Algorithmique I
Année 2008-2009

A. Benoit, B. Depardon
C. Mouilleron, C. Rezvoy

Partiel d'algorithmique

29 Octobre 2008

2 Heures

1 Un jeu de patience

Question 1.1

On considère un jeu de n cartes, dans lequel à chaque carte est associée une valeur entière. Le jeu est mélangé et initialement posé à l'envers sur la table (on ne connaît alors la valeur d'une carte que lorsqu'on la retourne). On souhaite trier les cartes en utilisant des piles de cartes posées sur la table. La construction des piles doit respecter les règles suivantes.

- Initialement il n'y a aucune pile. La première carte tirée forme une nouvelle pile (on voit la valeur de la carte lorsqu'on la tire) : on pose la carte sur la table avec la face indiquant la valeur vers le haut (la valeur est donc visible).
- Chaque nouvelle carte peut être posée soit sur une pile existante dont la carte du dessus de pile a une valeur supérieure à la valeur de la carte tirée, soit sur une nouvelle pile à droite des autres.
- Lorsqu'il n'y a plus de cartes à tirer on s'arrête.

Proposez un algorithme qui trie le jeu de cartes. On commencera par créer des piles (le plus petit nombre possible) en se basant sur les règles précédentes. Une fois les piles de cartes créées, on ne peut voir que la valeur des cartes du dessus, puis on dépile pour effectuer le tri. On ne peut prendre à chaque étape qu'une carte se situant sur le dessus d'une pile, et on veut avoir à la fin en main les cartes triées par ordre croissant. Quelle est la complexité de cet algorithme ? Précisez les structures de données utilisées. Donnez un exemple où votre algorithme se comporte mal (*i.e.,* il atteint sa complexité dans le pire cas).

Solution :

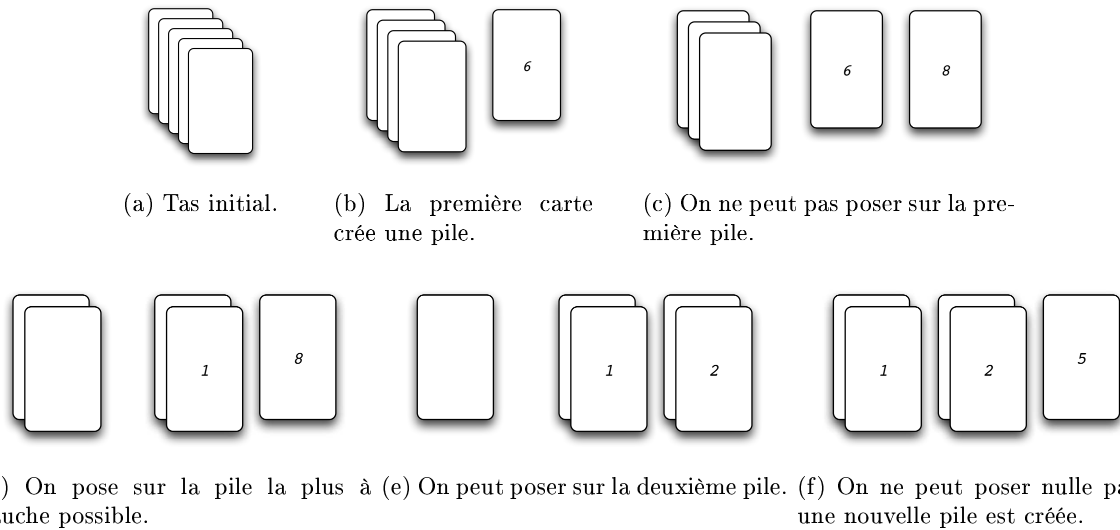
Stratégie gloutonne pour l'algorithme (la Figure 1 illustre l'utilisation de ces règles) :

- *Initialement il n'y a aucune pile. La première carte tirée forme une nouvelle pile (on voit la valeur de la carte lorsqu'on la tire) : on pose la carte sur la table avec la face indiquant la valeur vers le haut (la valeur est donc visible).*
- *Chaque nouvelle carte ne peut être posée sur une pile existante que si la carte du dessus de pile a une valeur supérieure à la valeur de la carte tirée. Si aucune pile ne respecte cette règle, on crée alors une nouvelle pile à droite de toutes les autres.*
- ***Lorsqu'on a le choix entre plusieurs piles sur lesquelles poser la carte, alors on la pose sur la pile la plus à gauche possible.***
- *Lorsqu'il n'y a plus de cartes à tirer on s'arrête.*

C'est en fait un jeu de solitaire consistant à trouver une stratégie pour obtenir le plus petit nombre de piles possible. La méthode gloutonne présentée ci-dessus est optimale.

Lemme 1. *(Tiré de [1])*

Soit π une permutation de $\{1, 2, \ldots, n\}$. Patience sorting joué avec la stratégie gloutonne présenté précédemment se termine avec exactement $l(\pi)$ tas, où $l(\pi)$ est la longueur de la plus grande sous séquence croissante. De plus, si le jeu est joué avec n'importe quelle autre stratégie



(a) Tas initial. (b) La première carte crée une pile. (c) On ne peut pas poser sur la première pile.

(d) On pose sur la pile la plus à gauche possible. (e) On peut poser sur la deuxième pile. (f) On ne peut poser nulle part, une nouvelle pile est créée.

Fig. 1: Exemple de la création des piles de cartes

pour choisir la pile, il se terminera alors avec au moins $l(\pi)$ piles. La stratégie gloutonne est donc optimale.

Preuve. Si les cartes $a_1 < a_2 < \cdots < a_l$ apparaissent par ordre croissant, alors quelque soit la stratégie adoptée il faudra poser les cartes à chaque fois sur une nouvelle pile à droite de la pile contenant a_{i-1}, puisque la carte sous a_i doit nécessairement être plus petite. Donc le nombre final de pile est d'au moins l, et donc au moins $l(\pi)$. Inversement, si l'on utilise la méthode gloutonne, quand une carte c est placée dans une pile autre que la première pile, alors on place un pointeur allant de la carte c vers la carte c' étant sur le dessus de la pile à gauche de la pile de c ($c' < c$). À la fin du jeu, soit a_l la carte sur la pile la plus à droite (pile l). La séquence $a_1 \leftarrow a_2 \leftarrow \cdots \leftarrow a_l$ obtenue à partir des pointeurs est une plus grande sous séquence croissante de la taille le nombre de piles.

Pour construire les piles on peut le faire en $O(n \log n)$ en faisant une dichotomie sur la valeur des cartes sur le dessus pour trouver l'endroit où poser la carte. Cependant le dépilement aboutit à un temps en $O(n^2)$.

Un exemple du pire cas : $k+1, \ldots, 2k, 1, \ldots, k$.

`http://en.wikipedia.org/wiki/Patience_sorting`

□

Question 1.2

Un arbre T de van Emde Boas [3] est une structure permettant de maintenir une liste triée d'entiers compris dans l'intervalle $[1, n]$ avec un temps $O(\log \log n)$ pour l'insertion et la suppression. On définit les opérations suivantes :

- *insérer*(i) : insérer i dans T, en $O(\log \log n)$
- *supprimer*(i) : supprimer i de T, en $O(\log \log n)$
- Si i est déjà inséré dans T, *suivant*(i)/*précédent*(i) : obtenir le suivant/précédent de i dans T (et retourne NIL si le suivant/précédent n'existe pas), en $O(1)$. *suivant*(i) correspond à la valeur supérieure à i, la plus proche de i, présente dans T. *precedent*(i) correspond à la valeur inférieure à i, la plus proche de i, présente dans T.

On dispose maintenant d'un jeu de cartes ayant des valeurs de 1 à n. En vous basant sur un arbre de van Emde Boas, proposez un algorithme en $O(n \log \log n)$ permettant de construire le plus petit nombre de piles de cartes respectant les règles de construction (on ne demande pas ici de trier les cartes, juste de construire les piles). La structure utilisée pour représenter les piles est laissée au choix.

Solution :

Tiré de l'article [2].

On utilise les variables suivantes :

- S *: une structure de van Emde Boas*
- P *: la liste des cartes sur le dessus des piles,* $P[k]$ *est la carte sur le dessus de la pile numéro* k
- *carte : tableau de taille* n*,* $carte[i]$ *est la valeur de la carte qui se trouve sous la carte de valeur* i *dans la pile où se trouve la carte de valeur* i *(0 si la carte est en dessous de la pile, où si elle n'a pas encore été placée dans une pile).*

Algorithme 1 Patience heaps

1: $S \leftarrow \emptyset$
2: **pour** $i = 1$ à n **faire** /* Initialisation : les piles sont vides et aucune carte n'a de carte sous elle. */
3: $P[i] = carte[i] = 0$
4: **pour** $i = 1$ à n **faire** /* On dépile le tas initial */
5: $k \leftarrow valeur_carte(i)$
6: $S.insérer(k)$
7: $j \leftarrow S.suivant(k)$
8: **si** $j \neq NIL$ **alors**
9: $carte[k] \leftarrow j$
10: $S.supprimer(j)$
11: $k \leftarrow i \leftarrow 1$
12: **tantque** $k \neq NIL$ **faire** /* Création des piles */
13: $P[i] \leftarrow k$
14: $k \leftarrow S.suivant(k)$
15: $i \leftarrow i + 1$

Explications de l'algo. *Initialement la structure de van Emde Boas est vide, aucune carte n'a de carte sous elle, et les piles sont vides (lignes 1 à 3). L'idée est de ne conserver dans la structure de van Emde Boas que les cartes situées sur le dessus des piles, les informations pour retrouver le reste des cartes dans une pile sont gardées dans le tableau carte (lignes 4 à 10). On insère ainsi chaque carte dans* S*, mais on enlève la carte suivante si elle existe (c'est la carte sur laquelle on a posé notre carte). Finalement il ne nous reste plus qu'à mettre à jour le tableau* P *pour qu'il contienne la liste des cartes sur le dessus des piles (lignes 11 à 15).*

□

Question 1.3

En utilisant votre algorithme précédent pour créer les piles de cartes avec des valeurs dans $[1, n]$, proposez un algorithme pour trier des cartes ayant des valeurs dans $[1, n]$ en $O(n \log \log n)$.

Solution :

Algorithme 2 Patience sorting

1: Créer les piles avec l'algorithme 1
2: $T \leftarrow []$ /* T : tableau qui va contenir les valeurs triées */
3: $pos \leftarrow 1$
4: $k \leftarrow P[1]$
5: **tantque** $k \neq NIL$ **faire**
6: $\quad T[pos] \leftarrow k$
7: $\quad v \leftarrow k$
8: $\quad k \leftarrow carte[v]$
9: $\quad$ **si** $k \neq 0$ **alors**
10: $\quad\quad S.insérer(k)$
11: $\quad k \leftarrow S.suivant(v)$
12: $\quad pos \leftarrow pos + 1$

□

2 Multiplication de deux entiers

Soient deux entiers x et y codés sur n bits (on supposera que n est une puissance de 2). On souhaite multiplier ces deux entiers entre eux, en travaillant au niveau des bits. La multiplication de deux entiers de n bits se fait de manière triviale en $O(n^2)$, la multiplication d'un entier par une puissance de 2, et l'addition de 2 entiers se font en temps linéaire $O(n)$.

Question 2.1

La méthode diviser pour régner n'est pas toujours meilleure qu'un algorithme naïf. Pour illustrer cela, donnez un algorithme diviser pour régner ayant une complexité en $O(n^2)$ pour multiplier x et y, 2 entiers de n bits.

Solution :

On peut écrire x et y de la façon suivante :

$$\begin{aligned} x &= \boxed{x_L}\boxed{x_R} = 2^{n/2}x_L + x_R \\ y &= \boxed{y_L}\boxed{y_R} = 2^{n/2}y_L + y_R \end{aligned}$$

On peut alors multiplier x et y de la façon suivante :

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

On a donc besoin de 4 appels récursifs pour multiplier des entiers de taille $n/2$ pour $x_L y_L$, $x_L y_R$, $x_R y_L$ et $x_R y_R$. Il faut ensuite faire les multiplications par $2^{n/2}$ et les additions, cela peut être fait en $O(n)$. On a donc la formule de récurrence suivante :

$$T(n) = 4T(n/2) + O(n)$$

ce qui nous donne une complexité totale en $O(n^2)$ par le master theorem.

□

Question 2.2

Proposez un algorithme diviser pour régner ayant une complexité inférieure à $O(n^2)$. Donnez la formule de récurrence pour votre algorithme et sa complexité finale (en O). On supposera pour simplifier que l'addition/multiplication de deux nombres de taille n est un nombre de taille n.

Solution :

En réorganisant un peu l'expression précédente, on peut améliorer l'algorithme pour avoir une complexité en $O(n^{\log_2 3})$. Seulement trois multiplications sont suffisantes : $x_L y_L$, $x_R y_R$ et $(x_L + x_R)(y_L + y_R)$, puisque $(x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R = x_L y_R + x_R y_L$.

On a donc l'algorithme suivant :

Algorithme 3 multiplier(x,y)

Entrées: 2 entiers x et y de n bits
Sorties: xy
1: **si** $n = 1$ **alors**
2:    **retourner** xy
3: $p_1 = multiplier(x_L, y_L)$
4: $p_2 = multiplier(x_R, y_R)$
5: $p_3 = multiplier(x_L + x_R, y_L + y_R)$
6: **retourner** $p_1 \times 2^n + (p_3 - p_1 - p_2) \times 2^{n/2} + p_2$

On a donc la formule de récurrence suivante :

$$T(n) = 3T(n/2) + O(n)$$

ce qui nous donne $O(n^{\log_2 3}) \approx O(n^{1.59})$.

□

3 Plus grande sous séquence croissante

On considère une suite de n entiers : $a_1, a_2, \dots, a_n$. Une sous séquence est un sous ensemble d'entiers de la suite, pris dans le même ordre que la suite : $a_{i_1}, a_{i_2}, \dots, a_{i_k}$, avec $1 \leq i_1 \leq i_2 \leq \dots \leq i_k \leq n$. Une sous séquence croissante est une sous séquence où les entiers sont strictement croissants. Par exemple, la plus grande sous séquence croissante de la suite $5, 2, 8, 6, 3, 6, 9, 7$ est $2, 3, 6, 9$.

Question 3.1

Donnez une plus longue sous séquence croissante de $11, 6, 2, 24, 25, 12, 21, 41, 34, 30$.

Solution :

Par exemple $11, 24, 25, 41$. *Longueur max = 4 ;*

□

Question 3.2

Donnez un algorithme de programmation dynamique permettant de trouver la plus grande sous séquence croissante d'une suite de n entiers et donnez sa complexité.

Solution :

Algorithme. *Pour que deux éléments i et j de la suite soient dans la solution, il faut que $i < j$ (on respecte l'ordre de la suite) et $a_i < a_j$ (on respecte le fait que la séquence soit strictement croissante).*

L'idée est de construire un graphe contenant les transitions possibles, i.e., *une arête ne peut exister entre deux nœuds i et j que si $i < j$ et $a_i < a_j$. Notre but est alors de trouver le chemin le plus long dans ce graphe.*

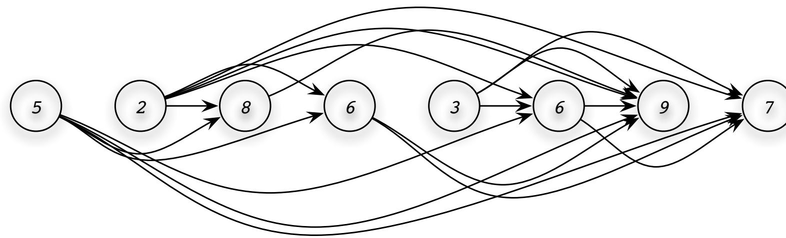


FIG. 2: DAG pour le problème de la plus longue sous séquence croissante.

Algorithme 4 Plus grande sous séquence croissante

1: $Long[1 \dots n] \leftarrow [1 \dots 1]$
2: $Pred[1 \dots n] \leftarrow [1 \dots n]$
3: **pour** $i = 1$ à n **faire**
4:   **pour** $j = 1$ à $i - 1$ **faire**
5:     **si** $a_i > a_j$ **alors**
6:       $Long[j] \leftarrow \max\{Long[j], 1 + Long[i]\}$
7:       **si** $Long[j] = 1 + Long[i]$ **alors**
8:         $Pred[j] = i$
9: **retourner** $\max_j\{Long[j]\}$

$Long[j]$ est la longueur du plus long chemin se terminant sur le nœud j. $Pred[j]$ contient le nœud précédant j dans le plus long chemin, ce tableau permet de reconstruire la solution.

Complexité. *On a une complexité en $O(n^2)$.*

□

Question 3.3

On peut cependant faire mieux que par programmation dynamique. Proposez un algorithme qui tourne en $O(n \log n)$.

Solution :

Algorithme. *On utilise les variables suivantes :*
- $Pred[i]$ *est le prédécesseur de* i *dans la plus longue sous séquence terminant à* i*.*
- L *est la longueur du plus long chemin.*
- $M[l]$ *est la position* j *du plus petit élément* a_j *tel que* $j \leq i$*, et il y a une séquence croissante de taille* l *terminant en* a_j*. La séquence* $a_{M[1]}, a_{M[2]}, \ldots, a_{M[L]}$ *est croissante.*

L'idée est de traiter les éléments dans l'ordre en gardant à chaque étape la plus grande sous séquence trouvée, puis de rechercher par dichotomie la plus grande sous séquence telle qu'on puisse insérer l'élément en cours.

Algorithme 5 Plus grande sous séquence croissante en $O(n \log n)$

1: $L \leftarrow 0$
2: $M[0] \leftarrow 0$
3: **pour** $i = 1$ à n **faire**
4:    Faire une dichotomie pour trouver le plus grand $l \leq L$, tel que $a_{M[l]} < a_i$ (ou alors $l \leftarrow 0$ si ce l n'existe pas).
5:    $Pred[i] \leftarrow M[l]$
6:    **si** $j == L$ ou $a_i < a_{M[l+1]}$ **alors**
7:       $M[l+1] \leftarrow i$
8:       $L \leftarrow \max\{L, l+1\}$
9: **retourner** L

Complexité. *Il faut* $O(\log n)$ *pour chaque dichotomie, et on boucle sur tous les éléments, soit une complexité en* $O(n \log n)$*.*

□

4 Alignement de séquences

Un problème récurrent en bioinformatique est l'alignement de séquences. Si l'on a deux mots u et v formés à partir d'un alphabet $\mathcal{A}$ (par exemple les nucléotides : $\mathcal{A} = \{G, T, A, C\}$), on appelle alignement de u et v un couple de mots u', v' sur l'alphabet $\mathcal{A} \cup \{e\}$ (où $e \notin \mathcal{A}$ est le caractère d'espacement). u' et v' satisfont les propriétés suivantes :
- u' et v' ont la même longueur ($|u'| = |v'| = n$)
- si on supprime tous les e dans u' et v' on obtient respectivement u et v
- les lettres e ne sont pas à la même position dans u' et v'

Par exemple un alignement de $u = CGATTAG$ et $v = GATCGA$ est

$$\begin{aligned} u' &= CGATTeAG \\ v' &= eGATCGAe \end{aligned}$$

Afin de déterminer le meilleur alignement, il nous faut une métrique. Soit p une fonction de similarité entre un couple de lettres. La valeur de $p(a, b)$ est un réel d'autant plus grand que les lettres sont considérées comme similaires d'un point de vue biologique. On prendra donc une valeur négative lorsque les lettres sont différentes. On a donc $b \neq a, p(a, a) > 0 > p(a, b)$, la fonction de similarité est également symétrique : $p(a, b) = p(b, a)$. De plus, on se donne un réel $q < 0$ qui exprime la similarité d'une lettre avec un espacement, elle est indépendante de

la lettre, on a donc $\forall a \in \mathcal{A}, p(a,e) = p(e,a) = q$. Le score d'un alignement est simplement la somme des similarités entre les lettres de u' et v' :

$$\sum_{i=1}^{n} p(u'_i, v'_i)$$

Question 4.1

On pose $p(a,b) = -1$ pour $a \neq b$, $p(a,a) = 5$ et $q = -2$. Quel est le score de l'alignement donné en exemple ? Donnez le meilleur alignement et le score correspondant pour les chaînes suivantes : $GATTACA$ et $ATACGTA$.

Solution :

Le score pour $u = CGATTAG$ et $v = GATCGA$ est 13.

Le meilleur alignement pour $GATTACA$ et $ATACGTA$ est :

$$\begin{aligned} u' &= eATeACGTA \\ v' &= GATTACeeA \end{aligned}$$

le score est de 17.

□

Question 4.2

Donnez un algorithme permettant de trouver le meilleur alignement (celui ayant le score le plus élevé). Expliquez comment vous reconstruisez la solution.

Solution :

Le meilleur alignement de ua et vb s'obtient soit à partir de celui de u et v en alignant a et b, soit à partir de celui de ua et v en alignant b avec un espacement, soit enfin à partir de celui de u et vb en alignant a avec un espacement.

Soit $|u| = n$ et $|v| = m$, on note $ms_{i,j}$ (pour $0 \leq i \leq n, 0 \leq j \leq m$) le score d'un meilleur alignement entre $u_1u_2\ldots u_i$ et $v_1v_2\ldots v_j$. On pose $ms_{0,j} = qj$ et $ms_{i,0} = qi$ ce qui correspond à aligner une séquence uniquement avec des espacements.

$$ms_{i,j} = \max \begin{cases} ms_{i-1,j-1} + p(u_i, v_j) \\ ms_{i-1,j} + q \\ ms_{i,j-1} + q \end{cases}$$

Pour reconstruire la solution, on part de $ms_{n,m}$ et on remonte dans le tableau de $ms_{i,j}$ vers celui des $ms_{i-1,j-1}, ms_{i-1,j}$ ou $ms_{i,j-1}$ qui a permis d'obtenir la valeur de $ms_{i,j}$.

Tableau pour le meilleur alignement pour $GATTACA$ et $ATACGTA$:

□

Question 4.3

Quelle est la complexité en temps et en espace de votre algorithme ?

Solution :

		0	A 1	T 2	A 3	C 4	G 5	T 6	A 7
	0	0	-2	-4	-6	-8	-10	-12	-14
		↑							
G	1	-2	-1	-3	-5	-7	-3	-5	-7
			↖						
A	2	-4	3	1	2	0	-2	-4	0
				↖					
T	3	-6	1	8	6	4	2	3	1
				↑					
T	4	-8	-1	6	7	5	3	7	5
					↖				
A	5	-10	-3	4	11	9	7	5	12
						↖			
C	6	-12	-5	2	9	16	← 14	← 12	10
									↖
A	7	-14	-7	0	7	14	15	13	17

L'algorithme s'exécute en $O(nm)$, et nécessite $O(nm)$ espace mémoire.

□

Question 4.4

Vous disposez maintenant d'une famille de k chaînes de caractères $S_1, S_2, \ldots, S_k$. On souhaite obtenir un alignement des k séquences, c'est à dire k nouvelles chaînes de caractères $S'_1, S'_2, \ldots, S'_k$ toutes de même longueur n. On veut en fait minimiser la somme des distances entre toutes les paires de séquences, en reprenant le principe d'alignement de séquences précédent : on fait correspondre au mieux les séquences en ajoutant au besoin des espacements. La distance entre deux séquences étant la somme des distances caractère par caractère :

$$\sum_{1 \leq i < j \leq k} \sum_{1 \leq l \leq n} \delta(S'_i[l], S'_j[l])$$

avec $\delta(S_i, S_i) = 0$, $\delta \geq 0$ et δ est symétrique et respecte l'inégalité triangulaire.

Pour $k = 2$, cela revient au problème précédent. Est-il possible d'étendre votre algorithme pour traiter le problème pour k quelconque ? Quelle serait alors la complexité temporelle de l'algorithme ? Est-ce polynomial en la taille des données (en le nombre de séquences à traiter, k ; et en la taille des chaînes, n) ?

Solution :

On peut étendre l'algorithme précédent en prenant un tableau k dimensionnel, mais on aurait alors une complexité en $O(n^k)$ pour k séquences de longueur n. On n'est donc pas polynomial.

En fait le problème de décision associé est $\mathcal{NP}$-complet.

□

Références

[1] D. Aldous and P. Diaconis. Longest increasing subsequences : From patience sorting to the Baik-Deift-Johansson theorem. *BULLETIN-AMERICAN MATHEMATICAL SOCIETY*, 36 :413–432, 1999.

[2] Sergei Bespamyatnikh and Michael Segal. Enumerating longest increasing subsequences and patience sorting. *Inf. Process. Lett.*, 76(1-2) :7–13, 2000.

[3] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. *Information Processing Letters*, 6(3) :80–82, 1977.