, all wanting to go skiing. We assume that there are enough

skis in the rental shop for everybody (i.e., $m \geqslant n$). The allocation is defined by an injective function $f : \{1, \ldots, n\} \to \{1, \ldots, m\}$, and f is optimal when it minimizes $A(n, m) = \sum_{k=1}^{n} |s_{f(k)} - h_k|$.

1. Design an efficient algorithm that returns an optimal allocation of the skis.

   (*Hint:* Prove that the tallest person can be allocated the longest pair of skis used.)

2. What is the complexity of the algorithm? You should refine the analysis to guarantee that the algorithm is in $O(n \log n)$ if $m = n$.

3. Prove that we can obtain a better complexity when $n^2 = o(m)$.

   (*Hint:* Restrict to $O(n^2)$ pairs of skis.)

Exercise 4.7: Building set (solution p. 102)

We want to build a tower as high as possible, from a set of bricks. We have n different types of bricks, and as many bricks of each type as we want. The brick of type i is a parallelepiped of size $\{x_i, y_i, z_i\}$, and it can be oriented in any way, two dimensions being the base of the brick, and the third one being the height. When building the tower, a brick can be placed on top of another only if the two dimensions of its base are *strictly* smaller than the dimensions of the brick on which we want to place it.

1. Design an optimal dynamic-programming algorithm to build a tower of maximum height.

2. What is the complexity of this algorithm?

4.5 Solutions to exercises

Solution to Exercise 4.1: Matrix chains

We want to compute $A_1 \times \cdots \times A_n$. Let us look for the optimal cost of computing the product $A_i \times \cdots \times A_j$. We denote this cost by $C(i, j)$. The optimal solution for the problem will be obtained for $i = 1$ and $j = n$.

We define $C(i, j)$ by induction. We partition in two the product of matrices $(A_i, \ldots, A_j)$ to indicate which two matrices were multiplied in the last matrix multiplication. In other words, if we cut $(A_i, \ldots, A_j)$ after the position k, this means that the last multiplication was between matrix $A_i \times A_{i+1} \times \cdots \times A_k$ and matrix $A_{k+1} \times \cdots \times A_j$. Let us assume that the optimal solution was to cut $(A_i, \ldots, A_j)$ after the matrix A_k. Then the optimal cost to compute

Chapter 5

Amortized analysis

In this chapter, we briefly discuss amortized analysis, the goal of which is to average the cost of n successive operations. This should not be confused with the average cost of an operation. We first describe the three classical methods with examples (Section 5.1) and then proceed with exercises in Section 5.2, with solutions in Section 5.3.

5.1 Methods for amortized analysis

First, we introduce two examples to illustrate the methods used to conduct an amortized analysis. Then, we present the three classical methods: aggregate analysis, the accounting method, and the potential method.

5.1.1 Running examples

The first example is a k-bit counter that we want to increment. Initially, the counter has a value of 0, and each operation increments it. Formally, this counter is represented by an array A of k bits, where $A[i]$ is the $(i+1)$-th bit, for $0 \leqslant i \leqslant k-1$. A number x represented by this counter is such that $x = \sum_{i=0}^{k-1} A[i].2^i$. For instance, if $k = 6$ and if we perform $n = 4$ operations, we obtain the following sequence:

0 0 0 0 0 0
0 0 0 0 0 1
0 0 0 0 1 0
0 0 0 0 1 1
0 0 0 1 0 0

The cost of an increment is defined as the number of bits that should be modified. This cost is not constant for each value of the counter; it is equal to the number of successive 1s at the right of the counter, plus 1 (switching the first 0 to 1).

The second example consists of inserting n elements in a table, dynamically, starting from an empty table. We insert a new element directly in the table if

there is space, with a cost 1. Otherwise, we create a new table that has twice the size of the original table (or a table of size 1 for the first insertion); we copy the content of the original table and insert the new element. The cost is then the size of the original table plus 1. Note that the table is always at least half full (an empty table is considered full), so even if the cost may be high for some operations, we then have free space for the next operations.

For both examples, the amortized analysis consists of asking the following question: What is the cost of n successive operations?

5.1.2 Aggregate analysis

The goal of this method is to show that the cost of n successive operations can be bounded by $T(n)$. Therefore, in the worst case, the cost per operation on average, i.e., the amortized cost per operation, is bounded by $T(n)/n$.

For the k-bit counter, it is obvious that the cost of n successive increment operations is bounded by nk. However, this upper bound can be improved. Indeed, the right-most bit flips each time, the second one flips every second time, and so on. Therefore, the cost of n operations is at most $n+\frac{n}{2}+\frac{n}{4}+\cdots \leqslant 2n$, regardless of the value of k. This leads to an amortized cost per operation of 2.

For the table insertion, for any integer $k \geqslant 0$, the cost of the (2^k+1)-th insertion is $c(i) = 2^k + 1$, i.e., the size of the table is doubled. Otherwise, the cost is $c(i) = 1$ (including the cost of the first insertion, $c(1)$). Therefore, we have:

$$\sum_{i=1}^{n} c(n) \leqslant n + \sum_{k=0}^{\lfloor \log_2(n) \rfloor} 2^k \leqslant 3n$$

and an amortized cost of 3.

5.1.3 Accounting method

The principle of this method is to pay in advance for costly operations that may happen afterwards, hence, keeping a constant cost per operation. One has to guarantee that at the time of operation i, one has enough credit (including advance payment and the payment for the operation) to cover the cost of the operation.

For the k-bit counter, each time we flip a bit from 0 to 1, we decide to pay 2 euros[1]: 1 euro for the flip and another one so that we will be able to flip back the bit from 1 to 0 without having to pay. For this example, since at each increment there is only one bit to flip from 0 to 1, the cost is 2 at each

[1] Yes, \$2 would be okay, too.

increment, and, hence, an upper bound of $2n$ for n operations (note that we may have paid for some operations that have not been done yet).

For the table insertion, we decide to pay €3 at each insertion: €1 is used to pay for the insertion, a second one will be used to pay for the transfer of the element when a new table will be required, and a third one is assigned to an element in the first half of the table that also will need to be transferred later when the table is full. Therefore, each time the size of the table is doubling, we can transfer all elements at no cost. This leads to an upper bound of $3n$.

5.1.4 Potential method

This last method consists of representing the prepaid work of the accounting method by a potential that can be used to pay for future operations. The prepaid work of the accounting method is no longer associated with objects but rather with the data structure itself. We define a potential function, which associates to each data structure a potential. This potential function should always be greater or equal to the potential function of the data structure before the first operation, so that there is always enough potential to pay for an operation. We introduce the following notations:

- Φ_0 is the potential before the first operation.
- $\Phi_i \geqslant \Phi_0$ is the potential of the data structure after i operations.
- c_i is the cost of operation i.
- $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$ is the amortized cost of operation i. A costly operation may have a small amortized cost if the potential function has decreased with operation i, i.e., $\Phi_i - \Phi_{i-1} < 0$.

Therefore, the amortized cost of a sequence of n operations can be computed as:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0.$$

Because $\Phi_n - \Phi_0 \geqslant 0$, the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ gives an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$. Note that we often define Φ so that $\Phi_0 = 0$ and $\Phi_i \geqslant 0$, for convenience.

For the k-bit counter, Φ_i is the number of bits that are at value 1 after operation i. This number is always positive or null, and it is initially null. Let $t(i)$ be the number of right-most successive 1s just before operation i. The potential after operation i is, therefore, $\Phi_i = \Phi_{i-1} - t(i) + 1$ because $t(i)$ 1s have been reset to 0, and one 0 has taken the value 1. Moreover, the cost of operation i is $c_i = t(i) + 1$:

$$\begin{array}{ll} \cdots 0\ 1 \cdots 1 & \Phi_{i-1} \\ \cdots 1\ \underbrace{0 \cdots 0}_{t(i)} & \Phi_i = \Phi_{i-1} - t(i) + 1. \end{array}$$

Therefore, the amortized cost of operation i is $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = t(i) + 1 + (-t(i) + 1) = 2$.

For the table insertion, the potential can be seen as the richness of the table; a table is rich when it is full. The table potential equals twice the number of elements in the table minus the size of the table. Because the table is always at least half full, this value cannot be negative. Formally, let num_i be the number of elements after i operations, and let $size_i$ be the size of the table after i operations. Initially, $num_0 = size_0 = \Phi_0 = 0$, and the potential function is expressed as $\Phi_i = 2num_i - size_i \geqslant 0$. Because we perform only insertions, $num_i = num_{i-1} + 1 = i$.

If the size of the table remains identical after operation i, we have $size_i = size_{i-1}$ and $c_i = 1$. Therefore, $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = 1 + 2 = 3$. However, if $size_i = 2size_{i-1}$, this means that the cost of the operation was $c_i = num_i$ and that the table was full after operation $i - 1$, i.e., $size_{i-1} = num_{i-1}$, and, therefore, $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = num_i + 2 - size_{i-1} = 3$.

5.2 Exercises

Exercise 5.1: Binary counter (solution p. 112)

Consider the running example of the k-bit counter introduced in Section 5.1.1 with an *increment* function.

1. Show that if we had also a *decrement* function on the counter, then a sequence of n operations could have a cost in $\Theta(nk)$.

2. We keep only the *increment* function, and we add a *reset* operation that resets the counter to its initial value 0. Show how to implement this counter as a table of bits so that any sequence of n operations (increment or reset) takes a time $O(n)$ on a counter with initial value 0, with the accounting method.

   (*Hint:* Keep a pointer to the highest-order 1.)

Exercise 5.2: Inserting and deleting (solution p. 113)

Consider the running example of table insertion introduced in Section 5.1.1. We consider now that it is also possible to delete elements from the table.

1. If we double the size of the table when it is full, and we halve the size of the table when it is less than half empty, what would be the amortized cost?

2. Propose an implementation of the insert and delete functions with a constant amortized cost. Apply the accounting method and then the potential method to compute the amortized cost.

Exercise 5.3: Stack (solution p. 114)

We consider a stack with the following operations: $push(S, x)$ pushes object x onto stack S, $pop(S)$ pops the top of the stack and returns the popped object (it returns an error if the stack is empty), and, finally, $multipop(S, k)$ removes the k top objects of the stack, and the entire stack if it contains fewer than k objects (it tests at each step whether the stack is empty). Initially, the stack is empty.

1. What is the time complexity of each of these operations? Use the aggregate analysis to obtain the amortized cost of a sequence of n operations.

2. Use the accounting method to analyze the amortized cost.

3. Use the potential method to analyze the amortized cost.

4. Propose an implementation of a first-in first-out queue with two stacks, such that adding an element in the queue and removing an element from the queue both have an amortized cost of $O(1)$.

Exercise 5.4: Deleting half the elements (solution p. 115)

We want to implement a data structure S with real numbers, with the following operations: $insert(S, x)$ inserts the object x in S, and $delete(S)$ removes the $\lceil |S|/2 \rceil$ largest elements of S. Propose an implementation such that the amortized cost of both operations is cost.

(*Hint:* You can find in linear time the median of a list, see Section 9.3 of [27].)

Exercise 5.5: Searching and inserting (solution p. 116)

We consider a data structure for n elements. Let $k = \lceil \log(n+1) \rceil$, and let $(n_{k-1}, n_{k-2}, \ldots, n_0)$ be the binary representation of n. The data structure consists of k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, and the size of A_i is 2^i for $0 \leqslant i \leqslant k-1$. The array A_i is full if $n_i = 1$, and empty otherwise, so that the total number of elements is $n = \sum_{i=0}^{k-1} n_i 2^i$. Note that each individual array is sorted, but there is no particular relationship between elements of two different arrays.

1. Propose a *search* operation for this data structure (find if an element is in the data structure), and analyze its worst-case running time.

2. Propose an *insert* operation for this data structure (insert a new element in the data structure), and analyze its worst-case and amortized running times.

3. Discuss how to implement a *delete* operation.

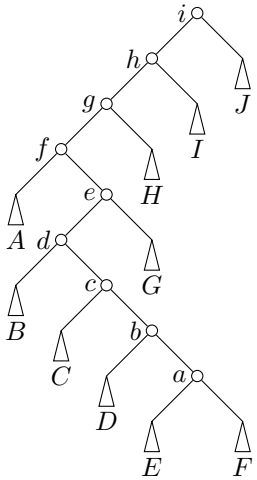
4. Compare the costs achieved by this data structure with the costs of searching and inserting in a sorted array of size n.

Exercise 5.6: Splay trees (solution p. 117)

The problem is to perform a sequence of m access operations on a set of n elements that are totally ordered. The elements are represented as a binary search tree: There is one element per node, and for any node x, all the elements in the left subtree of x are smaller that x, while all the elements in the right subtree of x are greater than x. The operation $access(i)$ is then in $O(d)$, where d is the depth of node x containing element i. In order to reduce the total access cost in a sequence of n accesses, we aim at moving frequently accessed elements toward the root. Therefore, each time any element x is accessed, we use the *splaying* heuristic. We repeat the following splaying steps until x is the root of the tree (see Figure 5.1).

- **zig**: If $p(x)$, the parent of x, is the tree root, rotate the edge joining x with $p(x)$ (this case is terminal).
- **zig-zig**: If $p(x)$ is not the root and x and $p(x)$ are both left or both right children, rotate the edge joining $p(x)$ with $p(p(x))$, and then rotate the edge joining x with $p(x)$.
- **zig-zag**: Otherwise, rotate first the edge joining x with $p(x)$ and then the edge joining x with the new $p(x)$ (that was initially $p(p(x))$).

1. Apply the splaying heuristic on node a of the tree below:



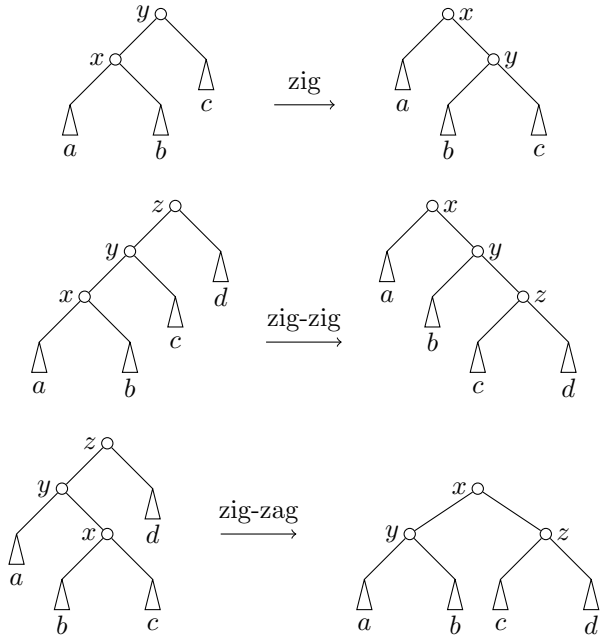


FIGURE 5.1: The different splaying steps, where x is the accessed element.

2. What is the time complexity of the splaying heuristic, in terms of number of rotations?

3. To analyze the amortized complexity of splaying, we use a potential function defined as follows: We assume that each element i has a positive weight $w(i)$, whose value is arbitrary but fixed. The size $s(x)$ of a node x is the sum of the weights of all elements in the subtree rooted in x, and the rank of x is $r(x) = \log s(x)$. The potential of a tree is the sum of the ranks of all its nodes. The cost of an operation is the number of rotations, but we still charge 1 if there is no rotation.

   Let $r(x)$ (resp. $r'(x)$) be the rank of x after (resp. before) the operation. Show that the amortized cost of a zig is at most $1+3(r'(x)-r(x))$ and the amortized cost of a zig-zig or a zig-zag is at most $3(r'(x)-r(x))$. (Note that if $a, b > 0$, $a + b \leqslant 1$, then $\log(a) + \log(b) \leqslant -2$.)

4. Deduce that the amortized time to splay a tree with root t at a node x is at most $3(r(t) - r(x)) + 1 = O(\log(s(t)/s(x)))$. Note that this is true for any positive weights.

5. Prove that the total access time is $O((m + n) \log n + m)$ (recall that n is the number of elements in the tree, and m is the number of accesses). Hint: Assign a weight to each element.

6. For any element i, $q(i)$ is the access frequency of i, i.e., the total number of times i is accessed (within the m accesses). Show that if every element is accessed at least once, then the total access time is $O\left(m + \sum_{i=1}^{n} q(i) \log\left(\frac{m}{q(i)}\right)\right)$. Hint: Assign a weight to each element.

Exercise 5.7: Half perimeter of a polygon (solution p. 119)

We consider a polygon with n vertices, numbered in the clockwise order from 0 to $n-1$. The edge from i to $i+1 \mod n$, for $0 \leqslant i < n$, has a length a_i.

1. We aim at finding the two vertices i and j that minimize the absolute value of the difference between the two portions of perimeters that they define, i.e., that minimize (the sums are modulo n) $\left|\sum_{l=i}^{j-1} a_l - \sum_{l=j}^{i-1} a_l\right|$.

   (a) Design a naive algorithm and give its complexity.

   (b) Design a linear-time algorithm.

2. Find in linear time the three vertices i, j, and k that minimize the difference between the larger *third* and the smaller *third* portions of the perimeter that they define, i.e.,
   $\max\left(\sum_{l=i}^{j-1} a_l, \sum_{l=j}^{k-1} a_l, \sum_{l=k}^{i-1} a_l\right) - \min\left(\sum_{l=i}^{j-1} a_l, \sum_{l=j}^{k-1} a_l, \sum_{l=k}^{i-1} a_l\right)$.

5.3 Solutions to exercises

Solution to Exercise 5.1: Binary counter

1. After $2^{k-1} - 1$ operations, the value of the counter is $0\ 1\ 1\ \cdots\ 1$. If we perform a sequence of operations (increment, decrement), the counter will alternate between $1\ 0\ 0\ \cdots\ 0$ and $0\ 1\ 1\ \cdots\ 1$, hence, having a cost k for each operation, and for n operations, a cost in $\Theta(nk)$.

2. We introduce a new variable, max_A, that contains the index of the highest-order 1 in the counter A. Initially, $max_A = -1$ because there are only 0s in the counter. This value is updated at each operation, see Algorithms 5.1 and 5.2, where $|A| = k$.

Chapter 6

NP-completeness

In this chapter, we introduce the complexity classes that are of paramount importance for algorithm designers: P, NP, and NPC. We take a strictly practical approach and determinedly skip the detour through Turing machines. In other words, we limit ourselves to NP-completeness, explaining its importance and detailing how to prove that a problem is NP-complete.

After introducing our approach in Section 6.1, we define the complexity classes P and NP in Section 6.2. NP-complete problems are introduced in Section 6.3, along with the practical reasoning to prove that a problem is NP-complete. Several examples are provided in Section 6.4. We discuss subtleties in problem definitions in Section 6.5 and strong NP-completeness in Section 6.6. Finally, we make our conclusions in Section 6.7.

6.1 A practical approach to complexity theory

This chapter introduces the key complexity classes that algorithm designers are confronted with: P, which stands for *Polynomial*, and NP, which stands for *Nondeterministic Polynomial*. In fact, we depart from the original definition of the class NP and use the (equivalent) characterization *Polynomial with Certificate*. Within the NP class, we focus on the subclass NPC of *NP-complete* problems.

When writing this chapter, we faced a cruel dilemma. Either we use a formal approach, which requires an introduction to Turing machines, explain their characteristics, and classify the languages that they can recognize, or we use a practical approach that completely skips the detour through the theoretical computer science framework and defines complexity classes out of nowhere (almost!). We firmly believe that there is no trade-off in between, and that a comprehensive exposure does require Turing machines. However, given the main objectives of this book, we chose the latter approach. The price to pay is that the reader will have to take for granted a key result, namely Cook's theorem [25], which we will state without proof. Cook's theorem provides the first NP-complete problem, and we will have to trust him on this. However, the main advantage is that we can concentrate on the art of the algorithm

designer, namely *polynomial reduction*.

First, why Turing machines? To assess the complexity of a problem, we need to define its size and the number of time steps required to solve it. But what is appropriate within a time step? A formal answer relies on Turing machines. The size of a problem is the number of consecutive positions used to store its data on the (infinite) ribbon of the machine. The number of time steps is the number of moves before the Turing machine terminates the execution of its program, given the data initially stored on its ribbon. Instead, in the practical approach, we simply define the size of a problem as the number of memory locations, or bits, that are needed to store its data, and we define a time step as the maximum time needed to execute an elementary operation. Here, an elementary operation is defined as any *reasonable* computation. And, the trouble begins. Fetching the values of two memory locations, adding them and storing them back into some memory location, is that an elementary operation? Yes—well, provided that the access to the memory locations takes constant time, which may require that the total memory is bounded, or at least that two different memory locations used to solve the problem are not too far apart in storage. We are not far from moving the head of the Turing machine from one position to another! Similarly, adding two bits or two bytes or two double-precision floating point numbers (64 bits) is indeed an elementary operation, but adding two integers of unbounded length is not. In fact, an elementary operation is anything that can be done in polynomial time by a Turing machine, but this statement is helpful mostly to those who are familiar with Turing machines. Here is an example of an operation that is not reasonable. If we have two prime numbers p and q of r bits, we can compute their product $n = p \times q$ in $O(r^2)$, but given n, we cannot find p and q in time polynomial in r.

We refer the reader interested in the formal approach to some excellent books. The big classic is the book by Garey and Johnson [38] with a comprehensive treatment of NP-completeness. A very intuitive proof of Cook's theorem is given by Wilf [108]. More on complexity theory is provided by Papadimitriou [82].

6.2 Problem classes

In this section, we first emphasize the importance of polynomials in the theory. Then, we discuss how to define the problem size and how to encode data. This is illustrated through classical examples; integers are coded in a logarithmic size, but we should be careful if objects must be enumerated (set of nodes in a graph, list of tasks, etc).

6.2.1 Problems in P

The following remark, admittedly simple, is fundamental: *The composition of two polynomials is a polynomial.* Thanks to this observation, key values (time, size) can be defined up to a polynomial factor. From the point of view of complexity classes, values like n, n^3, or $n^{27} + 17n^5 + 42$ are totally equivalent; all these values are polynomial in n. Hence, there is no difference if an elementary operation of the algorithm would cost n^3 or $n^{27}+17n^5+42$ time steps of a Turing machine; as long as there is a polynomial number of such operations, the total number of time steps for the Turing machine remains polynomial.

The theory deals with decision problems, with a yes/no answer, rather than with optimization problems (this is related to languages that are accepted by Turing machines). A decision problem is in the complexity class P if it can be solved in *polynomial time*. Owing to the previous remark, we do not need to specify the degree of the polynomial, which is not relevant as far as theory is concerned (we come back to this last point below). Hence, the key for understanding this class P is the notion of "polynomial time." As mentioned before, one must decide what can be done within one unit of time. One usually assumes that one can add, multiply, or access memory in constant time, but the multiplication of large numbers (respectively, memory accesses) can depend on the size of the numbers (respectively, of the memory).

From an algorithmic point of view, we usually suppose that we can add, multiply, access memory within one unit of time, as long as numbers and memory size are bounded, which seems reasonable. These operations are then of polynomial time, and thus this model is polynomial with respect to the theoretical one with the Turing machine, as long as we are careful when dealing with nonbounded integers.

Also, the resolution time must be a polynomial of the *data size*, so one needs to define this "data size" carefully. This data size can strongly depend on the way an instance is encoded. Intuitively, integers can be coded in binary, therefore requiring a logarithmic size rather than a linear one (when encoded in unary). The encoding with any other basis $b \neq 2$ has the same size as the binary encoding, up to a constant factor ($\log_2(n)/\log_b(n) = 1/\log_b(2)$). However, some integers describing a problem instance should not be encoded in binary when they code objects to be enumerated. Otherwise, some "elementary" operations would have a cost exponential in the data size. We illustrate this by detailing two problem examples.

Example: 2-partition

DEFINITION 6.1 (2-PARTITION)**.** Given n positive integers $a_1, \ldots, a_n$, is there a subset I of $\{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?

The input data of a problem instance is a set of n integers. In theory, these n integers could be encoded either in unary or in binary. However, by

convention, in complexity theory, any integer appearing in the coding of an instance must be encoded in binary. The only exception is for data whose encoding in unary would not change the overall data size of the instance (i.e., if the new data size remains polynomial in the original data size). For instance, for 2-PARTITION, the choice of encoding for the value n itself does not matter because encoding n integers requires a data size of at least one per integer and thus of at least n. Therefore, for the sake of simplicity, one usually encodes n in unary. Then, with the mandatory binary encoding of the n integers, the data size is $\sum_{1 \leqslant i \leqslant n} \log(a_i)$. With a unary encoding of the integers, the data size of an instance would have been $\sum_{1 \leqslant i \leqslant n} a_i$.

The choice of the encoding is vital for such a problem. Indeed, one can find an algorithm whose time is polynomial in $n \times \sum_{1 \leqslant i \leqslant n} a_i$. We design a simple dynamic-programming algorithm; we solve the problems $c(i, T)$, where $c(i, T)$ equals true if there is a subset of $\{a_1, \ldots, a_i\}$ of sum T (and false otherwise), for $1 \leqslant i \leqslant n$ and $0 \leqslant T \leqslant S = \sum_{1 \leqslant i \leqslant n} a_i$. The solution to the original problem is $c(n, \frac{S}{2})$. The recurrence relation is $c(i, T) = c(i-1, T-a_i) \vee c(i-1, T)$. This algorithm is in $O(nS)$. Therefore, this algorithm runs in a time that would be polynomial in the size of the data if we had allowed the integers to be coded in unary. However, the algorithm running time is exponential in the data size when integers are coded in binary, as mandated. Such an algorithm is said to be *pseudopolynomial*.

No one knows an algorithm that is polynomial in the data size (i.e., in $O(n \log(S))$), so the question whether the 2-PARTITION problem is or isn't in P is left open.

Example: Bipartite graphs

DEFINITION 6.2 (BIPARTITE)**.** Given a graph $G = (V, E)$, is G a bipartite graph?

This is a decision problem; the answer must be yes or no. The input data of a problem instance is a graph (V, E), where V is the set of vertices and E the set of edges. The size of the data depends on how the graph is stored (or encoded). The graph consists of $|V| = n$ vertices. One usually codes n in unary rather than in binary. Independent vertices are vertices that are not endpoints of any edge. Independent vertices play a trivial role with respect to the problem, and they can therefore be safely discounted. Then, each vertex is the endpoint of at least one edge, $|E| \geqslant n/2$, and encoding n in binary does not change the data size. Therefore, for the sake of simplicity, in any graph problem the number of vertices is always encoded in unary for the same reason.

Then, the identifier of a vertex can be encoded in binary, thus in $\log(n)$ for one vertex leading to a total of $n \log(n)$, which is still polynomial in n. The number of edges is also polynomial in n because there are at most n^2 edges. Then, altogether, the total size of the problem data is a polynomial

in n, where n is the number of vertices of the graph. When designing graph algorithms, one often denotes the size of data of a graph as $|V| + |E|$ (strictly speaking, it should be $|V| + |E| \log(|V|)$, but each expression is polynomial in the other one). This allows us to refine the cost study of the algorithms, in particular when $|E| \ll |V|^2$. However, $|V| + |E|$ is still polynomial in n, so this refinement does not alter the problem classification.

Now, given a graph, in order to answer the question (yes or no), we need to perform a number of operations that is polynomial in n (greedy graph coloring). This problem is, therefore, in the complexity class P because it can be solved in a time that is polynomial in the data size.

6.2.2 Problems in NP

To define the complexity class NP, we need to define the *certificate* of a problem, which is (an encoding of) a solution to the problem.

Problem solution: Certificate

Back to the 2-PARTITION problem, if we are given a subset $I \subseteq \{1, \ldots, n\}$, we can check in polynomial time (even in linear time) whether $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$, and, therefore, we can answer whether the problem has a positive answer in polynomial time. Moreover, the size of the certificate I is $O(n \log(n))$, which is polynomial in the problem size (the certificate contains $O(n)$ identifiers, each coded on $O(\log(n))$ bits).

Another way to provide a solution to 2-PARTITION would be to give the certificate $\{a_i\}_{i \in I}$, but if the a_is are coded in unary in the certificate, it is of exponential size. However, if the a_is are coded in binary, then the certificate has polynomial size and it is perfectly acceptable; a certificate is valid if it is polynomial in the problem size.

For the bipartite graph problem, the certificate would be the set of indices of vertices of one of the two subsets of the graph, whose size is polynomial in the problem size. Given this set, it is then easy to check that it is a correct solution by looking at each edge of the graph, which takes a polynomial time.

(See also Section 6.4.4 on scheduling problems for an illustration of the care that must be taken to design a certificate of polynomial size.)

Definition of NP

We are now ready to define the problem class NP. This is the class of decision problems for which we can verify a certificate in a time that is polynomial in the problem size. By *verify*, we mean *check that the certificate is indeed a solution*, i.e., that the answer to the problem is *yes*. Both previous examples are, therefore, in NP because, if we are given a certificate of polynomial size, we can check in polynomial time whether it is a solution to the problem.

We make a short digression to explain that NP stands for *Nondeterministic Polynomial*, for reference to nondeterministic Turing machines that were

originally used to define the class. As already mentioned, we define NP as *Polynomial with Certificate* in this book, and we ignore equivalent characterizations of the NP class, either older (nondeterministic Turing machines) or newer (the famous PCP theorem). (See [4] for more information.)

It is time to recapitulate; we have defined two classes of decision problems:

P: Given an instance I of the problem of size $|I|$ when encoded in binary, there is an algorithm whose running time is polynomial in $|I|$ and which reports whether the instance has a solution or not;

NP: Given an instance I of the problem of size $|I|$ when encoded in binary, and a certificate of size polynomial in $|I|$, there is an algorithm whose running time is polynomial in $|I|$ and which reports whether the certificate is indeed a solution to the instance.

We observe that P $\subseteq$ NP. If we can find a solution in polynomial time, then we can verify the solution in polynomial time, with an empty certificate. Most researchers believe that the inclusion is strict, i.e., P $\neq$ NP, because it should be easier to check whether a certificate is a solution to the problem than to find a solution to that problem. As you may have heard before, this question is open at the time of this writing.

We have already seen that BIPARTITE is in P; therefore, it is in NP. Also, 2-PARTITION is in NP, but we do not know whether it is in P or not. Below are a few more examples to illustrate the class NP.

Examples: Problems in NP

DEFINITION 6.3 (COLOR)**.** Given a graph $G = (V, E)$ and an integer k ($1 \leqslant k \leqslant |V|$), can we color G with at most k colors?

This is a graph coloring problem; two vertices connected with an edge cannot be assigned the same color. The size of the data is a polynomial in $|V| + \log(k)$. Indeed, we need to enumerate all vertices similarly to the problem bipartite, hence the term $|V|$, while the integer k is encoded in binary. Since $k \leqslant |V|$ (one never needs more colors than vertices), the size of the data is a polynomial in $|V|$. A certificate can be the list of the vertices together with their color, whose size is linear in the size of the problem instance. The verification would amount to checking that no two adjacent vertices are assigned the same color, and that no more than k colors are used in total, which can be done in linear time as well.

DEFINITION 6.4 (HC – Hamiltonian Cycle)**.** Given a graph $G = (V, E)$, is there a cycle that goes through each vertex once and only once?

Similarly to other graph problems, the size of the data is a polynomial in $|V|$. A certificate can be the ordered list of the vertices that constitute the cycle (with linear size again). As before, the verification is easy: Check that the cycle is built with existing edges in the graph, and that each vertex is visited once and only once.

DEFINITION 6.5 (TSP – Traveling Salesman Problem)**.** Given a complete graph $G = (V, E)$, a cost function $w : E \to \mathbb{N}$ and an integer k, is there a cycle $\mathcal{C}$ going through each vertex once and only once, with $\sum_{e \in \mathcal{C}} w(e) \leqslant k$?

This classical traveling salesman problem is a weighted version of the HC problem. There are several variants of the problem with various constraints on the cost function w: The weights can be arbitrary, satisfy the triangular inequality, or correspond to the Euclidean distance. The variants do not change the problem complexity. The size of the data is a polynomial in $|V| + \sum_{e \in E} \log(w(e)) + \log(k)$. We need to enumerate vertices, and other integers are coded in binary. A certificate can be the ordered list of the vertices that constitute the cycle, and the verification is similar to that for the HC problem.

No one knows how to find a solution to these three problems in polynomial time.

Problems not in NP?

One rarely encounters a problem whose membership status, with respect to NP, is unknown. It is even rarer to come across a problem that is known not to belong to NP. These problems are usually not very interesting from an algorithmic point of view. They are, however, fundamental for the theory of complexity. We provide a few examples below.

Negation of TSP: Given a problem instance of TSP, is it true that there is no cycle in the graph of length $|V|/2$?

This problem is similar to TSP, but the question is asked in the reverse way. It is difficult to think of a certificate of polynomial size that would allow us to check in polynomial time that the answer to the question is yes. Whether this problem belongs to NP is an open question.

Square: Given n squares whose areas sum up to 1, can we partition the unit square into these n squares?

We are interested in this problem because it plays a prominent role in the case study of Chapter 14. Its complexity depends on the exact definition that is used. First, we give the variant of the problem that is used in Chapter 14; we are given n squares of size a_i, with $\sum_{1 \leqslant i \leqslant n} a_i^2 = 1$. The a_i are rational numbers, $a_i = b_i/c_i$, and the problem size is $\sum_{1 \leqslant i \leqslant n} \log(b_i) + \sum_{1 \leqslant i \leqslant n} \log(c_i)$. A certificate can be the position of each square, for $1 \leqslant i \leqslant n$, for instance the coordinates of its top left corner. This certificate is of polynomial size. We can then check in polynomial time whether it is a solution of the problem or not (however, writing such a verification procedure requires some care). Hence, this variant is in NP.

Another variant consists of having as input m_i squares of size a_i, for $1 \leqslant i \leqslant p$, with $n = \sum_{1 \leqslant i \leqslant p} m_i$ and $\sum_{1 \leqslant i \leqslant p} m_i a_i^2 = 1$. The size of the data is then $n + \sum_{1 \leqslant i \leqslant p} \log(m_i) + \sum_{1 \leqslant i \leqslant p} \log(b_i) + \sum_{1 \leqslant i \leqslant p} \log(c_i)$. We do not need to enumerate all squares but only the p basic squares, while the m_is

can be coded in binary. Then, in a certificate of polynomial size, we cannot enumerate all the n squares to give their coordinates. There might exist a compact analytical formula that would characterize solutions (say, the j-th square of size a_i is placed at coordinates $f(i, j)$), but this is far from being obvious. We do not know whether this latter variant is in NP or not.

It is much harder to identify a problem that is known not to be in NP, at least without making any assumption like P $\neq$ NP. A (complicated) example is the problem of deciding whether two regular expressions represent different languages, where the expressions are limited to four operators: union, concatenation, the Kleene star (zero or more copies of an expression), and squaring (two copies of an expression). Any algorithm for this problem requires exponential space, hence, exponential verification time [77].

Another problem that is not in NP is the program termination problem, or halting problem (decide whether a program will terminate on a given input). However, this example is a little excessive because no algorithm can exist to solve it, regardless of its complexity [82].

6.3 NP-complete problems and reduction theory

As explained in the previous section, we do not know whether the inclusion P $\subseteq$ NP is strict or not. However, we are able to compare the complexity of problems in NP; Cook's idea was to prove that some problems of the NP class are at least as difficult as all other problems of the same class. These problems are called NP-complete and form the subclass NPC of the class NP. They are *the most difficult problems* of NP. If we are able to solve one NP-complete problem in polynomial time, then we will be able to solve all problems of NP in polynomial time, and we will have P = NP. The main objective of this section is to explain this line of reasoning in full detail and to explore some consequences.

We detail the theory of reduction, which aims at proving that a problem is *more difficult* than another one. However, if we want to prove that a problem is more difficult than any other one, we need to identify the first NP-complete problem, as explained in Section 6.3.2. Note that a set of NP-complete problems with the corresponding reductions is presented in Section 6.4.

6.3.1 Polynomial reduction

We start by explaining the mechanism of polynomial reduction, i.e., how to prove that a problem is more difficult than another. Consider two decision problems P_1 and P_2. How can we prove that P_1 is more difficult than P_2? We say that P_2 is polynomially reducible to P_1 and write $P_2 \xrightarrow{pr} P_1$ if, whenever

we are given an instance $\mathcal{I}_2$ of problem P_2, we can convert it, with only a polynomial-time algorithm, into an instance $\mathcal{I}_1$ of P_1, in such a way that $\mathcal{I}_2$ has the answer "Yes" if and only if $\mathcal{I}_1$ has the answer "Yes."

Now, if P_2 is polynomially reducible to P_1, then P_1 must be more difficult than P_2 (or more precisely, at least as difficult as P_2). Indeed, if there exists a polynomial algorithm to solve P_1, then by applying the polynomial reduction, and because the composition of two polynomials is a polynomial, there exists a polynomial algorithm to solve P_2. Given an instance $\mathcal{I}_2$ of P_2, we can indeed convert it into instance $\mathcal{I}_1$ of P_1, and since there is an equivalence between solutions of $\mathcal{I}_1$ and $\mathcal{I}_2$, the polynomial algorithm for P_1 executed on instance $\mathcal{I}_1$ returns the solution for instance $\mathcal{I}_2$. Take the contrapositive of this statement. If there is no polynomial algorithm to solve P_2, then there is none to solve P_1 either, so P_1 is more difficult.

We point out that polynomial reduction is a transitive operation: If $P_3 \xrightarrow{pr} P_2$ and $P_2 \xrightarrow{pr} P_1$, then $P_3 \xrightarrow{pr} P_1$. Again, this is because the composition of two polynomials is a polynomial, nothing more.

Note also that it is mandatory to have the equivalence of solutions, i.e., if $\mathcal{I}_1$ has a solution then $\mathcal{I}_2$ has one, and if $\mathcal{I}_2$ has a solution then $\mathcal{I}_1$ has one. Otherwise, the polynomial reduction $P_2 \xrightarrow{pr} P_1$ would not imply that P_1 is more difficult than P_2.

6.3.2 Cook's theorem

The fundamental result of the P versus NP theory is Cook's theorem [25], which shows that the satisfiability problem SAT is the most difficult problem in NP. This means that all other problems in NP are polynomially reducible to SAT. We introduce SAT and give a brief intuitive sketch of Cook's proof.

DEFINITION 6.6 (SAT)**.** Let F be a Boolean formula with n variables $x_1, \ldots, x_n$ and p clauses $C_1, \ldots, C_p$: $F = C_1 \wedge C_2 \wedge \cdots \wedge C_p$, with, for $1 \leqslant i \leqslant p$, $C_i = x^*_{i_1} \vee x^*_{i_2} \vee \cdots \vee x^*_{i_{f(i)}}$, $1 \leqslant i_k \leqslant n$ for $1 \leqslant k \leqslant f(i)$, and $x^* = x$ or $\overline{x}$. Does there exist an instantiation of the n variables such that F is true (i.e., C_i is true for $1 \leqslant i \leqslant p$)?

Clearly, SAT is in NPC, and a certificate can simply be the list of the instantiation of each variable (whether a given x_i is instantiated to true or false). However, it seems difficult to solve SAT without a certificate; because some clauses have x_i and other $\overline{x_i}$, we may have to try all 2^n possible instantiations to find one that satisfies the formula. In other words, SAT seems to be a hard problem indeed.

Cook's theorem states that all problems in NP are polynomially reducible to SAT. The main idea of the proof is the following: Consider any problem P in NP, and take an arbitrary instance I, together with its certificate C. The proof goes by simulating the execution of the Turing machine that accepts the couple (I, C) as input and outputs "Yes" after a polynomial number of steps.

Because Turing machines are simple, their behavior can be characterized by clauses linking a set of variables. We can define $x_{t,j,s}$ as a variable that is true if after t steps of computation, symbol s is in position j of the ribbon, and we can simulate the operation of the machine using these variables. There are many such variables, but only a polynomial number in $|I|$, and a polynomial number of clauses as well. A detailed, but easy-to-follow, proof is given by Wilf [108].

6.3.3 Growing the class NPC of NP-complete problems

Now that we have the first NP-complete problem handy, how can we find more? To prove that a problem, P_1, is in NPC, we merely have to prove that SAT is polynomially reducible to this problem. Indeed, by composition, all problems in NP are reducible to SAT, hence, to P_1. The reduction takes several steps:

1. Prove that $P_1 \in NP$: We must be able to build a certificate of polynomial size, and then, for any instance $\mathcal{I}_1$ of problem P_1, we must be able to check in polynomial time whether the certificate is a solution. Usually, this first step is easy, but it should not be forgotten.

2. Prove the completeness of P_1: We transform an arbitrary instance $\mathcal{I}$ of SAT into an instance $\mathcal{I}_1$ of P_1 in polynomial time, and such that:

   (a) the size of $\mathcal{I}_1$ is polynomial in the size of $\mathcal{I}$;

   (b) $\mathcal{I}_1$ has a solution $\Leftrightarrow$ $\mathcal{I}$ has a solution.

Let us come back to the construction of instance $\mathcal{I}_1$. The construction should be done in polynomial time, but this is usually implicit because the size of $\mathcal{I}_1$ should be polynomial in the size of $\mathcal{I}$, and because we perform only "reasonable" operations.

Assume that we have polynomially reduced SAT to P_1. We now have two problems in NPC, namely, P_1 and SAT. If we want to extend the class to a third problem, $P_2 \in NP$, should we reduce SAT or P_1 to P_2? Of course, the answer is that either reduction works. Indeed, we have so far:

- $P_1 \xrightarrow{pr} SAT$ and $P_2 \xrightarrow{pr} SAT$ (both by Cook's theorem).

- $SAT \xrightarrow{pr} P_1$ (our previous reduction).

We can prove that $SAT \xrightarrow{pr} P_2$ either directly or via the reduction $P_1 \xrightarrow{pr} P_2$ because $SAT \xrightarrow{pr} P_1$ and because, as we have already stated, polynomial reduction is a transitive operation.

In other words, to show that some problem P_2 in NP is in NPC, we can pick any NP-complete problem P_1 in NPC and show that $P_1 \xrightarrow{pr} P_2$. This will show that P_2 is in NPC, and P_2 will itself become a candidate NP-complete problem to pick up for later reductions.

A decision problem is said to be NP-hard when it can be polynomially reduced from an NP-complete problem, but it is not known whether it belongs to NP.

6.3.4 Optimization problems versus decision problems

We have been focusing so far on decision problems, but, in many practical situations, we have to solve an optimization problem in which we want to maximize or minimize a given criterion. Optimization problems (also called search problems) are more complex than decision problems, but one can always restrict an optimization problem so that it becomes a decision problem.

For instance, the graph coloring problem is usually an optimization problem: What is the minimum number of colors required to color the graph? The restriction to the decision problem is the COLOR problem of Definition 6.3: Can we color the graph with at most k colors? If we can solve the optimization problem, we have immediately the solution to the decision problem, for any value of k. In this particular case, we also can go the other way round. If we are able to solve the decision problem, then we can find the answer to the optimization problem by performing a binary search on k ($1 \leqslant k \leqslant |V|$) and computing the answer of the decision problem for each value of k. The binary search adds a factor $\log(|V|)$ to the algorithm complexity, so that if we had a polynomial algorithm, it remains polynomial. The two problems (optimization and decision) have the same complexity. In most cases, the optimization problem can be solved using a binary search as described above. However, this result is not always true; it can be difficult to find the answer to the optimization problem, even though we can solve the decision problem. In some extreme situations, there may be no solution to the optimization problem. For instance, there is no solution to the problem "Find the smallest rational number x such that $x^2 \geqslant 2$" because $\sqrt{2}$ is irrational, while it is easy to solve the decision problem in polynomial time: "Given a rational number x, do we have $x^2 \geqslant 2$?" (simply compute a square and compare it to 2).

Transforming a decision problem into an optimization problem may not be natural or even possible. However, we can always define the *associated* decision problem of an optimization problem: If the optimization problem aims at minimizing a value x with some constraints, the decision problem adds a value x_0 as an input to the problem, and the question is whether there is a solution achieving a value $x \leqslant x_0$.

A typical example is based on 2-PARTITION. Consider the scheduling problem with two processors, where we want to schedule n tasks of length a_i. Ideally, we want to 2-partition the tasks so that the execution finishes as soon as possible, but if this is not possible, we minimize the difference of finish times, which amounts to minimizing the global finish time. Formally, the objective is to find a subset I that minimizes $x = |\sum_{i \in I} a_i - \sum_{i \notin I} a_i|$. The associated decision problem with target value $x_0 = 0$ is exactly 2-PARTITION, that will be shown to be NP-complete (Exercise 7.20, p. 155). By misuse of language,

we say that an optimization problem is NP-complete if the associated decision problem for some well-chosen target value is NP-complete. Hence, the scheduling problem with two processors as defined above is NP-complete.

6.4 Examples of NP-complete problems and reductions

At this point, we know that SAT is NP-complete. As already discussed, we proceed by reduction to increase the list of NP-complete problems. In this section, we show that 3-SAT, CLIQUE, and VERTEX-COVER are in NPC. We also give references for the NP-completeness of 2-PARTITION, HC (Hamiltonian Cycle), and then show that TSP (Traveling Salesman Problem) is NP-complete.

6.4.1 3-SAT

DEFINITION 6.7 (3-SAT)**.** Let F be a Boolean formula with n variables $x_1, \ldots, x_n$ and p clauses $C_1, \ldots, C_p$: $F = C_1 \wedge C_2 \wedge \cdots \wedge C_p$, with, for $1 \leqslant i \leqslant p$, $C_i = x_{i_1}^* \vee x_{i_2}^* \vee x_{i_3}^*$, $1 \leqslant i_k \leqslant n$ for $1 \leqslant k \leqslant 3$, and $x^* = x$ or $\overline{x}$. Does there exist an instantiation of the variables such that F is true (i.e., C_i is true for $1 \leqslant i \leqslant p$)?

This problem is the restriction of SAT to the case where each clause consists of three variables, i.e., following the notations of Section 6.3.2, $f(i) = 3$ for $1 \leqslant i \leqslant p$. In fact, 3-SAT is so close to SAT that one might wonder why consider 3-SAT in addition to, or replacement of, SAT. The reason is that it is much easier to manipulate clauses with exactly three variables. Furthermore, proving the NP-completeness of 3-SAT is also a good exercise for our first reduction.

THEOREM 6.1. *3-SAT is NP-complete.*

Proof. This proof, as well as the next ones, follows the reduction method to prove that a problem is NP-complete.

First, we prove that 3-SAT is in NP. We can simply claim that it is in NP because it is a restriction of SAT, which itself is in NP. It also is easy to prove it directly. We consider an instance I of 3-SAT, which is of size $O(n + p)$. A certificate is a set of truth values, one for each variable. Therefore, it is of size $O(n)$, which is polynomial in the size of the instance. It is easy to check whether the certificate is a solution, and this takes a time $O(n + p)$. Altogether, 3-SAT is in NP.

To prove the completeness, we reduce an instance of SAT. So far, it is the only problem that we know to be NP-complete, thanks to Cook's theorem, so we have no choice.

Let $\mathcal{I}_1$ be an instance of SAT. First, we need to build an instance $\mathcal{I}_2$ of 3-SAT that will have a solution if and only if $\mathcal{I}_1$ has one. $\mathcal{I}_1$ consists of p clauses $C_1, \ldots, C_p$, of lengths $f(1), \ldots, f(p)$, and each clause is made of some of the n variables $x_1, \ldots, x_n$.

Instance $\mathcal{I}_2$ initially consists of the n variables $x_1, \ldots, x_n$. Then, we add to $\mathcal{I}_2$ variables and clauses corresponding to each clause C_i of $\mathcal{I}_1$. We build a set of clauses made of exactly three variables, and the goal is to have the equivalence between C_i and the constructed clauses. We consider various cases:

- If C_i has a single variable x, we add to instance $\mathcal{I}_2$ two new variables a_i and b_i and four clauses: $x \vee a_i \vee b_i$, $x \vee \overline{a_i} \vee b_i$, $x \vee a_i \vee \overline{b_i}$, and $x \vee \overline{a_i} \vee \overline{b_i}$.
- If C_i has two variables $x_1 \vee x_2$, we add to instance $\mathcal{I}_2$ one new variable c_i and two clauses: $x_1 \vee x_2 \vee c_i$ and $x_1 \vee x_2 \vee \overline{c_i}$.
- If C_i has three variables, we add it to $\mathcal{I}_2$.
- If C_i has k variables, with $k > 3$, $C_i = x_1 \vee x_2 \vee \cdots \vee x_k$, then we add $k-3$ new variables $z_1^i, z_2^i, \ldots, z_{k-3}^i$ and $k-2$ clauses: $x_1 \vee x_2 \vee z_1^i$, $x_3 \vee \overline{z_1^i} \vee z_2^i$, ..., $x_{k-2} \vee \overline{z_{k-4}^i} \vee z_{k-3}^i$, and $x_{k-1} \vee x_k \vee \overline{z_{k-3}^i}$.

Note that all clauses that are added to $\mathcal{I}_2$ are exactly made of three variables, and that the construction is done in polynomial time. Then, we must check the different points of the reduction.

First, note that $size(\mathcal{I}_2)$ is polynomial in $size(\mathcal{I}_1)$ (and even linear); indeed, $size(\mathcal{I}_2) = O(n + \sum_{i=1}^{p} f(i))$.

Then, we start with the easy side, which consists of proving that if $\mathcal{I}_1$ has a solution, then $\mathcal{I}_2$ has a solution. Let us assume that $\mathcal{I}_1$ has a solution. We have an instantiation of variables $x_1, \ldots, x_n$ such that C_i is true for $1 \leqslant i \leqslant p$. Then, a solution for $\mathcal{I}_2$ keeps the same values for the x_is, and set all a_j, b_j and c_j values to true. Therefore, if a clause with at most three variables is true in $\mathcal{I}_1$, all corresponding clauses in $\mathcal{I}_2$ are true. Consider now a clause C_i in $\mathcal{I}_1$ with $k > 3$ variables: $C_i = x_1 \vee x_2 \vee \cdots \vee x_k$. Let x_j be the first variable of the clause that is true. Then, for the solution of $\mathcal{I}_2$, we instantiate $z_1^i, \ldots, z_{j-2}^i$ to true and $z_{j-1}^i, \ldots, z_{k-3}^i$ to false. With this instantiation, all clauses of $\mathcal{I}_2$ are true, and thus $\mathcal{I}_1 \Rightarrow \mathcal{I}_2$.

For the other side, let us assume that $\mathcal{I}_2$ has a solution. We have an instantiation of all variables x_i, a_i, b_i, c_i, and z_j^i that is a solution of $\mathcal{I}_2$. Then, we prove that the same instantiation of $x_1, \ldots, x_n$ is a solution of the initial instance $\mathcal{I}_1$. First, for a clause with one or two variables, whatever the values of a_i, b_i, and c_i, we necessarily have x or $x_1 \vee x_2$ equal to true because we have added clauses constraining the extra variables. The clauses with three variables remain true since we have not modified them. Finally, let C_i be a clause of $\mathcal{I}_1$ with $k > 3$ variables, $C_i = x_1 \vee x_2 \vee \cdots \vee x_k$. We reason by contradiction. If this clause is false, then, necessarily, because of the first clause added to $\mathcal{I}_2$ when processing clause C_i, z_1^i must be true, and similarly we can prove that all z_j^i variables must be true. The contradiction arises for the last clause because it imposes that $\overline{z_{k-3}^i}$ should be true if x_{k-1} and x_k are

both false. Therefore, by contradiction, at least one of the x_js must be true and the clause of $\mathcal{I}_1$ is true. We finally have $\mathcal{I}_2 \Rightarrow \mathcal{I}_1$, which concludes the proof. □

As a final remark, we point out that not all restrictions of a given NP-complete problem remain NP-complete. For instance, 2-SAT, the SAT problem where each clause contains exactly two variables, belongs to P. Several variants of 3-SAT are shown NP-complete in the exercises.

6.4.2 CLIQUE

We now consider a problem that is very different from SAT.

DEFINITION 6.8 (CLIQUE)**.** Let $G = (V, E)$ be a graph and k be an integer such that $1 \leqslant k \leqslant |V|$. Does there exist a clique of size k (i.e., a complete subgraph of G with k vertices)?

This is a graph problem, and the size of the instance is polynomial in $|V|$ (recall that $|E| \leqslant |V|^2$, so we do not need to consider $|E|$ in the instance size).

THEOREM 6.2. *CLIQUE is NP-complete.*

Proof. First we prove that CLIQUE is in NP. The certificate is the list of vertices of a clique, and we can check in polynomial time (even quadratic time) whether it is a clique or not. For each vertex pair of the certificate, the edge between these vertices must be in E.

The completeness is obtained with a reduction from 3-SAT. We could do a reduction from SAT, but 3-SAT is more regular, so we give it preference for the reduction. Let $\mathcal{I}_1$ be an instance of 3-SAT with n variables and p clauses. Then we build an instance $\mathcal{I}_2$ of CLIQUE. We add three vertices to the graph for each clause (each vertex corresponds to one of the literals of the clause) and then we add an edge between two vertices if and only if (i) they are not part of the same clause and (ii) they are not antagonist (i.e., one corresponding to a variable x_i and the other to its negation $\overline{x_i}$). An example is shown in Figure 6.1, with the graph obtained for a formula with three clauses $C_1 \wedge C_2 \wedge C_3$, with $C_1 = x_1 \vee \overline{x_2} \vee \overline{x_3}$, $C_2 = \overline{x_1} \vee x_2 \vee x_3$, and $C_3 = x_1 \vee x_2 \vee x_3$.

Note that $\mathcal{I}_2$ is a graph with $3p$ vertices; the size of this instance, therefore, is polynomial in the size of $\mathcal{I}_1$. Moreover, we fix in instance $\mathcal{I}_2$ the integer k of the CLIQUE definition such that $k = p$. We are now ready to check the equivalence of the solutions.

Assume first that the instance $\mathcal{I}_1$ of 3-SAT has a solution. Then, we pick a vertex corresponding to a variable that is true in each clause, and it is easy to check that the subgraph made of these p vertices is a clique. Indeed, two of such vertices are not in the same clause, and they are not antagonistic; therefore, there is an edge between them.

On the other side, if there is a clique of size k in instance $\mathcal{I}_2$, then necessarily there is one vertex of the clique in each clause (otherwise, the two vertices

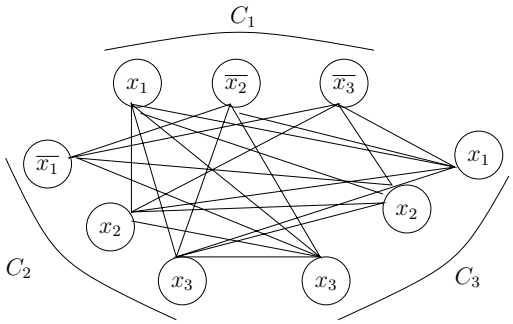


FIGURE 6.1: Example: Reduction of an instance of 3-SAT to an instance of CLIQUE.

within the same clause would not be connected). We choose these vertices to instantiate the variables, and we obtain a solution because we never make contradictory choices (because two antagonistic vertices cannot be part of the clique, there is no edge between them). This concludes the proof. □

We discuss variants of the CLIQUE problem in Section 6.5.

6.4.3 VERTEX-COVER

We continue to enrich the class NPC with another graph problem. We say that an edge $e = (u, v)$ is *covered* by its endpoints u and v.

DEFINITION 6.9 (VERTEX-COVER)**.** Let $G = (V, E)$ be a graph and k be an integer such that $1 \leqslant k \leqslant |V|$. Do there exist k vertices $v_{i_1}, \ldots, v_{i_k}$ such that each edge $e \in E$ is covered by (at least) one of the v_{i_j}, for $1 \leqslant j \leqslant k$?

THEOREM 6.3. *VERTEX-COVER is NP-complete.*

Proof. It is easy to check that VERTEX-COVER is in NP. The certificate is a set of k vertices, $V_c \subseteq V$, and for each edge $(v_1, v_2) \in E$, we check whether $v_1 \in V_c$ or $v_2 \in V_c$. The verification is done in time $|E| \times k$, and, therefore, it is polynomial in the problem size.

This problem is once again a graph problem, so we choose to use a reduction from CLIQUE, which turns out to be straightforward. Let $\mathcal{I}_1$ be an instance of CLIQUE: It consists of a graph $G = (V, E)$ and an integer k. We consider the following instance $\mathcal{I}_2$ of VERTEX-COVER. The graph is $\overline{G} = (V, \overline{E})$, which is the complementary graph of G, i.e., an edge is in $\overline{G}$ if and only if it is not in G (see the example in Figure 6.2). Moreover, we set the size of the covering set to $|V| - k$.

If instance $\mathcal{I}_1$ has a solution, G has a clique of size k, and, therefore, the $|V| - k$ vertices that are not part of the clique form a covering set of $\overline{G}$. Reciprocally,

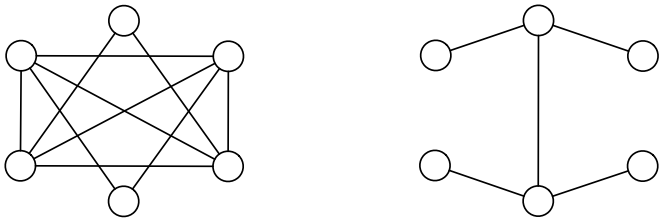


FIGURE 6.2: Example: Reduction of an instance of CLIQUE (on the left, graph G, $k = 4$) to an instance of VERTEX-COVER (on the right, graph $\overline{G}$, size of the cover $|V| - k = 2$).

if $\mathcal{I}_2$ has a solution, then the vertices that are not part of the covering set form a clique in the original graph G. This concludes the proof. □

6.4.4 Scheduling problems

Scheduling is the activity that consists of mapping an application onto a target platform and of assigning execution times to its constitutive parts. The application can often be represented as a task graph, where nodes denote computational tasks and edges model precedence constraints between tasks. For each task, an assignment (choose the processor that will execute the task) and a schedule (decide when to start the execution) are determined. The goal is to obtain an efficient execution of the application, which translates into optimizing some objective function. The traditional objective function in the scheduling literature is the minimization of the total execution time, or *makespan*; however, we will see examples with other objectives, such as those of the case study devoted to online scheduling (Chapter 15).

Traditional scheduling assumes that the target platform is a set of p identical processors, and that no communication cost is paid. In that context, a task graph is a directed acyclic vertex-weighted graph $G = (V, E, w)$, where the set V of vertices represents the tasks, the set E of edges represents precedence constraints between tasks ($e = (u, v) \in E$ if and only if $u \prec v$, where $\prec$ is the precedence relation), and the weight function $w : V \longrightarrow \mathbb{N}^*$ gives the weight (or duration) of each task. Task weights are assumed to be positive integers. A schedule σ of a task graph is a function that assigns a start time to each task: $\sigma : V \longrightarrow \mathbb{N}^*$ such that $\sigma(u) + w(u) \leqslant \sigma(v)$ whenever $e = (u, v) \in E$. In other words, a schedule preserves the *precedence constraints* induced by the precedence relation $\prec$ and embodied by the edges of the precedence graph. If $u \prec v$, then the execution of u begins at time $\sigma(u)$ and requires $w(u)$ units of time, and the execution of v at time $\sigma(v)$ must start after the end of the execution of u. Obviously, if there were a cycle in the task graph, no schedule could exist, hence, the restriction to acyclic graphs and, thus, the focus on Directed Acyclic Graphs (DAGs).

There are other constraints that must be met by schedules, namely, *resource constraints*. When there is an infinite number of processors (in fact, when there are as many processors as tasks), the problem is *with unlimited processors*, and denoted $P\infty|prec|Cmax$ in the literature [44]. We use the shorter notation SCHED(∞) in this book; each task can be assigned to its own processor. When there is a fixed number $p < n$ of available processors, the problem is *with limited processors*, and the general problem is denoted SCHED(p). SCHED(2) represents the scheduling problem with only two processors. Note that SCHED(∞) is equivalent to SCHED(q) for any value $q \geqslant n$, where n is the number of tasks. In the case with limited processors, a problem is defined by the task graph and the number of processors p. An allocation function $\mathsf{alloc} : V \longrightarrow \mathcal{P}$ is then required, where $\mathcal{P} = \{1, \ldots, p\}$ denotes the set of available processors. This function assigns a target processor to each task. The resource constraints simply specify that no processor can be allocated more than one task at the same time:

$$\mathsf{alloc}(T) = \mathsf{alloc}(T') \Rightarrow \begin{cases} \sigma(T) + w(T) \leqslant \sigma(T') \\ \text{or } \sigma(T') + w(T') \leqslant \sigma(T). \end{cases}$$

This condition expresses the fact that if two tasks T and T' are allocated to the same processor, then their executions cannot overlap in time.

The *makespan* $\mathrm{MS}(\sigma, p)$ of a schedule σ that uses p processors is its total execution time: $\mathrm{MS}(\sigma, p) = \max_{v \in V}\{\sigma(v) + w(v)\}$ (assuming that the first task(s) is (are) scheduled at time 0). The makespan is the total execution time, or finish time, of the schedule. Let $\mathrm{MS}_{opt}(p)$ be the value of the makespan of an optimal schedule with p processors: $\mathrm{MS}_{opt}(p) = \min_\sigma \mathrm{MS}(\sigma, p)$. Because schedules respect precedence constraints, we have $\mathrm{MS}_{opt}(p) \geqslant w(\Phi)$ for all paths Φ in G (weights extend to paths in G as usual). We also have $\mathrm{Seq} \leqslant p \times \mathrm{MS}_{opt}(p)$, where $\mathrm{Seq} = \sum_{v \in V} w(v) = \mathrm{MS}_{opt}(1)$ is the sum of all task weights.

While SCHED(∞) has polynomial complexity (simply traverse the graph and start each task as soon as possible using a fresh processor), problems with a fixed amount of resources are known to be difficult. Letting DEC be the decision problem associated with SCHED, and INDEP the restriction of DEC to independent tasks (no precedence constraints), i.e., $E = \emptyset$, well-known complexity results are summarized below:

- INDEP(2) is NP-complete but can be solved by a pseudopolynomial algorithm. Moreover, $\forall\, \varepsilon > 0$, INDEP(2) admits a $(1+\varepsilon)$-approximation whose complexity is polynomial in $\frac{1}{\varepsilon}$ (see Section 8.1.5, p. 187).
- INDEP is NP-complete in the strong sense (see Exercise 7.10, p. 152) but can be approximated up to some constant factor (see Exercise 9.5, p. 215). Moreover, $\forall \varepsilon > 0$, there is a $(1 + \varepsilon)$-approximation algorithm for this problem [50].
- DEC(2) (and hence DEC) is NP-complete in the strong sense (see Exercise 7.11, p. 152).

All these results are gathered here for the sake of comprehensiveness. The impatient reader who wonders what is the meaning of NP-complete *in the strong sense* may refer to Section 6.6, p. 145, and to understand what is *an approximation algorithm*, she/he may have a quick look at Section 8.1, p. 179 right now.

Scheduling and certificates

Scheduling problems provide a nice illustration of the attention that must be paid to certificates. Consider the DEC decision problem, namely, scheduling a task graph with p processors and a given deadline D. For the schedule to be valid, both precedence and resource constraints must be enforced. The question is to decide whether there exists a schedule whose makespan does not exceed the deadline.

A naive verification of the schedule is to describe which tasks are executed onto which processors at each time step. Unfortunately, this description may lead to a certificate of exponential size; the time steps range from 1 to D, and the size of the scheduling problem is $O(n + p + \log W + \log D)$, where $W = \sum_{v \in V} w(v)$.

A polynomial size verification of the schedule can be easily obtained using *events*, which are time steps where a new task begins or ends. There is a polynomial number of such events ($2n$), and for each of them we perform a polynomial number of checks. From the definition of the schedule, we first construct the ordered list of events in polynomial time. The basic idea is to maintain the set of tasks that have been completed and the set of processors that are currently idle. If the event corresponds to starting a new task, we check that all its predecessors have been completed, and that the target processor belongs to the set of idle processors (and then we remove it from this set). If the event corresponds to completing a task, we mark the task accordingly, and we re-insert the target processor into the set of idle processors. Note that if several events take place at the same time step, we should start with those that correspond to task completions. We perform these checks one event after the other until we reach the last one, which corresponds to the completion of the last task, and which much take place not later than D.

In summary, we see that the weights of the tasks (given by the function w) prevent us from using a naive verification of the validity of a schedule at each step of its execution. This is because the makespan is not polynomial in the problem size.

6.4.5 Other famous NP-complete problems

We have initiated discussions with the 2-PARTITION problem (Definition 6.1) that is one of the most widely used problems to perform reductions, since it turns out to be NP-complete while being quite simple in its formulation. The NP-completeness of 2-PARTITION will be shown in Exercise 7.20, p. 155, but

from now on, we assume that this problem is indeed NP-complete.

The COLOR problem (see Definition 6.3) given in Section 6.2.2 is also NP-complete and the proof is the purpose of Exercise 7.7, p. 151. Other problems will discuss variants of this graph coloring problem.

Another useful problem is HC (Hamiltonian Cycle, see Definition 6.4). We have already shown that HC is in NP (see Section 6.2.2). For the completeness, we refer the interested reader to involved reduction in [27]. There is a nice reduction from 3-SAT in the first edition of the book, and the current edition performs a reduction from VERTEX-COVER.

Starting from HC, it is easy to prove that TSP (see Definition 6.5) also is NP-complete. It is clear that TSP is in NP; a certificate is an ordered list of vertices. The reduction comes from HC. Let $\mathcal{I}_1$ be an instance of HC: This is a graph $G = (V, E)$. We build the following instance $\mathcal{I}_2$ of TSP. The graph $G' = (V, E')$ has the same set of vertices as G, but it is a complete graph. We set $k = 0$, i.e., we want to find a cycle of weight 0. Finally, for $e \in E'$ we define the cost function w such that $w(e) = 0$ if $e \in E$, and $w(e) = 1$ otherwise. This reduction is obviously of polynomial time, and the equivalence of solutions is straightforward. Note that this last NP-completeness result comes from the fact that TSP is a weighted version of HC.

For a reference list of problems known to be NP-complete, we refer the reader to the book by Garey and Johnson [38].

6.5 Importance of problem definition

In this section, we point out subtleties in problem definitions. A parameter can be either fixed for the problem or part of the problem instance. Consider the problem CLIQUE introduced in Section 6.4.2. Given a graph $G = (V, E)$, we introduce the notion of β-clique of size k, where β is a rational such that $0 < \beta \leqslant 1$ [83]; a β-clique is a subgraph of G of size k (k vertices), with edge density at least β. The edge density is the ratio of the number of edges in the subgraph over the number of edges in a clique of size k, i.e., $\binom{k}{2}$. We can now define a variant of the CLIQUE problem:

DEFINITION 6.10 (BCLIQUE)**.** Let $G = (V, E)$ be a graph, β be a rational number such that $0 < \beta \leqslant 1$, and k be an integer such that $1 \leqslant k \leqslant |V|$. Does there exist a β-clique of size k in G?

In the BCLIQUE problem, β is part of the instance. Therefore, we can do a trivial reduction from CLIQUE, letting $\beta = 1$, to prove that it is NP-complete. However, we may define the problem in a different way, where β is given. For a constant β such that $0 < \beta \leqslant 1$, we define:

DEFINITION 6.11 (BCLIQUE(β))**.** Let $G = (V, E)$ be a graph and k be an integer such that $1 \leqslant k \leqslant |V|$. Does there exist a β-clique of size k in G?

We have CLIQUE = BCLIQUE(1). However, the NP-completeness of CLIQUE does not imply the NP-completeness of BCLIQUE(β) for any value of β. We prove this NP-completeness for any fixed value $0 < \beta < 1$ in the following theorem:

THEOREM 6.4. *BCLIQUE(β) is NP-complete for any rational number $\beta = \frac{p}{q}$, where p and q are positive integer constants and $p < q$.*

Proof. It is clear that BCLIQUE(β) is in NP, and the reduction comes logically from the classical CLIQUE problem. The idea is to construct an auxiliary graph $G' = (V', E')$ and to prove that G has a clique of size k if and only if $G \cup G'$ has a β-clique of size $|V'| + k$.

We build the set of vertices V' of size $|V'| = 4(|V|^2 + k^2)q - k$, containing vertices v'_1 to $v'_{|V'|}$. For $1 \leqslant i \leqslant |V'|$ and $j \in [i+1, i+|V|] \mod |V'|$, we add an edge between v_i and v_j. Therefore, each node has $2|V|$ edges, and we have added a total of $|V||V'|$ edges. Next, we add random edges in order to have a total of $K = \frac{p}{q}\binom{|V'|+k}{2} - \binom{k}{2}$ edges between the $|V'|$ vertices. Because $|V'|+k$ is a multiple of $2q$, K is an integer. Moreover, $|V'|$ is large enough so that we can prove that $\binom{|V'|}{2} \geqslant K \geqslant |V||V'|$ (see [83]), i.e., there were initially fewer than K edges, and we can have a total of K edges without exceeding the maximum number of edges in $|V'|$.

There remains to prove that G has a clique of size k if and only if $G \cup G'$ has a $\frac{p}{q}$-clique of size $|V'| + k$. Suppose first that there is a clique C of size k in G. We consider the subgraph Q of $G \cup G'$ containing vertices $C \cup V'$. We have $|Q| = |V'| + k$ and the number of edges is $K + \binom{k}{2} = \frac{p}{q}\binom{|V'|+k}{2}$; therefore, Q is a $\frac{p}{q}$-clique by definition.

Suppose now that there is a $\frac{p}{q}$-clique Q of size $|V'| + k$ in $G \cup G'$. We first construct a $\frac{p}{q}$-clique Q' such that $|Q'| = |Q|$ and $V' \subset Q'$. Since $|Q| > |V'|$, $|V' \setminus Q| \leqslant |V|$, and each vertex in $V' \setminus Q$ cannot be connected to more than $|V| - 1$ vertices of $V' \setminus Q$. Moreover, each vertex of $V' \setminus Q$ is of degree at least $2|V|$ and, therefore, it is connected to at least $|V|+1$ vertices of Q, while vertices of $Q \cap V$ are connected to at most $|V|-1$ vertices (all of them from V). Therefore, we can replace $|V' \setminus Q|$ vertices of $Q \cap V$ with the remaining vertices of V', with no reduction in the edge density. We obtain a $\frac{p}{q}$-clique Q' such that $|Q'| = |Q|$ and $V' \subset Q'$. Then, $|Q' \cap V| = k$. To see that $Q' \cap V$ is a clique of size k in V, consider the density of G'; it is K by construction. If $Q' \cap V$ does not contribute $\binom{k}{2}$ edges, then Q' cannot have density $\frac{p}{q}$. Therefore, $Q' \cap V$ is a clique of size k, hence concluding the proof. □

In scheduling problems (see Section 6.4.4, p. 140), the same distinction is often implicitly made, whether the number of processors p is part of the problem instance or not. For instance, if all tasks are unit-weighted, DEC is

NP-complete (with p in the problem instance), while DEC(2) can be solved in polynomial time and DEC(3) is an open problem [38].

6.6 Strong NP-completeness

The last technical discussion of this chapter is related to *weak* and *strong* NP-completeness. This refinement of the NPC problem class applies to problems involving numbers, such as 2-PARTITION, but also TSP, because of edge weights.

Consider a decision problem P, and let I be an instance of this problem. We have already discussed how to compute $size(I)$, the size of the instance, encoded in binary. We now define $max(I)$, which is the *maximum* size of the instance, typically corresponding to the problem instance with integers coded in unary. To give an example, consider an instance I of 2-PARTITION with n integers $a_1, \ldots, a_n$. As already discussed, we can have $size(I) = n + \sum_{1 \leqslant i \leqslant n} \log(a_i)$, or any similar (polynomially related) expression. Now we can have $max(I) = n + \sum_{1 \leqslant i \leqslant n} a_i$, or $max(I) = n + \max_{1 \leqslant i \leqslant n} a_i$, or any similar (polynomially related) expression.

Then, given a polynomial p, we define P_p, the problem P restricted to p, as the problem restricted to instances such that $max(I)$ is smaller than $p(size(I))$, i.e., the size of the instance coded in unary is bounded applying p to the binary size of the instance. A problem P (in NP) is NP-complete *in the strong sense* if and only if there exists a polynomial p such that P_p remains NP-complete. Otherwise, if the problem restricted to p can be solved in polynomial time, the problem is NP-complete *in the weak sense*; intuitively, in this case, the problem is difficult only if we do not bound the size of the input in the problem instance.

Note that for a graph problem such as the bipartite graph problem, there are no numbers, so $max(I) = size(I)$ and the problem is NP-complete in the strong sense. For problems with numbers (including weighted graph problems), one must be more careful. Coming back to 2-PARTITION, we have seen in Section 6.2.1 that it can be solved by a dynamic-programming algorithm running in time $O(n \sum_{i=1}^{n} a_i)$, or equivalently in time $O(max(I))$. Therefore, any instance I of 2-PARTITION can be solved in time polynomial in $max(I)$, which is the definition of a pseudopolynomial problem. And 2-PARTITION is not NP-complete in the strong sense (one says it is NP-complete in the weak sense).

To conclude this section, we introduce a problem with numbers that is NP-complete in the strong sense: 3-PARTITION. The name of this problem is misleading because this problem is different from partitioning n integers into three sets of same size.

DEFINITION 6.12 (3-PARTITION)**.** Given an integer B, and $3n$ integers $a_1, \ldots, a_{3n}$, can we partition the $3n$ integers into n triplets, each of sum B? We can assume that $\sum_{i=1}^{3n} a_i = nB$ (otherwise, there is no solution), and that $B/4 < a_i < B/2$ (so that one needs exactly three elements to obtain a sum B).

Contrary to 2-PARTITION, 3-PARTITION is NP-complete in the strong sense [38].

6.7 Why does it matter?

We conclude this chapter with a discussion on polynomial problems. Why focus on polynomial problems? If the size of the data is in n, from a practical perspective it is much better to have an algorithm in $(1.0001)^n$, which is exponential, than a polynomial-time algorithm in n^{1000}. In such a case, the polynomial-time algorithm is still slower than the exponential one for $n = 10^9$, and, therefore, the exponential algorithm is faster in any practical situation. However, n^{1000} is not practical either. In general, polynomial algorithms have a small degree, typically not exceeding 4 and almost always smaller than 10.

Polynomial-time algorithms are likely to be efficient algorithms, so when confronted with a new problem, the first thing we do is to look for an algorithm that would solve it in polynomial time. If we succeed, we are finished. If we do not succeed, we have another way to go—prove that the problem is NP-complete. Then the chance of somebody else coming later and providing an optimal solution to the problem is very small because it is very unlikely that P = NP. In other words, if we can show that our problem is more difficult than one (hence all) of these famous NP-complete problems, then we show strong evidence of the intrinsic difficulty of the problem.

Of course, proving a problem NP-complete does not make it go away. One needs to keep a constructive approach, such as proposing an algorithm that provides a near-optimal solution in polynomial time (or again, proving that no such approximation algorithm exists). This is the subject of Chapter 8.

6.8 Bibliographical notes

As already mentioned, our approach to NP-completeness is original. See the book by Garey and Johnson [38] for a comprehensive treatment of NP-completeness and a famous catalog of NP-complete problems. A very intuitive proof of Cook's theorem is given in the book by Wilf [108]. A theory-oriented approach with Turing machines and complexity results is available in the book

by Papadimitriou [82]. The more adventurous reader can investigate the book by Arora and Barak [4].

Chapter 8

Beyond NP-completeness

At the conclusion of Chapter 6, we stated that proving a problem is NP-complete does not make it go away. The subject of this chapter is to go beyond NP-completeness and to describe the various approaches that can be taken when confronted with an NP-complete problem.

The first approach (see Section 8.1) is the most elegant. When deriving *approximation algorithms*, we search for an approximate solution, but we also guarantee that it is of good quality. Of course, the approximated solution must be found in polynomial time.

The second approach (see Section 8.2) is less ambitious. Given an NP-complete problem, we show how to characterize particular instances that have polynomial complexity.

The third approach (see Section 8.3) often provides useful lower bounds. The idea is to cast the optimization problem under study in terms of a *linear program*. While solving a linear program with integer variables is NP-complete, solving a linear program with rational variables has polynomial complexity (we are restricted to rational variables because of the impossibility of efficiently encoding real numbers). The difficulty is then to reconstruct a solution of the integer linear program from an optimal solution of that program with rational variables. This is not always possible, but this method at least provides a lower bound on any optimal integer solution.

We briefly introduce, in Section 8.4, *randomized algorithms* as a fourth approach that solves "most" instances of an NP-complete problem in polynomial time.

Finally, we provide in Section 8.5 a detailed discussion of *branch-and-bound* and *backtracking* strategies, where one explores the space of all potential solutions in a clever way. While the worst-case exploration may require exponential time, on average, the optimal solution is found in "reasonable" time.

8.1 Approximation results

In this section, we first define polynomial-time approximation algorithms and (fully) polynomial-time approximation schemes (PTAS and FPTAS). Then,

we give some examples of approximation and inapproximability results.

8.1.1 Approximation algorithms

In Chapter 6, we have defined the NP-completeness of problems and exhibited several NP-complete decision problems. As discussed in Section 6.3.4, the target problem is often an optimization problem that has been restricted to a decision problem so that we can prove its NP-completeness.

If the optimal solution of an optimization problem cannot be found in polynomial time, one may want to find an approximate solution in polynomial time.

DEFINITION 8.1. A λ-approximation algorithm is an algorithm whose execution time is polynomial in the instance size and that returns an approximate solution guaranteed to be, in the worst case, at a factor λ away from the optimal solution.

For instance, for each instance $\mathcal{I}$ of a minimization problem, the solution of the approximation algorithm for instance $\mathcal{I}$ must be smaller than or equal to λ times the optimal solution for instance $\mathcal{I}$.

The closer λ to 1, the better the approximation algorithm. We categorize some particular approximation algorithms for which λ is close to 1 as polynomial-time approximation schemes.

DEFINITION 8.2. A *Polynomial-Time Approximation Scheme* (PTAS) is such that for any constant $\lambda = 1+\varepsilon > 1$, there exists a λ-approximation algorithm, i.e., an algorithm that is polynomial in the instance size and guaranteed at a factor λ.

Note that the algorithm may not be polynomial in $1/\varepsilon$ and thus have a high complexity when ε gets close to zero. A Fully PTAS is such that the algorithm is polynomial both in the instance size and in $1/\varepsilon$.

DEFINITION 8.3. A *Fully Polynomial-Time Approximation Scheme*, or FPTAS, is such that for any constant $\lambda = 1 + \varepsilon > 1$, there exists a λ-approximation algorithm that is polynomial in the instance size *and* in $1/\varepsilon$.

The difference between PTAS and FPTAS is simply that the $\forall\varepsilon$ quantifier changes sides. For a PTAS, ε is a fixed constant, so that $2^{\frac{1}{\varepsilon}}$ is a constant as well. On the contrary, the complexity of an FPTAS scheme must be polynomial in $\frac{1}{\varepsilon}$. Of course, having an FPTAS is a stronger property than having a PTAS (i.e., FPTAS $\Rightarrow$ PTAS).

Finally, we define asymptotic PTAS and FPTAS, which add a constant to the approximation scheme. We define formally only the APTAS for a minimization problem, and the definition can easily be extended for maximization problems and AFPTAS.

DEFINITION 8.4. An *Asymptotic Polynomial-Time Approximation Scheme*, or APTAS, is such that for any constant $\lambda = 1 + \varepsilon > 1$, there exists an algorithm, polynomial in the instance size, such that $C_{APTAS} \leqslant \lambda C_{opt} + \beta$ (for a minimization problem), where C_{APTAS} is the cost of the solution of the algorithm, C_{opt} is the cost of an optimal solution, and β is a constant that may depend on ε but should be independent of the problem size.

In the following, we discuss several approximation algorithms, and we show how to prove that an algorithm is an approximation algorithm (possibly an (A)PTAS or (A)FPTAS) or how to prove that a problem cannot be approximated in polynomial time up to any fixed constant λ.

8.1.2 Vertex cover

We consider here the classical vertex cover problem, which was shown to be NP-complete in Section 6.4.3. We discuss a weighted version of this problem in Section 8.3.

We first recall the definition of the vertex cover problem in its optimization problem formulation. Given a graph $G = (V, E)$, we want to find a set of vertices of minimum size that is covering all edges (i.e., any edge in E includes at least one of the vertices of the set).

We consider the following greedy algorithm to solve the problem, called **greedy-vc**. Initialize $S = \emptyset$. Then, while some edges are not covered (i.e., neither of their end vertices are in set S), pick one edge $e = (u, v)$, add both vertices u and v to set S, and mark all edges including u or v as covered. It is clear that **greedy-vc** returns a valid vertex cover, and that it is polynomial in the size of the instance. We now prove that it is a 2-approximation algorithm for the vertex cover optimization problem.

THEOREM 8.1. Greedy-vc *is a 2-approximation algorithm for vertex cover.*

Proof. Let A be the set of edges selected by the greedy algorithm. Two edges of A cannot have a common vertex, and, therefore, the size of the cover of this algorithm is $C_{\textbf{greedy-vc}} = 2|A|$. However, all edges selected greedily are independent, and each of them must be covered in any solution; hence, an optimal solution has at least $|A|$ vertices: $C_{opt} \geqslant |A|$. We deduce that $C_{\textbf{greedy-vc}} \leqslant 2 \times C_{opt}$, which concludes the proof. □

Note that this approximation factor of 2 is achieved, for instance, if G consists of two vertices joined by an edge. There is a polynomial-time algorithm that is a $2 - \frac{\log(\log(|V|))}{2\log(|V|)}$ approximation [79], but, for instance, we do not know any polynomial-time algorithms that would be a 1.99 approximation (the problem is still open).

8.1.3 Traveling salesman problem (TSP)

Let $G = (V, E)$ be a complete graph and $w : E \to \mathbb{N}$ be a cost function. The TSP problem consists of finding a cycle $\mathcal{C}$ going through each vertex once and only once, with $\sum_{e \in \mathcal{C}} w(e) \leqslant k$. The decision problem, in which k is a fixed integer, is NP-complete, as mentioned in Section 6.4.5. For the optimization problem, the goal is to minimize k.

First, we prove that TSP cannot be approximated unless P = NP. Then, we propose an approximation algorithm in the particular case where the cost function follows the triangle inequality.

Inapproximability of TSP

THEOREM 8.2. *For any constant $\lambda \geqslant 1$, there does not exist any λ-approximation algorithm for TSP unless P = NP.*

To prove such a result, the methodology is often as follows. The idea consists of assuming that there is a λ-approximation algorithm for the target problem (by definition, this is a polynomial-time algorithm). Then, one uses this approximation algorithm to solve in polynomial time a problem that is known to be NP-complete. For TSP, we show how any instance of problem Hamiltonian Cycle (HC, see Definition 6.4) can be solved in polynomial time using any approximation algorithm for TSP.

Proof. Let us assume that there is a λ-approximation algorithm for TSP. We consider an instance $\mathcal{I}_{hc}$ of HC, which is a graph $G = (V, E)$, with $n = |V|$. Then, we build an instance $\mathcal{I}_{tsp}$ of TSP as follows. In the complete graph, we build a cost function such that $w(e) = 1$ if $e \in E$, and $w(e) = \lambda n + 1$ otherwise. The size of $\mathcal{I}_{tsp}$ is obviously polynomial in the size of $\mathcal{I}_{hc}$.

We use the λ-approximation algorithm to solve $\mathcal{I}_{tsp}$. Let C_{algo} be its solution. This solution is such that $C_{algo} \leqslant \lambda C_{opt}$, where C_{opt} is the optimal solution.

We consider the two following cases:

- If $C_{algo} \geqslant \lambda n + 1$, then $C_{opt} > n$. This means that instance $\mathcal{I}_{hc}$ has no solution. Indeed, a Hamiltonian Cycle for $\mathcal{I}_{hc}$ would be a solution of cost n for $\mathcal{I}_{tsp}$.
- Otherwise, $C_{algo} < \lambda n + 1$, and therefore the solution of $\mathcal{I}_{tsp}$ is not using any edge not in E (otherwise, the cost would be at least $\lambda n + 1$). This solution is therefore a Hamiltonian Cycle for $\mathcal{I}_{hc}$, which means that instance $\mathcal{I}_{hp}$ has a solution.

Therefore, the result of the algorithm for $\mathcal{I}_{tsp}$ allows us to conclude whether there is a Hamiltonian Cycle in $\mathcal{I}_{hc}$, which concludes the proof. □

Note that we assumed that λ is constant, but we can even have $\lambda = 1 + 2^{-n}$, since the algorithm would still be polynomial in the instance size (λ can be

encoded in logarithmic size, hence in $O(n)$). However, Theorem 8.2 does not forbid the existence of a $2^{2^{-n}}$-approximation algorithm.

Approximation algorithm with triangle inequality

We now assume that the cost function w satisfies the triangle inequality, i.e., for all vertices $v_1, v_2, v_3 \in V$, $w(v_1, v_3) \leqslant w(v_1, v_2) + w(v_2, v_3)$.

The approximation algorithm **spanning-tsp** works as follows. First, we build a minimum spanning tree T of the graph G, which can be done in polynomial time with a greedy algorithm (remove edges by nonincreasing costs while keeping a connected graph, see Section 3.4). Then, we perform a tree traversal of T (once a node u is visited, one completely visits the subtree rooted at one of the children of u before starting to visit any subtree rooted at another child). Each edge of T is visited exactly twice. We extract a solution for TSP, i.e., a Hamiltonian Cycle, by recording the order in which vertices are visited for the first time. From this ordered list of vertices, we build a cycle by taking the edges that link consecutive vertices (recall that the graph is complete).

We now prove that this algorithm is a 2-approximation.

THEOREM 8.3. Spanning-tsp *is a 2-approximation algorithm for the traveling salesman problem with the triangle inequality.*

Proof. The optimal cost C_{opt} is at least equal to the sum of the costs of the edges in the minimum spanning tree T, denoted by $w(T)$. Indeed, an optimal solution is a cycle. If we remove an edge from an optimal solution, we obtain a spanning tree, and T is a spanning tree of minimum weight. Therefore, $C_{opt} \geqslant w(T)$.

Now, we consider the cost of the solution returned by the algorithm. We denote this solution by S and its cost by $C_{\textbf{spanning-tsp}}$. Let O be the order in which the vertices are visited in the traversal of T. Vertices that are not leaves of T appear several times in O. S is obtained from O by keeping only the first occurrence of each vertex. Because of the triangular inequality, deleting a vertex from O does not increase the cost of the associated path. (Suppose we delete the vertex y in the sequence (x, y, z) of O; this is equivalent to replacing the two edges (x, y) and (y, z) with the single edge (x, z).) Hence, $C_{\textbf{spanning-tsp}}$ is less than or equal to the cost of the path associated with O. Furthermore, the path associated with O contains each edge exactly twice, and its cost is exactly $2 \times w(T)$. Therefore, $C_{\textbf{spanning-tsp}} \leqslant 2C_{opt}$, which proves the approximation result. □

8.1.4 Bin packing

In this section, we introduce a new classical problem that is the bin packing problem.

DEFINITION 8.5 (BP – Bin Packing)**.** Given n rational numbers (also called objects) $a_1, \ldots, a_n$, with $0 < a_i \leqslant 1$, for $1 \leqslant i \leqslant n$, can we partition them in k bins $B_1, \ldots, B_k$ of capacity 1, i.e., for each $1 \leqslant j \leqslant k$, $\sum_{i \in B_j} a_i \leqslant 1$?

First, we prove the NP-completeness of this problem, then we exhibit several approximation results.

NP-completeness of BP

THEOREM 8.4. *BP is NP-complete.*

Proof. It is straightforward to see that BP is in NP: A certificate is the list, for each bin, of the indices of the numbers it contains.

The reduction comes from 2-PARTITION. We consider an instance $\mathcal{I}_1$ of 2-PARTITION, with n integers $b_1, \ldots, b_n$. We build the following instance $\mathcal{I}_2$ of BP: For $1 \leqslant i \leqslant n$, $a_i = \frac{2b_i}{S}$, with $S = \sum_{i=1}^{n} b_i$, and we set $k = 2$.

It is then straightforward to see that the size of the new instance is polynomial and to check the equivalence of solutions. □

Inapproximability of BP

THEOREM 8.5. *For all $\varepsilon > 0$, there does not exist any $(\frac{3}{2}-\varepsilon)$-approximation algorithm for BP unless P = NP.*

Proof. Let us assume that there is a $(\frac{3}{2} - \varepsilon)$-approximation algorithm for BP. We then exhibit a polynomial algorithm to solve 2-PARTITION.

Given an instance of 2-PARTITION, we execute the algorithm for BP with the a_i as defined earlier. If there exists a 2-PARTITION of the b_i, the algorithm returns at most $2 \times (\frac{3}{2} - \varepsilon) = 3 - 2\varepsilon$ bins, so it returns two bins. Otherwise, the algorithm returns a solution with at least three bins. Thanks to the polynomial approximation algorithm, we can solve 2-PARTITION in polynomial time, which implies that P = NP. This concludes the proof. □

Approximation algorithms for BP

We start with a simple greedy algorithm in which we select objects in a random order, and, at each step, we place the object either in the last used bin where it fits (**next-fit** algorithm) or in the first used bin where it fits (**first-fit** algorithm); otherwise (i.e., the object is not fitting in any used bin), we create a new bin and place the object in this new bin. We prove below that **next-fit** (and, hence, **first-fit**) is a 2-approximation algorithm for the BP problem.

THEOREM 8.6. Next-fit *is a 2-approximation algorithm for BP.*

Proof. Let $A = \sum_{i=1}^{n} a_i$. We have a lower bound on the cost of the optimal solution (the number of bins used by the optimal solution): $C_{opt} \geqslant \lceil A \rceil$.

Now we bound the cost of **next-fit** as follows. If we consider two consecutive bins, the sum of the objects that they contain is strictly greater than 1;

otherwise, we would not have created a new bin. Therefore, if $C_{\textbf{next-fit}} = K$, and B_k is the k-th bin of the solution returned by **next-fit**, for $1 \leqslant k \leqslant K$, then by summing the contents of two consecutive bins, we get

$$\sum_{k=1}^{K-1} \left(\sum_{i \in B_k} a_i + \sum_{i \in B_{k+1}} a_i \right) > K - 1 \; .$$

Moreover, by definition of A, we have $\sum_{k=1}^{K-1} \left(\sum_{i \in B_k} a_i + \sum_{i \in B_{k+1}} a_i \right) \leqslant 2A$, and, therefore, $K - 1 < 2A \leqslant 2\lceil A \rceil$. Finally, $C_{\textbf{next-fit}} = K \leqslant 2\lceil A \rceil \leqslant 2C_{opt}$, which concludes the proof. □

Note that the approximation ratio is tight for the **next-fit** algorithm. Consider an instance of BP with $4n$ objects such that $a_{2i-1} = \frac{1}{2}$ and $a_{2i} = \frac{1}{2n}$, for $1 \leqslant i \leqslant 2n$. Then, if **next-fit** chooses the objects in the sequential order, its solution uses $2n$ bins (one object a_{2i-1} and one object a_{2i} in each bin), while the optimal solution uses only $n + 1$ bins (for $1 \leqslant i \leqslant 2n$, the $2n$ objects a_{2i} in one bin and two objects a_{2i-1} in each of the other n bins).

The previous algorithms can be qualified as *online* algorithms because no sorting is done on the objects, and we can pack them in the bins when they arrive, on the fly. If we have the knowledge of all objects before executing the algorithm, we can refine the algorithm by sorting the objects beforehand. Such algorithms are called *offline* algorithms. The **first-fit-dec** algorithm sorts the objects by nonincreasing size (**dec** stands for decreasing), and then it applies the **first-fit** rule: The object is placed in the first used bin in which it fits; otherwise, a new bin is created.

THEOREM 8.7. *$C_{\textbf{first-fit-dec}} \leqslant \frac{3}{2} C_{opt} + 1$, where $C_{\textbf{first-fit-dec}}$ is the cost returned by the* **first-fit-dec** *algorithm, and C_{opt} is the optimal cost.*

Note that this is not an approximation algorithm as defined above because of the "+1" in the expression, which corresponds to one extra bin that the **first-fit-dec** algorithm may use. This is rather an *asymptotic* approximation algorithm, which is similar to an A(F)PTAS scheme. Indeed, the constant 1 is independent of the problem size, and the algorithm is asymptotically a $\frac{3}{2}$-approximation.

Proof. We split the a_i in four categories:

$$A = \left\{ a_i > \frac{2}{3} \right\} \quad B = \left\{ \frac{2}{3} \geq a_i > \frac{1}{2} \right\} \quad C = \left\{ \frac{1}{2} \geq a_i > \frac{1}{3} \right\} \quad D = \left\{ \frac{1}{3} \geq a_i \right\}$$

Case 1: There is at least one bin containing only objects of category D in the solution of **first-fit-dec**. In this case, at most one bin (the last one) has a sum of objects of less than $\frac{2}{3}$, and it contains only objects of category D. Indeed, if the objects of D of the last bin have not fit in the previous bins, it means that each bin (except the last one) has a sum of objects of at least $\frac{2}{3}$.

Therefore, if we ignore the last bin, $C_{opt} \geqslant \sum_{i=1}^{n} a_i \geqslant \frac{2}{3}(C_{\textbf{first-fit-dec}} - 1)$, which concludes the proof for this case.

Case 2: There is no bin with only objects of category D. In this case, we can ignore the objects of category D because they are added into the bins at the end of the algorithm, and they do not lead to the creation of new bins. We now prove that the solution of **first-fit-dec** for the objects of A, B, and C is optimal. Indeed, in any solution, objects of A are alone in a bin, and there are at most two objects of B and C in a bin, with at most one object of B. The **first-fit-dec** algorithm is placing first each object A and B in a separate bin, then it does the best matching of objects C, because they are placed in the bins by decreasing order. In this case, **first-fit-dec** is optimal. □

Note that the reasoning does not hold if the categories are made differently, with, for instance, $\frac{1}{4}$ instead of $\frac{1}{3}$. Indeed, we can then fit three objects of category C in a single bin, and the reasoning does not hold anymore. However, we point out that it is also possible to prove that $C_{\textbf{first-fit-dec}} \leqslant \frac{11}{9}C_{opt} + 1$, and we refer to [112] for further details. The idea of the proof is similar, but more categories of objects are considered, and the algorithm turns out to be much more complex.

Without allowing an extra bin, we can finally prove that **first-fit-dec** is a $\frac{3}{2}$-approximation algorithm.

THEOREM 8.8. First-fit-dec *is a* $\frac{3}{2}$*-approximation algorithm for the bin packing problem.*

Proof. Let $k = C_{\textbf{first-fit-dec}}$ be the cost returned by the **first-fit-dec** algorithm, and let $j = \left\lceil \frac{2}{3}k \right\rceil$. Bins are numbered from 1 to k, and we consider two cases.

Case 1: If bin j contains an object a_i such that $a_i > \frac{1}{2}$, then if $j' < j$, there is an object $a_{i'}$ in bin j' such that $a_{i'} \geqslant a_i > \frac{1}{2}$. This is true for $1 \leqslant j' < j$, and, therefore, there are at least j objects of size greater than $\frac{1}{2}$ that should be placed in distinct bins. This implies that the optimal cost C_{opt} is greater than j.

Case 2: None of the bins $j' \geqslant j$ contains any object of size strictly greater than $\frac{1}{2}$; there are at least two objects per bin, except for bin k that may contain only one object, hence $2(k-j)+1$ objects in bins $j, j+1, \ldots, k$. None of these objects fits into bins $1, 2, \ldots, j-1$, by definition of **first-fit-dec**. We show below that $2(k-j)+1 \geqslant j-1$, and by combining $j-1$ of these objects with each of the first $j-1$ bins, we obtain that the sum of the a_is is strictly greater than $j-1$, i.e., C_{opt} is greater than j. In order to prove the inequality $2(k-j)+1 \geqslant j-1$, we show that $j = \left\lceil \frac{2}{3}k \right\rceil \leqslant \frac{2}{3}(k+1)$. Let $y = j - \frac{2}{3}k$. Note that j and k are integers, and $0 \leqslant y < 1$. Moreover, $k = \frac{3}{2}j - \frac{3}{2}y$. If j is even, then $\frac{3}{2}j$ is an integer; therefore, $\frac{3}{2}y$ is an integer strictly smaller than $\frac{3}{2}$, i.e., $\frac{3}{2}y \leqslant 1$ and $y \leqslant \frac{2}{3}$. Otherwise, $\frac{3}{2}y + \frac{1}{2}$ is an integer, and because $y < 1$, we have $\frac{3}{2}y + \frac{1}{2} < 2$, i.e., $\frac{3}{2}y + \frac{1}{2} \leqslant 1$ and $y \leqslant \frac{1}{3}$. Altogether, $\left\lceil \frac{2}{3}k \right\rceil = j = \frac{2}{3}k + y \leqslant \frac{2}{3}k + \frac{2}{3}$.

In both cases, we have

$$C_{opt} \geqslant j = \left\lceil \frac{2}{3}k \right\rceil \geqslant \frac{2}{3} C_{\textbf{first-fit-dec}} ,$$

which concludes the proof. □

8.1.5 2-PARTITION

We discuss approximation algorithms for the 2-PARTITION problem. The optimization problem associated with 2-PARTITION is the following: Given n integers $a_1, \ldots, a_n$, find a subset I of $\{1, \ldots, n\}$ such that $\max\left(\sum_{i \in I} a_i, \sum_{i \notin I} a_i\right)$ is minimum. Note that the minimum is always at least $\max\left(P_{max}, P_{sum}/2\right)$, where $P_{max} = \max_{1 \leqslant i \leqslant n} a_i$ and $P_{sum} = \sum_{i=1}^{n} a_i$.

This problem is similar to a scheduling problem with two identical processors. There are n independent tasks $T_1, \ldots, T_n$, and task T_i $(1 \leqslant i \leqslant n)$ can be executed on one of the two processors in time a_i. The goal is to minimize the total execution time. The processors are denoted by P_1 and P_2.

We start by analyzing two greedy algorithms for this problem. Then, we show how to derive a PTAS for 2-PARTITION and even an FPTAS.

Greedy algorithms

The two natural greedy algorithms are the following. We choose tasks in a random order (online algorithm, **greedy-online**) or sorted by nonincreasing execution time (offline algorithm, **greedy-offline**), and we assign the chosen task to the processor that has the lowest current load.

The idea of sorting in the offline algorithm is that a task with a large execution time, if considered at the end of the algorithm, may unbalance the entire execution. However, the offline version requires that all execution times are known beforehand. The online algorithm can be applied in a problem where tasks arrive dynamically (for instance, scheduling user jobs on a biprocessor server).

THEOREM 8.9. Greedy-online *is a* $\frac{3}{2}$*-approximation algorithm, and* **greedy-offline** *is a* $\frac{7}{6}$*-approximation algorithm for the 2-PARTITION problem. Moreover, these approximation ratios are tight.*

Proof. First, we consider the **greedy-online** algorithm. Let us assume that processor P_1 finishes the execution at time $M_1 \geqslant M_2$ (where M_2 is the time at which P_2 finishes its execution), and that T_j is the last task executed on P_1. We have $M_1 + M_2 = P_{sum}$. Moreover, since the greedy algorithm chose processor P_1 to execute task T_j, it means that $M_1 - a_j \leqslant M_2$; otherwise, T_j would have been scheduled on P_2. Finally, the cost of **greedy-online** is such that:

$$C_{\textbf{online}} = M_1 = \frac{1}{2}(M_1 + (M_1 - a_j) + a_j) \leqslant \frac{1}{2}(M_1 + M_2 + a_j) = \frac{1}{2}(P_{sum} + a_j),$$

and since $C_{opt} \geqslant P_{sum}/2$ and $C_{opt} \geqslant a_i$ for $1 \leqslant i \leqslant n$, we have $C_{\textbf{online}} \leqslant C_{opt} + \frac{1}{2}C_{opt} = \frac{3}{2}C_{opt}$, which concludes the proof.

For the offline version of the greedy algorithm, we start as before, but we refine the inequality $a_j \leqslant C_{opt}$. If $a_j \leqslant \frac{1}{3}C_{opt}$, we obtain the approximation ratio of the theorem, i.e., $C_{\textbf{offline}} \leqslant \frac{7}{6}C_{opt}$. We focus now on the case where $a_j > \frac{1}{3}C_{opt}$. Then, $j \leqslant 4$. Indeed, if a_j were the fifth task, because the tasks are sorted by nonincreasing execution times, there would be at least five tasks of time at least $\frac{1}{3}C_{opt}$, and any schedule would need to schedule at least three of these tasks on the same processor, leading to an execution time strictly greater than C_{opt}, and hence a contradiction. Then, we note that, in this case, the cost $C_{\textbf{offline}}$ when we restrict to the scheduling of the first four tasks is identical to the cost when scheduling all tasks. Finally, it is easy to check (exhaustively) that **greedy-offline** is optimal when scheduling at most four tasks. We conclude that $C_{\textbf{offline}} = C_{opt}$ in this case, which ends the proof.

Finally, we prove that the ratios are tight. For **greedy-online**, we consider an instance with two tasks of time 1 and one task of time 2. The greedy algorithm schedules the tasks in time 3 (each task of time 1 on a distinct processor, then the task of time 2 after one of those), while the optimal algorithm takes a time 2 (with the two first tasks on the same processor). For **greedy-offline**, we consider an instance with two tasks of time 3 and three tasks of time 2. The greedy algorithm schedules each task of time 3 on a distinct processor, leading to a total execution time of 7, while the optimal solution consists of grouping those two tasks on the same processor, with a total time of 6. □

PTAS: A $(1+\varepsilon)$-approximation algorithm

THEOREM 8.10. *$\forall \varepsilon > 0$, there is a $(1+\varepsilon)$-approximation algorithm for the 2-PARTITION problem. In order words, 2-PARTITION has a PTAS.*

Proof. We consider an instance $\mathcal{I}$ of 2-PARTITION, $a_1, \ldots, a_n$ (recall that the a_is can be interpreted as the execution time of tasks), and $\varepsilon > 0$.

We classify the tasks into two categories. Let $L = \max(P_{max}, P_{sum}/2)$. The *big* tasks are in the set $T_{big} = \{i \mid a_i > \varepsilon L\}$, while the *small* tasks are in the set $T_{small} = \{i \mid a_i \leqslant \varepsilon L\}$. We consider an instance $\mathcal{I}^*$ of the problem with the tasks of T_{big}, and $\left\lfloor \frac{S}{\varepsilon L} \right\rfloor$ tasks of identical size εL, where $S = \sum_{i \in T_{small}} a_i$.

The proof goes as follows. We show that the optimal schedule for instance $\mathcal{I}^*$ has a cost C^*_{opt} close to the cost C_{opt} of the optimal schedule for instance $\mathcal{I}$, i.e., $C^*_{opt} \leqslant (1+\varepsilon)C_{opt}$. Moreover, it is possible to compute the optimal schedule for instance $\mathcal{I}^*$ in a polynomial time. Building upon this schedule, we finally construct a solution to the original instance $\mathcal{I}$, with a guaranteed cost.

First, we prove that $C^*_{opt} \leqslant (1+\varepsilon)C_{opt}$. Let *opt* be an optimal schedule for instance $\mathcal{I}$, of cost C_{opt}. Then, let S_1 (resp. S_2) be the sum of the small tasks in this optimal schedule on processor P_1 (resp. P_2). We build a new schedule *sched** in which the big tasks of the optimal schedule *opt* remain on

the same processors, but small tasks are replaced with $\left\lceil \frac{S_i}{\varepsilon L} \right\rceil$ tasks of size εL on processor P_i, for $i = 1, 2$. Because

$$\left\lceil \frac{S_1}{\varepsilon L} \right\rceil + \left\lceil \frac{S_2}{\varepsilon L} \right\rceil \geqslant \left\lfloor \frac{S_1 + S_2}{\varepsilon L} \right\rfloor = \left\lfloor \frac{S}{\varepsilon L} \right\rfloor ,$$

we have scheduled at least as many tasks of size εL as the total number of small tasks in instance $\mathcal{I}^*$. Moreover, the execution time on processor P_i, for $i = 1, 2$, has been increased of at most

$$\left\lceil \frac{S_i}{\varepsilon L} \right\rceil \times \varepsilon L \ - \ S_i \leqslant \varepsilon L ,$$

which means that the cost of this schedule is such that $C^*_{sched} \leqslant C_{opt} + \varepsilon L$. Moreover, this schedule is a schedule for instance $\mathcal{I}^*$ and, therefore, $C^*_{sched} \geqslant C^*_{opt}$. Finally, $C^*_{opt} \leqslant C_{opt} + \varepsilon L \leqslant C_{opt} + \varepsilon \times C_{opt}$, which concludes the proof that $C^*_{opt} \leqslant (1 + \varepsilon) C_{opt}$.

Next, we discuss how to find an optimal schedule for instance $\mathcal{I}^*$. First, we provide a bound on the number of tasks in $\mathcal{I}^*$. Because we replaced small tasks of $\mathcal{I}$ with tasks of size εL, we have not increased the total execution time, which is at most $P_{sum} \leqslant 2L$. Each task of $\mathcal{I}^*$ has an execution time of at least εL (small tasks), so there are at most $\frac{2L}{\varepsilon L} = \frac{2}{\varepsilon}$ tasks. Note that this is a constant number because ε is a constant. Moreover, we note that the size of $\mathcal{I}^*$ is polynomial in the size of instance $\mathcal{I}$ (because the size of $\mathcal{I}^*$ is a constant). We can optimally schedule $\mathcal{I}^*$ by trying all $2^{\frac{2}{\varepsilon}}$ possible schedules and keeping the best one. Of course, this algorithm is not polynomial in $1/\varepsilon$, but it is polynomial in the size of the instance $\mathcal{I}$ because it is a constant.

Now we have an optimal schedule *opt** for instance $\mathcal{I}^*$, of cost C^*_{opt}, and we aim to build a schedule *sched* for instance $\mathcal{I}$. For $i = 1, 2$, we let $L^*_i = B^*_i + S^*_i$ be the total execution time of processor P_i in the schedule *opt**, where B^*_i (resp. S^*_i) is the time spent on big (resp. small) tasks. Then, we build the schedule *sched* in which the big tasks are kept on the same processor as in *opt**, and we greedily assign small tasks to processors. First, we assign small tasks to processor P_1 until their processing time does not exceed $S^*_1 + 2\varepsilon L$. Then, we schedule the remaining small tasks to processor P_2. Let us prove now that once all small tasks have been scheduled, the execution time has not increased by more than $2\varepsilon L$.

Because small tasks have a size of at most εL, the greedy algorithm assigns at least a total of $S^*_1 + \varepsilon L$ small tasks on processor P_1. Then, there are at most a total of $S - (S^*_1 + \varepsilon L)$ small jobs to assign to processor P_2. However, by construction of $\mathcal{I}^*$, we have $S^*_1 + S^*_2 = \varepsilon L \left\lfloor \frac{S}{\varepsilon L} \right\rfloor > S - \varepsilon L$ and, therefore, $S - (S^*_1 + \varepsilon L) \leqslant S^*_2$, and the execution time of P_2 in the new schedule *sched* is not greater than in the schedule *opt**.

The schedule *sched* is a schedule for instance $\mathcal{I}$, which is built in polynomial time. The cost of this schedule is at most $C_{sched} \leqslant C^*_{opt} + 2\varepsilon L$. We use the

previous result that $C^*_{opt} \leqslant (1+\varepsilon)C_{opt}$ and the fact that $L \leqslant C_{opt}$ to conclude that $C_{sched} \leqslant C^*_{opt} + 2\varepsilon L \leqslant (1+3\varepsilon)C_{opt}$. This is true for all ε, so we can apply this algorithm with $\varepsilon/3$ to obtain the desired ratio. □

Note that a simpler proof can be done by using an optimal schedule for the big tasks, of cost C_{big}, and the **greedy-online** algorithm introduced above. Once the small tasks have been scheduled greedily on the two processors, there are two cases. If the total time has not changed, i.e., it is C_{big}, it is optimal. Otherwise, the processor that ends the execution is executing a small task a_j. This means that before the greedy choice of scheduling task a_j onto this processor, the finishing time of the processor was less than $P_{sum}/2$; otherwise, task a_j would have been assigned to the other processor because of the greedy choice. Finally, the cost of the schedule returned by this algorithm is at most $P_{sum}/2 + a_j \leqslant L + \varepsilon L \leqslant (1+\varepsilon)C_{opt}$.

A PTAS provides an approximate solution that is as close to the optimal as one wants. The only downside is that the algorithm running time increases with the quality of the approximate solution. Some readers may thus be puzzled by the idea of having a PTAS or an FPTAS for an NP-complete problem whose objective function takes values in a discrete set, such as 2-PARTITION. Indeed, a PTAS, for such a problem, enables one to obtain an optimal solution whenever one is ready to pay the cost. Let us consider any given instance $\mathcal{I}$ of 2-PARTITION. Let S be the sum of the elements of $\mathcal{I}$. If $\varepsilon < \frac{1}{S}$, then any $1+\varepsilon$ approximation produces an optimal solution. Indeed, $(1+\varepsilon)C_{opt} < (1+\frac{1}{S})C_{opt} \leqslant C_{opt} + 1$ because $C_{opt} \leqslant S$ and because the objective function can take only integral values. This may be surprising at first sight, but it does not contradict anything we have written so far. One should not forget that the running time of an FPTAS is polynomial in the size of $\frac{1}{\varepsilon}$, that is, in our example, in the size of S. The running time of a PTAS can even be exponential in the size of $\frac{1}{\varepsilon}$. Finding the optimal solution for 2-PARTITION in time exponential in the size of S is quite simple. One generates all the subsets of $\mathcal{I}$ and computes the sum of the elements of each subset. If $\mathcal{I}$ includes n elements, there are $2^n = O(2^S)$ subsets of $\mathcal{I}$. The sum of the elements of each of them is computed in time $O(n)$ and thus $O(S)$. Therefore, readers should not be surprised that, for a given value of ε, an algorithm whose running time is polynomial in the size of the instance can find an optimal solution to 2-PARTITION.

FPTAS for 2-PARTITION

We have provided a PTAS for 2-PARTITION, but the algorithm finds an optimal schedule for instance $\mathcal{I}^*$ (i.e., an optimal schedule of the big tasks), and this is not polynomial in $1/\varepsilon$. Below, we provide an FPTAS, i.e., a $(1+\varepsilon)$-approximation algorithm that is polynomial in the size of $\mathcal{I}$ and in $1/\varepsilon$.

THEOREM 8.11. *$\forall \varepsilon > 0$, there is a $(1+\varepsilon)$-approximation algorithm for the 2-PARTITION problem that is polynomial in $1/\varepsilon$. In order words, 2-PARTITION has an FPTAS.*

Proof. The idea of the proof is to encode the schedules as *vector sets*, in which the first (resp. second) element of a vector represents the running time of the first (resp. second) processor. Formally, for $1 \leqslant k \leqslant n$, VS_k is the set of vectors representing schedules of tasks $a_1, \ldots, a_k$: $VS_1 = \{[a_1, 0], [0, a_1]\}$, and we build VS_k from VS_{k-1} as follows. For all $[x, y] \in VS_{k-1}$, we add $[x + a_k, y]$ and $[x, y + a_k]$ to VS_k. The optimal schedule is represented by a vector $[x, y] \in VS_n$, and it is such that $\max(x, y)$ is minimized.

The approximation algorithm enumerates all possible schedules, but some of them are discarded on the fly so that we keep a polynomial algorithm.

Let $\Delta = 1 + \frac{\varepsilon}{2n}$. We partition the square $P_{sum} \times P_{sum}$ following the power of Δ, from 0 to Δ^M. We have $M = \lceil \log_\Delta(P_{sum}) \rceil = \left\lceil \frac{\ln(P_{sum})}{\ln(\Delta)} \right\rceil \leqslant \left\lceil \left(1 + \frac{2n}{\varepsilon}\right) \ln(P_{sum}) \right\rceil$. Indeed, note that if $z \geqslant 1$, then $\ln(z) \geqslant 1 - \frac{1}{z}$.

The idea of the algorithm consists of building the vector sets but adding a new vector to a set only if there are no other vectors in the same square of the partitioned $P_{sum} \times P_{sum}$ square. Because M is polynomial in $1/\varepsilon$ and in $\ln(P_{sum})$, and the size of instance $\mathcal{I}$ is greater than $\ln(P_{sum})$, the algorithm is polynomial both in the size of $\mathcal{I}$ and in $1/\varepsilon$. We need to prove that this algorithm is a $(1+\varepsilon)$-approximation to conclude the proof.

First, let us formally describe the algorithm. Initially, $VS_1^\# = VS_1$. Then, for $2 \leqslant k \leqslant n$, we build $VS_k^\#$ from $VS_{k-1}^\#$ as follows. For all $[x, y] \in VS_{k-1}^\#$, we add $[x + a_k, y]$ (resp. $[x, y + a_k]$) to $VS_k^\#$ if and only if there is no vector from $VS_k^\#$ in the same square. Note that two vectors $[x_1, y_1]$ and $[x_2, y_2]$ are in the same square if and only if $\frac{x_1}{\Delta} \leqslant x_2 \leqslant \Delta x_1$ and $\frac{y_1}{\Delta} \leqslant y_2 \leqslant \Delta y_1$.

We keep at most one vector per square at each step, which gives an overall complexity in $n \times M^2$, which is polynomial both in the size of instance $\mathcal{I}$ and in $1/\varepsilon$.

Next, we prove that for all $1 \leqslant k \leqslant n$ and $[x, y] \in VS_k$ there exists $[x^\#, y^\#] \in VS_k^\#$ such that $x^\# \leqslant \Delta^k x$ and $y^\# \leqslant \Delta^k y$. The proof is done recursively. The result is trivial for $k = 1$. If we assume that the result is true for $k - 1$, then let us consider $[x, y] \in VS_k$. Either $x = u + a_k$ and $y = v$ (case 1), or $x = u$ and $y = v + a_k$ (case 2), with $[u, v] \in VS_{k-1}$. By recursion hypothesis, there exists $[u^\#, v^\#] \in VS_{k-1}^\#$ with $u^\# \leqslant \Delta^{k-1} u$ and $v^\# \leqslant \Delta^{k-1} v$. For case 1, note that $[u^\# + a_k, v^\#]$ may not be in $VS_k^\#$, but we know that there is at least one vector in the same square in $VS_k^\#$; there exists $[x^\#, y^\#] \in VS_k^\#$ such that $x^\# \leqslant \Delta \left(u^\# + a_k\right)$ and $y^\# \leqslant \Delta v^\#$. Finally, we have $x^\# \leqslant \Delta^k u + \Delta a_k \leqslant \Delta^k (u + a_k) = \Delta^k x$ and $y^\# \leqslant \Delta v^\# \leqslant \Delta^k y$, and case 2 is symmetrical. This proves the result.

For $k = n$, we can deduce that $\max(x^\#, y^\#) \leqslant \Delta^n \max(x, y)$. There remains to be proven that $\Delta^n \leqslant (1+\varepsilon)$, where $\Delta^n = \left(1 + \frac{\varepsilon}{2n}\right)^n$. We rearrange the last

inequality and study the function $f(z) = \left(1 + \frac{z}{n}\right)^n - 1 - 2z$, for $0 \leqslant z \leqslant 1$, $f'(z) = \frac{1}{n} n \left(1 + \frac{z}{n}\right)^{n-1} - 2$. We deduce that f is a convex function, and that its minimum is reached in $\lambda_0 = n\left(\sqrt[n-1]{2} - 1\right)$. Moreover, $f(0) = -1$ and $f(1) = \left(1 + \frac{1}{n}\right)^n - 3 \leqslant 0$. Because f is convex, and $f(z) \leqslant 0$ for $z = 0$ and $z = 1$, we can deduce that $f(z) \leqslant 0$ for $0 \leqslant z \leqslant 1$. This concludes the proof. □

8.2 Polynomial problem instances

When confronted with an NP-complete problem, one algorithmic solution consists of finding good approximation algorithms. While some problems may have good approximation schemes, such as PTAS or FPTAS (see Section 8.1), some problems cannot be approximated. However, with a slight change of the problem parameters (constant value for a parameter, different rule of the game, etc.), it may be possible to find a good approximation algorithm or even to be able to solve the problem in pseudopolynomial or polynomial time.

The analysis of a problem is comprehensive when we are able to identify at which point the problem becomes NP-complete and then at which point the problem cannot be approximated any more. We refine the problem complexity as follows:

- The class P consists of all optimization problems that can be solved in polynomial time.
- The class FPTAS consists of all optimization problems that have an FPTAS, and it contains P.
- The class PTAS consists of all optimization problems that have a PTAS, and it contains FPTAS.
- The class APX consists of all optimization problems that have a polynomial-time approximation algorithm with a constant ratio, and it contains PTAS.
- Finally, the class NP contains APX: Some problems may be in NP but not in APX.

We also consider the class of problems that can be solved in pseudopolynomial time, PPT. This class includes P but none of the other previous classes. Some problems that can be solved in pseudopolynomial time may not have an FPTAS or may not even be in APX. The problems of (i) finding a pseudopolynomial-time algorithm to solve the problem exactly and (ii) finding good polynomial-time approximation algorithms are not correlated.

In the following, we illustrate how the problem can move from one category to another when parameters are modified. In particular, we check whether the problem can be solved in polynomial time or in pseudopolynomial time, and if there is no polynomial-time algorithm to solve the problem, we investigate polynomial-time approximation algorithms.

8.2.1 Partitioning problems

First, we provide both the optimization and decision versions of the partitioning problem that we consider, and then we investigate variants of the problem.

Optimization problem (PART-OPT). Let $a_1, \ldots, a_n$ be n positive integers. The goal is to partition these integers into p subsets $A_1, \ldots, A_p$, in order to minimize the maximum (over all subsets) of the sum of the integers in a subset:

$$\min \left(\max_{1 \leqslant j \leqslant p} \sum_{i \in A_j} a_i \right) .$$

Decision problem (PART-DEC). The associated decision problem is the following: Let $a_1, \ldots, a_n$ be n positive integers. Given a bound K, is it possible to partition these integers into p subsets $A_1, \ldots, A_p$, such that the sum of the integers in each subset does not exceed K? In other words,

$$\text{for all } \; 1 \leqslant j \leqslant p, \quad \sum_{i \in A_j} a_i \leqslant K \; .$$

We can easily prove, from a reduction from 3-PARTITION, that PART-DEC is NP-complete in the strong sense. No pseudopolynomial algorithm is known to solve PART-DEC. However, PART-OPT is a classical scheduling problem. The goal is to schedule n independent tasks onto p processors, where a_i is the execution time of task T_i, for $1 \leqslant i \leqslant n$, and the goal is to minimize the total execution time. There is a PTAS to approximate this problem [49].

One way to simplify the problem is to restrict it to the case $p = 2$. The problem is then equivalent to 2-PARTITION, and it can be solved in pseudopolynomial time using a dynamic-programming algorithm (see Section 6.2.1). Moreover, this problem is in the class FPTAS, as was shown in Section 8.1.5.

In order to identify polynomial instances of this problem, we consider the following variants:

1. We consider the case in which all integers are equal, i.e., $a_1 = a_2 = \cdots = a_n = a$. In this case, we can find the solution to the optimization problem, which is simply $\left\lceil \frac{n}{p} \right\rceil \times a$. Therefore, we also can solve the decision problem in polynomial time, even in constant time.

2. We change the rule of the game. The subsets must contain only continuous elements, for instance, $[a_i, a_{i+1}, \ldots, a_{i'}]$. The subsets are then

intervals, and the problem can be solved in polynomial time. It is the classical chains-on-chains partitioning problem (see Chapter 11), which can be solved, for instance, with a dynamic-programming algorithm in time $O(n^2 \times p)$.

If we consider the problem as a scheduling problem where we must schedule n tasks onto p processors, we can conclude that the problem becomes difficult (NP-complete) as soon as the tasks are different (the case of identical tasks is case 1) and as soon as we are allowed any mapping (no fixed ordering to enforce, such as in case 2). Moreover, while the problem is in PPT and has an FPTAS with $p = 2$ processors, it is no longer in PPT for an arbitrary number of processors and has only a PTAS.

For a deeper analysis of partitioning problems, the interested reader can refer to the chains-on-chains partitioning case study (Chapter 11).

8.2.2 Assessing problem complexity

In this section, we mention two classical approaches when facing NP-complete problems and aiming at identifying polynomial instances. We illustrate these approaches with two different problems.

The first problem is a routing problem, which is discussed extensively in Chapter 13. Given a directed graph $G = (V, E)$ and a set of terminal pairs $\mathcal{R} = \{R_i = (s_i, t_i)\}$, the goal is to connect as many pairs as possible using edge-disjoint simple paths. In a solution $\mathcal{A}$, each $R_i \in \mathcal{A}$ must be assigned a simple path π_i from s_i to t_i in G so that no two paths π_i and π_j, where $R_i \in \mathcal{A}$, $R_j \in \mathcal{A}$ and $i \neq j$, have an edge in common.

The goal is to maximize $|\mathcal{A}|$, the cardinality of $\mathcal{A}$, i.e., the number of connected terminal pairs. It turns out that this routing problem is NP-complete, and Chapter 13 presents approximation algorithms. But how can we find polynomial instances? A first idea is to bound the number of terminal pairs with a constant, but this does not work, as it turns out that the problem remains NP-complete with only two terminal pairs [35]. Another idea is to restrict the problem to some special classes of graphs. We show in Chapter 13 that the problem is polynomial for linear chains and stars, regardless of the number of terminal pairs.

The second problem is a geometric problem, which is investigated in Chapter 14. How can we partition the unit square into p rectangles of given area $s_1, s_2, \ldots, s_p$ (such that $\sum_{i=1}^{p} s_i = 1$) so as to minimize the sum of the p half perimeters of the rectangles? In Chapter 14, we explain the relevance of this problem to parallel computing, and we show that it is NP-complete. What can we do here? The problem becomes polynomial if we restrict to same-size rectangles [64], but this is very restrictive. Another approach is to change the rules of the game and ask for some specific partitioning of the unit square. Indeed, we show in Chapter 14 that the problem becomes polynomial when restricting to column-based partitioning, i.e., imposing that the

rectangles are arranged along several columns within the unit square. Going further in that direction, we show that the optimal column-based partitioning is indeed a good approximation of the general solution. We hope that this short discussion will urge the reader to read the full case study of Chapter 14.

8.3 Linear programming

Sometimes the solution of an NP-complete problem can be expressed as the solution of an integer linear program. Once we have written an optimization problem as an integer linear program, we can do three things:

1. Solve the integer linear program to obtain optimal solutions for (very) small instances.
2. Relax the integer linear program into a (rational) linear program and solve it to obtain a bound on the optimal solution for the original problem.
3. Relax the integer linear program into a (rational) linear program, solve the latter program to obtain a rational solution, and build an integral solution from the rational one.

We first introduce the necessary notions and definitions (Section 8.3.1). Then we describe several *rounding* approaches to transform a solution of a relaxed linear program into a solution of the original integer linear program (Section 8.3.2).

8.3.1 Formal definition

Linear programming is a mathematical method in which an optimization problem is expressed as the minimization (or maximization) of a linear function whose arguments are constrained by a set of affine equations and inequalities.

DEFINITION 8.6 (Linear program)**.** A linear program is an optimization problem of the form:

$$\text{MINIMIZE} \quad c^T \cdot x \quad \text{SUBJECT TO}$$
$$Ax \leqslant b \quad \text{and} \quad x \geqslant 0$$

where x is an (unknown) vector of variables of size n, A is a (known) matrix of coefficients of size $m \times n$, and b and c are the two (known) vectors of coefficients of respective size m and n (and where c^T is the transpose of vector c).

An *integer linear program* is a linear program whose variables can take only integral values. A *mixed linear program* is a linear program in which some variables must take integral values and some can take rational values.

In the above formal definition, linear programs are given under a canonical form. Therefore, the formal definition of linear programs may look more restrictive than the informal definition we gave right before the formal one. In fact, both definitions are equivalent:

- A maximization problem with the objective function $c^T \cdot x$ is equivalent to a minimization problem with the objective function $-c^T \cdot x$.
- An equality $d^T \cdot x = e$ is equivalent to the set of two inequalities:

$$\begin{cases} d^T \cdot x \leqslant e \\ -d^T \cdot x \leqslant -e. \end{cases}$$

- A variable that can take both positive and negative values can be equivalently replaced by the difference of two nonnegative variables.

An example: Weighted vertex cover

In Section 8.1.2, we have seen the classical version of the vertex cover problem. Given a graph $G = (V, E)$, we want to return a set U of vertices ($U \subset V$) of minimum size that is covering all edges, i.e., such that for each edge $e = (i, j) \in E$, $i \in U$ and/or $j \in U$.

Here, we consider the weighted version of this problem. We assign a weight w_i to each vertex $i \in V$. The problem is then to minimize $\sum_{i \in U} w_i$, where U is once again a vertex cover. This problem amounts to the classical one if $w_i = 1$ for all $i \in V$ and is also NP-complete.

We express this minimization problem as an integer linear program. We introduce a set of Boolean variables, one for each vertex, stating whether the corresponding vertex belongs to the cover. Let x_i be the variable associated with vertex $i \in V$. We will have $x_i = 1$ if i belongs to the cover ($i \in U$) and $x_i = 0$ otherwise.

$$\text{Minimize} \quad \sum_{i \in V} x_i w_i \quad \text{subject to}$$

$$\begin{cases} \forall (i,j) \in E & -x_i - x_j \leqslant -1 \\ \forall i \in V & x_i \leqslant 1 \\ \forall i \in V & x_i \geqslant 0 \end{cases} \tag{8.1}$$

We now show that solving the Integer Linear Program (8.1), with $x_i \in \{0, 1\}$, is absolutely equivalent to solving the minimum weighted vertex cover problem for the graph G.

One can easily check that, if U is an optimal solution to the weighted vertex cover problem, then, by letting $x_i = 1$ for any vertex i in U and $x_j = 0$ for any vertex j not in U, one builds a solution to the above linear program for which the objective function takes the value of the cost of the cover U.

Reciprocally, consider an optimal solution to the Integer Linear Program (8.1), with $x_i \in \{0, 1\}$. From this solution, we build a subset U of V as follows. For any vertex i of V, i belongs to U if and only if $x_i = 1$. For any edge

$e = (i, j) \in E$ we have $-x_i - x_j \leqslant -1$, which is equivalent to $x_i + x_j \geqslant 1$. In other words, either x_i or x_j or both variables are equal to 1 (remember that here the x_is are integer variables). Therefore, at least one of the two vertices i and j is a member of U, and U is thus a cover. The objective function is obviously the cost of the cover U. Therefore, U is a cover of minimum weight.

Complexity

In the general case, the decision problem associated with the problem of solving integer linear programs is an NP-complete problem [58, 38]. However, (rational) linear programs can be solved in polynomial time [93]. Hence, the motivation, when confronted with an NP-complete problem, is to express it as an integer or mixed linear program and then to solve this program as if it were a rational linear program. This method is called *relaxation*. However, the solution obtained this way may be meaningless. For instance, in the case of the linear program for the weighted vertex cover problem (Linear Program (8.1)), one of the variables x_i can have a value different from 0 and 1, which does not make any sense because a vertex cannot be *partially* included in the solution. The problem then becomes how to build an integral solution from a rational one. We now focus on this problem, which is called *rounding*.

8.3.2 Relaxation and rounding

Rounding to the nearest integer

The simplest rounding method is the rounding of any rational variable to the nearest integer. (Obviously, this method is not fully defined because one will still have to decide how to handle variables whose values are of the form $z+0.5$ where z is an integer.) We illustrate this method with the weighted vertex cover problem.

Algorithm **lp-wvc** is defined as follows. First, solve the Linear Program (8.1) over the rationals rather than on the integers, and let $\{x_i^*\}_{i \in V}$ be the found optimal solution. Then, any vertex i of V belongs to the cover U if and only if $x_i^* \geqslant \frac{1}{2}$. In other words, we build from the x_i^*s the Boolean variables x_is, by: $x_i = 1 \Leftrightarrow x_i^* \geqslant \frac{1}{2}$. Not only is Algorithm **lp-wvc** correct, it is even an approximation algorithm, as we now prove.

THEOREM 8.12. **lp-wvc** *is a* 2*-approximation algorithm for weighted vertex cover.*

Proof. First, we check that **lp-wvc** returns a cover. Let $(i, j) \in E$ be an edge. Then, because the x_i^*s are a rational solution to the linear program, we have $x_i^* + x_j^* \geqslant 1$, and at least one of them is greater than or equal to $1/2$. Therefore, in the solution of our problem, we have either $x_i = 1$ or $x_j = 1$ (we also can have $x_i = x_j = 1$). Therefore, the edge (i, j) is covered, $x_i + x_j \geqslant 1$.

To prove that the algorithm is a 2-approximation, we compare the cost of the algorithm $C_{\textbf{lp-wvc}} = \sum_{i \in V} x_i w_i$ with the cost of an optimal solution C_{opt}.

The result comes from two observations: (i) For all i, we have $x_i \leqslant 2x_i^*$ (whether i has been chosen to be part of the cover or not), and (ii) the optimal solution of the linear program over the integers has necessarily a higher cost than the rational solution (the integer solution is a solution to the rational problem). Because $\sum_{i \in V} x_i^* w_i$ is an optimal solution to the rational problem, $C_{opt} \geqslant \sum_{i \in V} x_i^* w_i$. Finally, we have

$$C_{\textbf{lp-wvc}} = \sum_{i \in V} x_i w_i \leqslant \sum_{i \in V} (2x_i^*) w_i \leqslant 2C_{opt},$$

which concludes the proof. □

Threshold rounding

We do not have any a priori guarantee that the rounding to the nearest integer will produce a valid integer solution. We illustrate this potential problem with the set cover problem.

DEFINITION 8.7 (SET-COVER)**.** Let V be a set. Let $\mathcal{S}$ be a collection of k subsets of V: $\mathcal{S} = \{S_1, \ldots, S_k\}$ where, for $1 \leqslant i \leqslant k$, $S_i \subset V$. Let K be an integer, with $K < k$. Is there a subcollection of at most K elements of $\mathcal{S}$ that covers all elements of V?

SET-COVER is an NP-complete problem [58, 38]. It easily can be coded as an integer linear program. Let $\delta_{i,j}$ be a Boolean constant indicating whether the element $v \in V$ belongs to the subset $s \in \mathcal{S}$. As previously, variable x_s indicates whether the set $s \in \mathcal{S}$ belongs to the solution. The following integer linear program then searches for a minimum set cover. The first inequality just states that, whatever the element v of V, at least one of the subsets containing v must be picked in the solution.

$$\textsc{Minimize} \quad \sum_{s \in \mathcal{S}} x_s \quad \textsc{subject to}$$

$$\begin{cases} \forall v \in V & -\displaystyle\sum_{s \in S} \delta_{v,s} x_s \leqslant -1 \\ \forall s \in \mathcal{S} & x_s \leqslant 1 \\ \forall s \in \mathcal{S} & -x_s \leqslant 0 \end{cases} \tag{8.2}$$

Now, consider the following particular instance of minimum cover: $V = \{a, b, c, d\}$ and $\mathcal{S} = \{S_1 = \{a, b, c\}, S_2 = \{a, b, d\}, S_3 = \{a, c, d\}, S_4 = \{b, c, d\}\}$. One can easily see that any two elements of $\mathcal{S}$ define an optimal solution. We

write explicitly the Linear Program (8.2) for that instance:

$$\text{MINIMIZE} \quad x_{S_1} + x_{S_2} + x_{S_3} + x_{S_4} \quad \text{SUBJECT TO}$$

$$\begin{cases} & -x_{S_1} - x_{S_2} - x_{S_3} \leqslant -1 \\ & -x_{S_1} - x_{S_2} - x_{S_4} \leqslant -1 \\ & -x_{S_1} - x_{S_3} - x_{S_4} \leqslant -1 \\ & -x_{S_2} - x_{S_3} - x_{S_4} \leqslant -1 \\ \forall s \in \{S_1, S_2, S_3, S_4\} & x_s \leqslant 1 \\ \forall s \in \{S_1, S_2, S_3, S_4\} & -x_s \leqslant 0. \end{cases} \tag{8.3}$$

By summing the first four inequalities, we obtain $x_{S_1} + x_{S_2} + x_{S_3} + x_{S_4} \geqslant \frac{4}{3}$. Hence, the optimal value of the objective function is not smaller than $\frac{4}{3}$. Then, one can check that $x^*_{S_1} = x^*_{S_2} = x^*_{S_3} = x^*_{S_4} = \frac{1}{3}$ defines an optimal solution of the relaxed (rational) version of the Linear Program (8.3). Rounding this optimal rational solution to the nearest integer would lead to $x_{S_1} = x_{S_2} = x_{S_3} = x_{S_4} = 0$, which, obviously, does not define a cover. To circumvent this problem, rather than to round each variable to the nearest integer, one can use a generalization of this technique: threshold rounding. When variables are 0-1 variables, that is, when variables can take only the values 0 or 1, one first sets a threshold and then rounds to 1 exactly those variables whose values are not smaller than the threshold. This technique leads to an approximation algorithm for the minimum set cover problem.

THEOREM 8.13. *Let $\mathcal{P} = (V, \mathcal{S})$ be an instance of the minimum set cover problem in which each element of V belongs to at most p elements of $\mathcal{S}$. Then, solving the Linear Program* (8.3) *over the rationals and rounding the solution with the threshold $\frac{1}{p}$ builds a cover whose size is at most p times the optimal.*

Proof. Let us consider an optimal solution x^* of the relaxed linear program. Let v be any element of V. By definition of p, v belongs to $q \leqslant p$ elements of $\mathcal{S}$: $S_{\sigma(1)}, \ldots, S_{\sigma(q)}$. The Linear Program (8.2) contains the constraint $-x^*_{S_{\sigma(1)}} - x^*_{S_{\sigma(2)}} - \cdots - x^*_{S_{\sigma(q)}} \leqslant -1$. Therefore, there exists at least one $i \in [1, q]$ such that $x_{S_{\sigma(i)}} \geqslant \frac{1}{q} \geqslant \frac{1}{p}$ and the solution contains at least one element of $\mathcal{S}$ that includes v, namely, $S_{\sigma(i)}$. Thus, the solution is a valid cover. Then, for any element s of $\mathcal{S}$, $x_s \leqslant p \times x^*_s$. Indeed, if $x^*_s \geqslant \frac{1}{p}$, then $x_s = 1$ and $x_s = 0$ otherwise. This completes the proof for the approximation ratio. □

Randomized rounding

In the previous two approaches, the value of a variable in a rational solution was considered to be a deterministic indication of what should be the value of this variable in an integer solution. In the randomized rounding approach, the fractional part of such a value is interpreted as a probability.

Let us consider a nonintegral component x_i^* of an optimal rational solution x^*, and let y_i^* be its fractional part: $x_i^* = \lfloor x_i^* \rfloor + y_i^*$, with $0 < y_i^* < 1$. Then, in randomized rounding, y_i^* is considered to be the probability that, in the integral solution, x_i will be equal to $\lceil x_i^* \rceil$ rather than to $\lfloor x_i^* \rfloor$. In practice, using any uniform random generator over the interval $[0, 1]$, one generates a number $r \in [0, 1]$. If $r \geqslant y_i^*$, then we let $x_i = \lceil x_i^* \rceil$, and $x_i = \lfloor x_i^* \rfloor$ otherwise.

Iterative rounding

In all the previously described rounding approaches, a single relaxed linear program is solved, and then one tries to build an integral solution from the rational solution. A potential problem of these approaches is that the assignment of a particular value to one of the variables may force the value of some other variables in any valid solution. For instance, let us go back to the example showing that rounding to the nearest integer could lead to nonfeasible solutions to the minimum cover problem. There, setting $x_{S_1} = 0$ and $x_{S_2} = 0$ imposes that $x_{S_3} = x_{S_4} = 1$ (because, respectively, a and b must be covered). Rounding to the nearest integer ignores this implication and leads to an infeasible solution. A way to avoid such a problem is to assign values only to a subset of the variables and then solve the relaxed version of the linear program while taking into account the assignments made so far. This way, we obtain a new rational solution where fewer variables have nonintegral values. The process is then iterated until an integral solution is built (or the transformed linear program has no solution). The smaller the number of variables assigned at each iteration, the higher the probability to end up with a valid solution but also the higher the number of iterations, the complexity, and the execution time.

8.4 Randomized algorithms

In this section, we briefly explore how randomized algorithms can help deal with NP-complete problems. We restrict ourselves to a randomized algorithm to solve the NP-complete HC problem (recall that HC stands for Hamiltonian Cycle, see Definition 6.4, p. 130). Given an undirected graph, the algorithm incrementally builds a cycle, taking random decisions on the next vertex to visit to augment the current path. The algorithm will indeed output a Hamiltonian cycle with high probability as soon as the graph contains enough edges. We will quantify this last statement in what follows.

8.4.1 The algorithm

Consider a graph $G = (V, E)$. How can we build a Hamiltonian cycle in G by taking random decisions? The first idea is to grow a path iteratively by picking any neighbor of the current path head that has not been picked so far. Start by picking a vertex, say v_1, at random, and make it the head of the path. Then, pick any neighbor of v_1, say v_2, and make it the new head of the path. Progress likewise at each step; pick any neighbor v_{k+1} of the current path head v_k, and make it the new head of the path. But what if v_{k+1} is equal to some vertex v_i, $1 \leqslant i \leqslant k-1$, that is already present in the path? Then, the algorithm can perform a *rotation*, as illustrated by Figure 8.1.

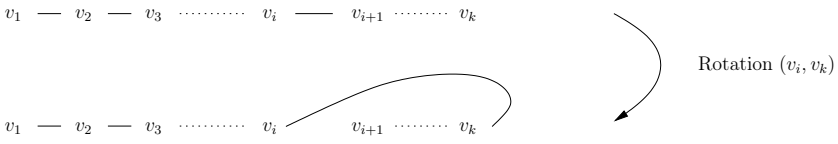


FIGURE 8.1: Rotation (v_k, v_i) of the path. The new head is v_{i+1}.

We obtain the following algorithm, where at each step we pick at random a neighbor u of the current path head v_k among the set of edges originating from v_k that have not been used so far. At the beginning, no edge has been used yet.

8.4.2 Results

What is the probability that Algorithm 8.1 will successfully build a Hamiltonian cycle for G? We would like to express this probability as a function of $n = |V|$, the number of vertices in G. Note that there exist exactly $2^{n(n-1)/2}$ different graphs with n vertices because there are $\binom{n}{2}$ possible edges that can or cannot be added to the graph.

THEOREM 8.14. *There exist constants c and d such that if we pick at random a graph G with n vertices and at least $c \log n$ edges, then with probability at least $1 - \frac{1}{n}$, Algorithm 8.1 will find a Hamiltonian cycle during its first $dn \log n$ steps.*

Proving this theorem is not difficult. This requires, however, some basic knowledge about probability theory (binomial distributions and Markov bound essentially) that is out of the scope of this chapter. We refer the reader to [78] for a proof and many more details about random graphs. We limit ourselves to some comments. First, the randomized algorithm does not give any insight on the P versus NP problem, nor does it help solve all instances of the HC problem. However, on the positive side, we have a fast algorithm that

```
   Input: graph G = (V, E) with n vertices
   Output: a Hamiltonian cycle in G or failure
 1 foreach v ∈ V do
 2     unused(v) := {(v, u) | (v, u) ∈ E}
 3 pick a vertex at random and make it the head of the path
 4 while true do
 5     let (v_1, ..., v_k) be the current path (with head v_k)
 6     if unused(v_k) = ∅ then return failure
 7     else let (v_k, u) be the first element in unused(v_k)
 8     delete edge (v_k, u) from unused(v_k) and unused(u)
 9     if u ∉ {v_1, ..., v_{k-1}} then
10         add u to the path and let v_{k+1} = u be the new path head
11     else
12         let i be such that v_i = u
13         if k = n and v_i = v_1 then return {v_1, ..., v_n}
14         else  rotate (v_k, v_i) and let v_{i+1} be the new path head
```

ALGORITHM 8.1: Randomized algorithm for the HC problem.

solves HC in most instances, as soon as the graph has enough edges. This is expected news, as we expect a random graph to be connected and then to have large cliques, or a Hamiltonian cycle, when its number of edges grow. But the beauty of Theorem 8.14 is to quantify this observation.

8.5 Branch-and-bound and backtracking

In this last section, we introduce branch-and-bound and backtracking techniques. The principle is to represent as a tree the search space (i.e., all candidate solutions) and then to explore this tree and remove branches that either lead to no valid solution or lead to solutions that are less good. Such algorithms return exact solutions to an NP-complete problem. For decision problems, the technique is called backtracking, while it is called branch-and-bound for optimization problems. While there is no guarantee on the execution time of such algorithms (the worst case may well be exponential because we may need to explore the entire search space), they are offering practical and often efficient solutions to deal with NP-complete problems.

We first present a small example of a backtracking algorithm with the n-queens problem. Then, we investigate branch-and-bound with the knapsack problem. Finally, we discuss some more complex graph algorithms.

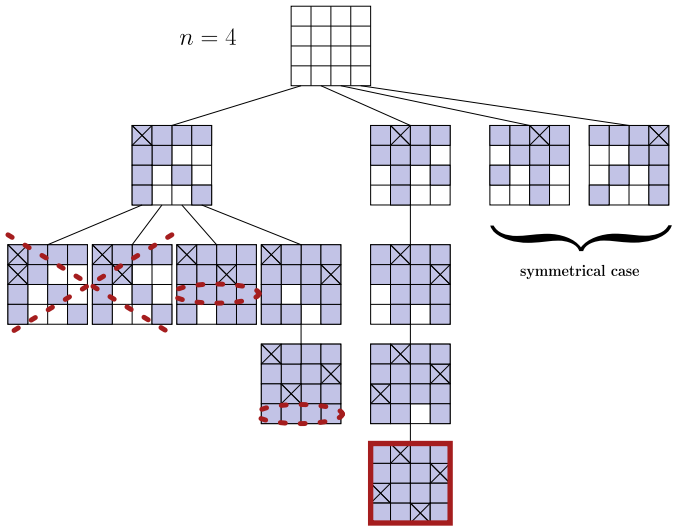


FIGURE 8.2: The n-queens backtracking tree.

8.5.1 Backtracking: The n queens

In a chess game, a queen can move as far as she wants: horizontally, vertically, or diagonally. We consider a chess board with n rows and n columns. The problem is to place n queens on this chess board so that none of them can attack any other in one move.

In any solution, there is exactly one queen per row. Therefore, the search space is of size n^n. However, because of the many constraints, many solutions can be discarded. The idea of the backtracking algorithm is to place a queen on the first row (n possible choices) and then perform a recursive call for the next row. We discard the choices that lead to no solution, and if no solution is found on a branch of the tree, we go up in the tree and try the next possibility (the next branch).

Figure 8.2 illustrates the tree for $n = 4$. Because the problem is symmetrical, we develop only the portion of the tree in which we place the first queen either on the first or on the second column. Once a queen has been placed, the squares on which it is not possible to place another queen have been colored. Therefore, if we place the first queen on the top left corner, the queen on the second row can be placed only on the third or fourth column. If we place it on the third column, there is no further choice for the third queen. If we place it on the fourth column, we can still place the third queen on the second column, but then there is no possibility for the last queen. However, a solution is found by exploring the second branch of the tree.

8.5.2 Branch-and-bound: The knapsack

A branch-and-bound algorithm works in two phases. The *branch* consists of splitting a set of solutions into subsets, while the *bound* consists of evaluating the solutions of a subset by bounding the value of the best solution in this subset.

We consider the knapsack problem, which was introduced in Section 4.2 and that we redefine briefly. Given a set of items $I_1, \ldots, I_n$, where item I_i has a weight w_i and a value c_i ($1 \leqslant i \leqslant n$), we want to determine the items to include in the collection so that the total weight is less than a given limit W and the total value is as large as possible. We consider the variant of the problem where we have as many units of each item as we want. Let x_i be the number of units of item I_i that we decide to add into the knapsack. The goal is to maximize $\sum_{i=1}^{n} x_i \times c_i$, under the constraint $\sum_{i=1}^{n} x_i \times w_i \leqslant W$.

We consider the running example from [15]. There are four items, and the goal is to find $\max(4x_1 + 5x_2 + 6x_3 + 2x_4)$, under the constraint $33x_1 + 49x_2 + 60x_3 + 32x_4 \leqslant 130$.

The search space is represented as a tree. The leaves of the tree correspond to maximal solutions, i.e., solutions to which we cannot add any item because of the constraint on total weight. At the root of the tree, we have not chosen any item. The root has $\left\lceil \frac{W}{w_1} \right\rceil + 1$ children, which corresponds to picking, respectively, $0, 1, \ldots, \left\lceil \frac{W}{w_1} \right\rceil$ units of I_1. Then, for each of these nodes, we add one child for each possible number of units of the next item that can be chosen. For the last item, we fill the knapsack by adding systematically as many units of this item as we can. A part of the tree corresponding to this example is depicted in Figure 8.3. Its height is equal to the number of different items, n. Each leaf corresponds to a solution, and the number of leaves is exponential in the problem size.

Note that we have ordered the items such that the c_i/w_i are nonincreasing, i.e., the first item has the best value/weight ratio.

Given a search space represented by a tree, the branch-and-bound algorithm works as follows. At the beginning, there is only one active node, the root of the tree. At each step, we choose an active node, and we process its children nodes. If a child has only one child itself, we traverse the branch until we eventually find a leaf or a node with at least two children. Then we evaluate the node as follows: (i) If the node is a leaf, it corresponds to a solution, and we can compute the exact value of this solution. We keep the best solution between case (i) and the previously best known solution; (ii) otherwise, we provide an upper bound on the solutions in the branch by filling the unused weight with the item that has not yet been considered and that has the best value/weight ratio as if it were a liquid, that is, as if we were allowed to use a noninteger number of items. All the nodes from case (ii) become active. Before moving to the next step (i.e., picking up a new active node), we remove the active nodes that will never lead to a better solution than one of the solutions

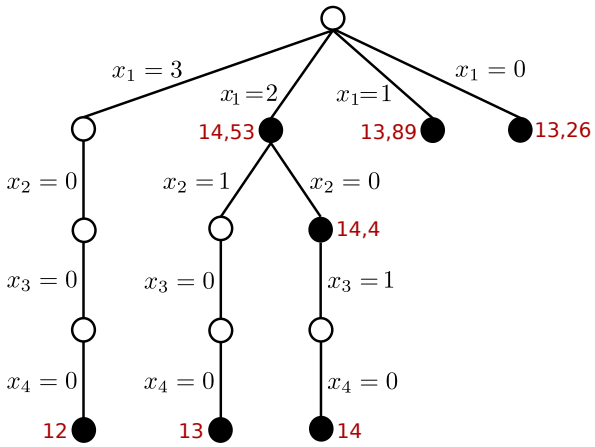


FIGURE 8.3: Branch-and-bound algorithm for the knapsack problem.

already found, i.e., if their upper bound is smaller than the value of the best solution. This corresponds to the pruning of the search space.

In the example (see Figure 8.3), we first process the child node corresponding to $x_1 = 3$. We cannot add any other item in the knapsack, so we reach a leaf of the tree. The value of the solution is $3 \times 4 = 12$. This is the best current solution. Then, we consider the second child node of the root, corresponding to $x_1 = 2$. It has two children, corresponding to $x_2 = 1$ and $x_2 = 0$ (we cannot add more than one unit of item I_2 in the knapsack). Therefore, we evaluate this node. The upper bound is computed with $x_1 = 2$, and all the remaining space $(130 - 66)$ is filled with item I_2, which is the remaining item with the best value/weight ratio. We obtain $2 \times 4 + 5/49 \times (130 - 66) = 14.53$. Because $14.53 \geqslant 12 + 1$, it may be possible to find a better solution than the current one (whose value is 12) in this tree, i.e., a solution whose value is at least 13 (solutions are integers). Therefore, this node becomes active. With $x_2 = 1$, we obtain a solution of value 13. Then, we evaluate the node for $x_2 = 0$ because there may still be a solution of value 14 in this subtree. The evaluation is done by filling the remaining space with item I_3, leading to $2 \times 4 + 6/60 \times (130 - 66) = 14.4$. This branch leads to a solution 14, with $x_3 = 1$. Because the upper bound for this subtree is 14.4, we cannot find a better solution. We evaluate the third child of the root to 13.89 and then the last child to 13.26; therefore, no better solution can be found. There are no more active nodes. The nodes of the tree colored in black are the nodes that have been evaluated.

Note that several strategies can be considered for the choice of the next active node. A depth-first search, as we have done in the example, is very practical because there are few nodes that are simultaneously active. A breadth-first

search often leads to poor results. Another strategy consists of picking the active node with the best evaluation. Some hybrid strategies also can be considered. For instance, one can perform a depth-first search until a solution is found and then use a best evaluation strategy to find even better solutions. Such a strategy may allow the pruning of several branches.

Note that other strategies can be used to solve this kind of problem. The branch-and-bound algorithm is often not very efficient in the worst case. However, it often leads to efficient algorithms on average, as we detail in the next section.

8.5.3 Graph algorithms

In this section, we consider two important NP-complete graph problems that we aim to solve with backtracking algorithms. First, we investigate the problem of finding the largest independent set, and then we investigate the graph coloring problem.

8.5.3.1 Independent sets

Let $G = (V, E)$ be a graph with n vertices, numbered from 1 to n. The problem is to find the size of the largest independent set of G, i.e., a subset $S \subseteq V$ such that, for all $i, i' \in S$, $(i, i') \notin E$, and $|S|$ is maximum.

The backtracking algorithm is easy to describe and analyze for this problem. The idea is to explore all possible independent sets and to build a tree with all the solutions to the problem. The root of the tree corresponds to the empty set. The children of the root node correspond to independent sets of size 1, and we add a node only if it is an independent set. The tree is built in a depth-first traversal. First, we search for independent sets containing vertex 1, which correspond to the first child of the root, denoted $\{1\}$ (if $(1, 1) \notin E$). We then try to increase the size of this set by adding vertex 2. The children of $\{1\}$ are the independent sets of size 2 containing vertex 1. If $(1, 2) \in E$, then there is no independent set containing both 1 and 2; therefore, we do not add any node in the solution tree and proceed with vertices $3, \ldots, n$. Otherwise, we add $\{1, 2\}$ as a child node of $\{1\}$ and move to the next level of the tree, trying to add vertices $3, \ldots, n$ to this independent set and building independent sets of size 3. When no vertex can be further added, we *backtrack* up in the tree and develop all remaining branches of the solution tree. The height of the solution tree gives the maximum size of an independent set.

The solution tree has one node per independent set and, therefore, the complexity of the algorithm depends on the number of independent sets, which can be exponential: For a graph with $E = \emptyset$, this number is 2^n. However, for a clique of size n, there are only $n + 1$ independent sets. The analysis aims at determining the average complexity of the algorithm, i.e., the average number of independent sets, denoted I_n.

Let $I(G)$ be the number of independent sets of a graph $G = (V_G, E)$.

$H(G,S)$ equals 1 if S is an independent set of G, and 0 otherwise. Therefore, $I(G) = \sum_{S \subseteq V_G} H(G,S)$, and the sum contains the 2^n possible subsets of V.

The average number of independent sets I_n is then the sum over all possible graphs G with n vertices, divided by the number of such graphs, $2^{n(n-1)/2}$. We obtain

$$I_n = 2^{-n(n-1)/2} \sum_{|V_G|=n} \sum_{S \subseteq V_G} H(G,S) \; .$$

We can invert the two sums, and we examine $\sum_{|V_G|=n} H(G,S)$. Given a set S, this value corresponds to the number of graphs with n nodes that contain S as an independent set. If $|S| = k$, there are $k(k-1)/2$ edges that cannot exist in G, and there are $n(n-1)/2 - k(k-1)/2$ possible edges, which leads to $2^{n(n-1)/2-k(k-1)/2}$ graphs with n vertices such that $H(G,S) = 1$. Finally, since the number of sets S with k vertices is $\binom{n}{k}$, we obtain

$$I_n = \sum_{k=0}^{n} \binom{n}{k} 2^{-k(k-1)/2} \; .$$

On average, the algorithm is much better than in the worst case; for instance, with $n = 40$, $I_n = 3862.9$, while $2^n > 10^{12}$. In fact, for large values of n, $I_n = O(n^{\log(n)})$ and, therefore, the average complexity of the algorithm remains subexponential.

8.5.3.2 Graph coloring

For the graph coloring problem, the backtracking algorithm leads to more efficient results on average than for the independent sets problem because it turns out that the average complexity is, in fact, constant for a fixed number of colors, even when the number of vertices tends to infinity.

Let $G = (V, E)$ be a graph with n vertices, numbered from 1 to n, and K be an integer. The K-coloring problem is to associate a color with each vertex such that two vertices connected by an edge have a different color, where K is the number of colors.

The backtracking algorithm builds all partial colorings of the graph with only a subset of vertices $\{1, \ldots, L\}$, with $1 \leqslant L \leqslant n$. The root of the tree corresponds to the coloring of the empty graph; it is represented by an empty set. It has K children nodes, corresponding to the possible colors for vertex 1. The node is labeled by the set of colors for the vertices that we consider, i.e., the children of the root are labeled $1, \ldots, K$. Similar to the backtracking algorithm for the independent sets problem, we build the tree in a depth-first traversal. We add a node 11 as a child of 1 if and only if $(1,2) \notin E$, then we assign the lowest possible color to the third vertex, and so on. If there is no possible color for one of the vertices, or if we have successfully colored all vertices, we go up in the tree until we can try another color for one of

the vertices. (Remember that the backtracking algorithm builds *all* partial colorings of the graph.)

Note that the branch of a tree may stop before a color has been assigned to each vertex, and it may happen that no valid coloring can be found. At level L of the tree, we have all partial colorings of vertices $\{1,\ldots,L\}$, and a valid coloring has been found if the tree has nodes of level n. Graph G restricted to vertices $\{1,\ldots,L\}$ is denoted $H_L(G)$ in the following.

The goal is to determine the average number of nodes $A_{n,K}$ of a backtrack tree generated when coloring a graph of size n with at most K colors. There are $2^{n(n-1)/2}$ different graphs, and we decompose the backtrack trees into levels. If G is a graph with n vertices, we denote by $P(K, H_L(G))$ the number of nodes at level L of the backtrack tree of G. It is equal to the number of correct colorings of graph $H_L(G)$ with K colors. Finally,

$$A_{n,K} = 2^{-n(n-1)/2} \sum_{|V_G|=n} \sum_{L=0}^{n} P(K, H_L(G)) \; .$$

We invert the two sums and examine $\sum_{|V_G|=n} P(K, H_L(G))$, given a level L. Note that there are exactly $2^{n(n-1)/2-L(L-1)/2}$ graphs that share the same graph $H_L(G)$, and, therefore,

$$A_{n,K} = 2^{-n(n-1)/2} \sum_{L=0}^{n} 2^{n(n-1)/2-L(L-1)/2} B_{L,K} = \sum_{L=0}^{n} 2^{-L(L-1)/2} B_{L,K} \; ,$$

where $B_{L,K}$ is the total number of correct colorings with K colors of all graphs with L vertices. Given a coloring, we denote by s_i the number of vertices that are colored with the color i, for $1 \leqslant i \leqslant K$. Because the graphs have L vertices, we have $\sum_{i=1}^{K} s_i = L$. Moreover, an edge can connect only two vertices of different colors, and, thus, the maximum number of edges is $E_{n,K} = s_1 s_2 + s_1 s_3 + \cdots + s_1 s_K + s_2 s_3 + \cdots + s_{K-1} s_K = \sum_{1 \leqslant i < j \leqslant K} s_i s_j$. We compute this value as follows:

$$\begin{aligned} E_{n,K} &= \tfrac{1}{2} \textstyle\sum_{i \neq j} s_i s_j = \tfrac{1}{2} \left(\sum_{i,j=1}^{K} s_i s_j - \sum_{i=1}^{K} s_i^2 \right) \\ &= \tfrac{1}{2} \left(\sum_{i=1}^{K} s_i \right)^2 - \tfrac{1}{2} \sum_{i=1}^{K} s_i^2 = \tfrac{1}{2} L^2 - \tfrac{1}{2} \sum_{i=1}^{K} s_i^2 \; . \end{aligned}$$

It is easy to check that $\sum_{i=1}^{K} s_i^2 \geqslant L^2/K$, because $L = \sum_{i=1}^{K} s_i$:

$$\begin{aligned} \textstyle\sum_{i=1}^{K} s_i^2 - L^2/K &= \textstyle\sum_{i=1}^{K} s_i^2 - 2L^2/K + L^2/K \\ &= \textstyle\sum_{i=1}^{K} \left(s_i^2 - 2L s_i/K + L^2/K^2 \right) = \sum_{i=1}^{K} \left(s_i - L/K \right)^2 \geqslant 0 \; . \end{aligned}$$

Therefore, $E_{n,K} \leqslant \frac{1}{2} L^2 - \frac{1}{2} L^2/K = L^2(1 - 1/K)/2$. The number of graphs $H_L(G)$ with the same coloring is at most $2^{L^2(1-1/K)/2}$. Because there are at most K^L different colorings (counting invalid ones), we obtain $B_{L,K} \leqslant K^L 2^{L^2(1-1/K)/2}$, and, finally,

$$A_{n,K} \leqslant \sum_{L=0}^{n} 2^{-L(L-1)/2} K^L 2^{L^2(1-1/K)/2} \leqslant \sum_{L=0}^{\infty} K^L 2^{L/2} 2^{-L^2/2K} .$$

This infinite series is converging; therefore, $A(n, K)$ is bounded for all n.

8.6 Bibliographical notes

The FPTAS for scheduling independent tasks on two processors (Section 8.1.5) is presented in [95]. Further references for approximation algorithms are the books by Ausiello et al. [5] and by Vazirani [103]. Randomized algorithms (Section 8.4) are dealt with in the books by Mitzenmacher and Upfal [78] and by Motwani and Raghavan [80]. Section 8.5.2 (branch-and-bound) is inspired from [15]. The backtracking graph algorithms (Section 8.5.3) are analyzed in [108].