TD 4

Hanoï itératif et correction du partiel de mars 2007

Exercice 1 (Tours de Hanoi : jouer sans ordinateur)**.**
Le jeu des tours de Hanoi se résout très simplement et élégamment de manière récursive au sens où on peut obtenir la liste des actions à effectuer, pour un n quelconque. Cependant cela ne permet pas à un joueur humain de résoudre le problème s'il ne dispose pas d'ordinateur : la solution suppose de mémoriser et surtout de reproduire un grand nombre d'états du jeu ce qui est hors de portée de notre esprit. Nous proposons ici de trouver une solution sous la forme d'une méthode à appliquer pas à pas pour résoudre le jeu.

Un état *du jeu est une distribution des disques sur les trois tiges a (départ), b (intermédiaire), c (arrivée). Dans l'état initial la tige a contient n disques empilés du plus large (à la base), au plus petit (au sommet). Les deux autres tiges sont vides. Il faut déplacer un par un les disques d'une tige vers une autre sans que jamais un disque ne soit posé sur un disque plus petit. L'état final que l'on veut atteindre est celui où la tige d'arrivée contient les n disques.*

1. *Il n'est pas très simple de trouver une méthode pas à pas mais une suite de remarques et de questions va nous aider.*
   (a) *Il existe une solution au jeu : la méthode récursive effectue une succession d'actions sans* essais-erreurs *et en un nombre minimal de déplacements pour parvenir au résultat.*
   (b) *On ne fera jamais deux déplacements successifs conduisant à un état que l'on pouvait atteindre avec un seul déplacement, (optimalité).*
   (c) *On ne fera jamais deux déplacements successifs* opposés *(d'un état e_0 vers un état e_1 puis retour à e_0) (optimalité).*
   (d) *Combien au plus de déplacements successifs peut on effectuer sans déplacer le plus petit disque (noté par la suite* ppd*) ?*
   (e) *Combien au plus de déplacements successifs du plus petit disque peut on effectuer ?*
   (f) *En déduire qu'une fois sur deux on déplace* ppd*.*
   (g) *Montrer que le déplacement suivant un déplacement de* ppd *est forcé.*
   (h) *La méthode proposée devra donc uniquement déterminer à chaque fois (un déplacement sur deux) le déplacement de* ppd*. Devra t'on distinguer les cas où n est pair et n impair (voir $n = 1$ et $n = 2$ puis* n =3*) ?*
   (i) *Pour déterminer le déplacement de* ppd *on va chercher une régularité. Si on essaie des permutations circulaires, peut-on formuler une hypothèse permettant de résoudre* n=4 *? (Il est conseillé sur le papier de reproduire dans l'ordre a, b, c, a , b , c ...l'état des tiges après chaque déplacement de* ppd*)*
   (j) *En induire une solution pour n pair*
   (k) *En induire une solution pour n impair*
   (l) *On peut démontrer par récurrence la correction de l'algorithme en utilisant l'idée de la solution récursive mais c'est un peu délicat ...*
2. *Nous allons maintenant mettre en oeuvre cette solution à l'aide de* piles *: une pile p est un objet muni des opérations suivantes :*
   - *empiler(e, p) ajoute l'élément e au sommet de la pile p*
   - *dépiler(p) ôte l'élément au sommet de la pile p*
   - *sommet(p) renvoie la valeur du sommet de la pile p*
   - *nouvelle$(nomp)$crée une pile vide de nom nomp et renvoie sa référence p.*
   - *vide(p) renvoie Vrai si la pile p est vide*
   - *hauteur(p) renvoie le nombre d'éléments dans la pile.*

Nous y ajoutons la fonction suivante :
– nom(p) renvoie le nom de la pile (ici ce sera a, b *ou* c*).*
Ecrire un algorithme implantant la méthode proposée. Pour cela les variables de type pile *doivent être considérés comme des références : si on écrit* `nouvelle(p2,"A"); p1<-p2` *cela signifie que* p1 *représente la même pile que* `p2` *mais une seule pile de nom* `"A"` *a été crée. Cela permettra de désigner les piles lorsque l'on fera des permutations circulaires (la même référence* p1 *représentera tout à tour a, b et c). On affichera les déplacements comme pour la version récursive.*

Exercice 2.
Rappeler les définitions utilisées et justifier vos réponses. 4 pt

1. *Est-ce que* $\log(n/2) = \Omega(\log n)$ *?* (1 pt)
2. *Est-ce que* $n = \Omega(n \log n)$ (1 pt)
3. *Est-ce que* $\log(n!) = O(n \log n)$ (1 pt)
4. *Soit un algorithme A. Est-il correct de dire du temps d'exécution de A que : si le pire cas est en* $O(f(n))$ *et le meilleur cas en* $\Omega(f(n))$ *alors en moyenne A est en* $\Theta(f(n))$ *?* (1 pt)

Exercice 3 (Arbre de décision, meilleur cas)**.**
Rappel : le tri bulle s'arrête lorsqu'il a fait une passe au cours de laquelle il n'y a eu aucun échange. 4,5 pt

1. *Dessiner l'arbre de décision du tri bulle sur un tableau de trois éléments* $[a, b, c]$*.* (1 pt)
2. *On note* $C(N)$ *le nombre de comparaisons faites par le tri bulle dans le meilleur des cas sur un tableau de taille* N*. Quel tableau en entrée donne le meilleur cas du tri bulle ? Combien vaut* $C(N)$ *exactement ? (En fonction de* N*.)* (0,5 pt)

On raisonne maintenant sur les algorithmes de tri généralistes (par comparaison).

3. *Est-il possible qu'un algorithme de tri fasse moins que* $N-1$ *comparaisons en meilleur cas ? (Démontrer.)* (1 pt)
4. *Rappeler la borne asymptotique inférieure du nombre de comparaisons nécessaires dans un tri généraliste.* (0,5 pt)
5. *Sans utiliser le résultat que vous venez de rappeler, montrer que pour* $N > 2$*, il n'y a pas d'algorithme A qui trie n'importe quel tableau de taille* N *en au plus* $N-1$ *comparaisons. (Considérer les* $N!$ *permutations de* $0, \ldots, N-1$ *en entrée et raisonner sur la hauteur de l'arbre de décision de A.)* (1,5 pt)

Exercice 4 (Min et max)**.**
On se donne un tableau d'entiers T*, non trié, de taille* N*. On cherche dans* T *le plus petit entier (le minimum) ainsi que le plus grand (le maximum).* Si vous écrivez en C : ne vous souciez pas de la manière de rendre les deux entiers : `return(a, b)` où a est le minimum et b le maximum sera considéré comme correct. 3 pt

1. *Écrire un algorithme* Minimum(T) *(C ou pseudo-code) qui prend en entrée le tableau* T *et rend le plus petit de ses entiers.* (1 pt)

Pour trouver le maximum, on peut d'écrire l'algorithme Maximum(T) *équivalent (en inversant simplement la relation d'ordre dans* Minimum(T)*).*
On peut alors écrire une fonction MinEtMax(T) *qui renvoie* (Minimum(T), Maximum(T))*.*

2. *Combien la fonction* MinEtMax *fera t-elle de comparaisons, exactement, sur un tableau de taille* N *?* (1 pt)

On propose la méthode suivante pour la recherche du minimum et du maximum. Supposons pour simplifer que N *est pair et non nul.*
On considère les éléments du tableau par paires successives : $(T[0], T[1])$ *puis* $(T[2], T[3])$*,* $(T[4], T[5])$*,* $\ldots (T[N-2], T[N-1])$*.*
– *On copie la plus petite valeur entre* $T[0]$ *et* $T[1]$ *dans une variable* Min *et la plus grande dans une variable* Max*.*

- *Pour chacune des paires suivantes,* $(T[2i], T[2i+1])$ *:*
  - *on trouve le minimum entre* $T[2i]$ *et* $T[2i+1]$ *et on le range dans* MinLocal *de même le maximum entre* $T[2i]$ *et* $T[2i+1]$ *est rangé dans* MaxLocal *;*
  - *On trouve le minimum entre* Min *et* MinLocal *et on le range dans* Min *de même le maximum entre* MaxLocal *et* Max *est rangé dans* Max.
- *On rend* Min *et* Max.

3. *On réalise cet algorithme avec un minimum de comparaisons pour chaque étape : expliquer quelles seront les comparaisons (mais inutile d'écrire tout l'algorithme). Combien fait-on de comparaisons au total ?* (1 pt)

Exercice 5 (Réussite *(patience sort)*). 8,5 pt

On se donne un paquet σ de N cartes sur lesquelles sont inscrites des valeurs deux à deux comparables. Pour simplifier, on considérera que ces valeurs sont les entiers de 0 *à* $N-1$*, dans le désordre mais sans répétition.*

On fait une réussite de la manière suivante :
- *prendre la première carte du paquet et la poser devant soi, à sa gauche, inscription au dessus (de manière à pouvoir voir sa valeur).*
- *Poser tour à tour les cartes suivantes sur la table, inscription au dessus, en formant des piles de cartes. Pour chaque nouvelle carte* x *on peut :*
  - *soit la poser sur une carte déjà posée sur la pile* y*, à la condition que la valeur de* x *soit inférieure à la valeur de* y *;*
  - *soit commencer une nouvelle pile de carte en la posant à droite de toutes les piles existantes.*
- *On s'arrête losqu'il n'y a plus de cartes dans le paquet.*

Le but du jeu est de finir avec un nombre minimum de piles de cartes devant soi.

On emploie la stratégie qui consiste à placer une nouvelle carte toujours le plus à gauche possible. Par exemple, si le paquet de cartes contient au départ la suite de cartes :

$$\sigma = 6, 1, 5, 0, 7, 2, 9, 4, 3, 8$$

alors les piles de cartes seront successivement comme ceci (en gras le dernier élément empilé) :

			0	0	0	0	0	0 **3**	0 3
	1	1	1	1	1 **2**	1 2	1 2 **4**	1 2 4	1 2 4 **8**
6	6	6 **5**	6 5	6 5 **7**	6 5 7	6 5 7 **9**	6 5 7 9	6 5 7 9	6 5 7 9

Le nombre de piles est alors 4.

On va montrer que la stratégie du plus à gauche *minimise le nombre de piles.*

On suppose que l'on a écrit l'algorithme Réussite(σ) *qui prend σ en entrée (comme un tableau d'éléments) et rend les p piles obtenues par la stratégie du plus à gauche. Cet algorithme rend son résultat sous la forme d'un tableau* T *de* p *piles non vides (la première pile est* $T[0]$*, la dernière* $T[p-1]$*).*

1. *Le nombre de piles dépend de la suite σ des cartes du paquet de départ. Donner une suite de* N *cartes réalisant le meilleur cas, c'est à dire donnant le nombre minimum de piles et une suite de* N *cartes réalisant le pire cas, c'est à dire donnant le nombre maximum de piles (pour l'algorithme* Réussite(σ)*).* (1 pt)

On s'intéresse maintenant à l'écriture de l'algorithme Réussite(σ)*. On dispose des fonctions standards sur les piles :*
- EstVide(P) *qui rend* 1 *si la pile* P *est vide et* 0 *sinon.*
- Tête(P) *qui rend la valeur de l'élément du dessus de la pile* P *sans dépiler cet élément.*
- Empiler(e, P) *qui empile l'élément e sur la pile* P.
- Dépiler(P) *qui dépile l'élément du dessus de la pile* P *et rend sa valeur.*

Au départ de l'algorithme Réussite(σ) *on dispose d'un tableau* T *de piles suffisement grand, et qui ne contient que des piles vides (p vaut* 0*). Au cours de l'exécution de l'algorithme* Réussite(σ)*, on empile des éléments de σ dans les piles de* T*. Pour poser une nouvelle carte* x *il faut comparer la valeur de* x *avec les valeurs des têtes de piles.*

2. *Quelle méthode peut-on utiliser pour chercher la place de x en limitant le nombre de comparaisons ? S'il y a p piles combien fera t-on au plus de comparaisons pour insérer une nouvelle carte x (donner un majorant asymptotique) ? En déduire que* RÉUSSITE(σ) *peut être écrite de manière à faire $O(N \log N)$ comparaisons.* (1 pt)

Une suite extraite de σ est une suite formée d'éléments de σ pris dans le même ordre qu'ils apparaissent dans σ, sans forcément choisir des éléments successifs. On considère les suites extraites croissantes de σ. Par exemple, $1, 5, 7, 8$ et $0, 2, 3$ sont des suites extraites croissantes de $\sigma = 6, 1, 5, 0, 7, 2, 9, 4, 3, 8$. On note $l(\sigma)$ la longueur maximum des suites extraites croissantes de σ. Dans notre exemple $l(\sigma) = 4$.

3. *Montrer que $l(\sigma)$ minore le nombre de piles obtenues quel que soit la stratégie. (Indication : il faut montrer que si $a_1 < \ldots < a_k$ est une suite extraite de σ alors il faut au moins k piles).* (1 pt)

On veut montrer que si RÉUSSITE(σ) *forme p piles, alors il existe une suite extraite croissante de σ de longueur p. À chaque fois qu'on pose une nouvelle carte x sur une pile, sauf si on la pose sur la première pile, on dessine une flêche de x vers la carte du dessus de la pile à sa gauche.*

Voici ce que ça donne sur l'exemple, pour les six premières insertions.

	1	1	**0**	0	0
	6	6←**5**	1	1	1←**2**
6			6←5	6←5←**7**	6←5←7

4. *Compléter l'exemple précédent (sur votre copie).* (0,5 pt)
5. *En prenant un élément d'une pile et en suivant les flêches on obtient une suite d'éléments de σ (en ordre inverse). Par exemple, 1←5←7. À quoi correspond une telle suite par rapport à σ ? (Justifier.)* (0,5 pt)
6. *Montrer que si* RÉUSSITE(σ) *a formé p piles, alors il existe une suite extraite croissante de σ de longueur p.* (0,5 pt)
7. *Comment peut-on déduire de la question 3 et de la question précédente que $p = l(\sigma)$? Que la stratégie du plus à gauche est optimale ?* (1 pt)

Ainsi la stratégie du plus à gauche, qui peut s'écrire comme un algorithme RÉUSSITE(σ)*, permet de caluler la taille de la plus grande suite extraite croissante en $O(N \log N)$ comparaisons.*

On veut maintenant écrite un algorithme RASSEMBLERPILES(T) *qui prend en entrée le tableau T rendu par* RÉUSSITE(T) *et rend le tableau trié des éléments de σ. Dans chacune des p piles les éléments sont triés (le plus petit en haut de la pile), il suffit donc d'interclasser les p piles d'éléments. On peut rendre l'interclassement plus efficace, en observant qu'au départ les éléments du dessus des piles sont également bien ordonnés :*

$$\text{TÊTE}(T[0]) < \text{TÊTE}(T[1]) < \ldots < \text{TÊTE}(T[p-1]). \quad (1)$$

Mais si on retire un élément du dessus d'une des piles, la propriété (1) n'est plus forcément vraie.

8. *Pouvez-vous donner une suite σ la plus courte possible, telle que depiler un élément de la première pile rend la suite des têtes de piles non croissante ?* (0,5 pt)
9. *Après un* DÉPILER($T[0]$) *tel que $T[0]$ contient encore un élément, que faut-il faire pour réordonner les piles de manière à avoir de nouveau la propriété (1) ?* (0,5 pt)
10. *Écrire l'algorithme* RASSEMBLERPILES(T)*, en pseudo-code ou en C.* (2 pt)

Au besoin on pourra faire appel à une fonction ECHANGER(T, i, j) *qui échange dans le tableau T, la pile $T[i]$ avec la pile $T[j]$. En C, on pourra considérer tous les arguments auxilliaires éventuellement nécessaires. Par exemple on pourra considérer que le premier argument est le tableau de piles, que p est donné comme second argument, que le tableau dans lequel ranger le résultat est donné comme troisième argument et que N est donné comme quatrième argument :* `B(pile_t T[], int nb_piles, elements_t res[], int nb_elts)`.

Corrigé

Correction de l'exercice 1.

1. (a) voir la méthode récursive
   (b)
   (c)
   (d) une seule (voir remarque 1c) sinon on effectue deux déplacement opposés entre les deux tiges ne contenant pas *ppd*.
   (e) un seul (voir remarque 1b) sinon on effectue deux déplacements qu'on aurait pu faire en un seul (*ppd* est déplacé d'une tige 1 vers une tige 2 puis de 2 vers 3, ce qu'on aurait pu faire directement par le déplacement $1->3$)
   (f) évident d'après les questions 1d et 1e : on ne peut déplacer deux fois de suite *ppd* et on ne peut faire deux déplacements sans déplacer *ppd*.
   (g) L'un des deux disques est plus grand que l'autre : un seul mouvement est possible.
   (h) Oui. dans le cas$n = 1$ *ppd* est déplacé de la tige de départ (a) vers l'arrivée (c), et pour n = 2 c'est de a vers b.
   (i) On déplace toujours *ppd* vers la tige suivante (ordre a,b,c,a,b,c, ...)
   (j) On déplace toujours *ppd* vers la tige suivante à distance 2 (ordre a,c,b, a, c,b ....)
   (k) voir wikipedia tours de hanoi (attention la démonstration est donnée pour b pile d'arrivée et c pile intermédiaire).
2. Ci-dessous une version en C (sans les implémentations de piles).

```
#include <stdio.h>
#include "piles.h"

/* hanoi iteratif : on transfère n disques de la pile pa vers la pile
   pc (pb est la pile intermediaire).

   Les piles pa pb pc une fois initialisées avec nouvelle(lettre) ont
   des adresses fixes pa pb pc, et p1 p2 p3 vont prendre ces adresses
   et faire des permutations circulaires pour résoudre le problème,
   voir wikipedia tours de hanoi (attention dans wiki c'est b la pile
   d'arrivee, ici c'est c). Selon si n est pair ou impair ce n'est pas
   la même permutation sinon on n'arrive pas sur la bonne pile d'arrivee.

   Strategie : un coup sur deux on deplace le pp disque, le deuxième
   coup est forcé.

   - Cas impair (n=1, ...) on commence par pp disque de a ->b (depart
   vers inter) puis de b-> c puis de c-> a etc ..

   - Cas pair (n=2, ...) on commence par pp disque de a -> c (depart
   vers arrivee) puis de c-> b puis de b-> a puis de a -> b etc ...
*/

int main () {
pile *pa, *pb, *pc;
pile *p1, *p2, *p3, *paux;
int n, i;
element s;
printf("nombre de disques?\n");
```

```
scanf("%d",&n);

pa=nouvelle('a'); pb=nouvelle('b'); pc=nouvelle('c');
for (i = n; i >= 1; i--){
        empiler(i, pa);
}
printf("depart\n");
affiche(pa);
p1 = pa;
if (n % 2 == 0){
/* a b c = 1 2 3 pour les deplacements du pp disque
   (une fois sur deux) a -> b -> c -> a -> b -> c ->
   a....... */
p2 = pb; p3 = pc;
}
else {
/* a b c = 1 3 2 pour les deplacements du pp disque
   (une fois sur deux) a -> c -> b -> a -> c -> b -> a
   ....... */
p2 = pc; p3 = pb;
}
while(  hauteur(pc) != n ){/* ou :  ! (vide(pa) &&  vide(pb) ) */
s=sommet(p1);
depiler(p1);
empiler(s,p2);
printf("%c -> %c", p1->nom, p2->nom);
if (! (vide(p1) &&  vide(p3) )) {/* sinon pb resolu */
if (vide(p1) ){
s = sommet(p3);
depiler(p3);
empiler(s, p1);
printf(" et %c -> %c ", p3->nom, p1->nom);
}
else if (vide(p3)){
s = sommet(p1);
depiler(p1);
empiler(s, p3);
printf(" et %c -> %c ", p1->nom, p3->nom);
}
else if (sommet(p1) < sommet (p3)){
s = sommet(p1);
depiler(p1);
empiler(s,p3);
printf(" et %c -> %c ", p1->nom, p3->nom);
}
else {
s = sommet(p3);
depiler(p3);
empiler(s,p1);
printf(" et %c -> %c ", p3->nom, p1->nom);
}
}
printf("\n");
paux = p1;
```

```
p1 = p2;
p2 = p3;
p3 = paux;
}
printf("depart\n");
affiche(pa);

printf("inter\n");
affiche(pb);

printf("arrivee\n");
affiche(pc);
return 0;
}
// --- execution ---
//nombre de disques?
//2
//depart
//     1
//     2
//pile a
//a -> b et a -> c
//b -> c
//depart
//pile a
//inter
//pile b
//arrivee
//     1
//     2
//pile c
```

Correction de l'exercice 2.

1. Réponse Oui. Par définition, on a $\log(n/2) = \Omega(\log n)$ si et seulement si :

$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 < c \log n \leq \log(n/2).$$

   On a $\log(n/2) = \log n - 1 = \frac{1}{2}\log n + (\frac{1}{2}\log n - 1)$. Posons $c = \frac{1}{2}$ et $n_0 = 4$. Lorsque $n \geq n_0$, $\frac{1}{2}\log n - 1 \geq \frac{1}{2}\log n_0 - 1 = 0$ et donc $\log(n/2) \geq \frac{1}{2}\log n$. Ce qui prouve bien que $\log(n/2) = \Omega(\log n)$.

2. Réponse non. Par définition, on a $n = \Omega(n \log n)$ si et seulement si :

$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 < cn \log n \leq n.$$

   Supposons l'existence d'un tel c et d'un tel n_0. Alors pour n'importe quel $n \geq n_0$ on doit avoir $\log n \geq \frac{1}{c}$. Ceci est en contradiction avec le fait que $\lim_{+\infty} \log n = +\infty$.

3. Réponse oui. Par définition, on a $\log(n!) = O(n \log n)$ si et seulement si :

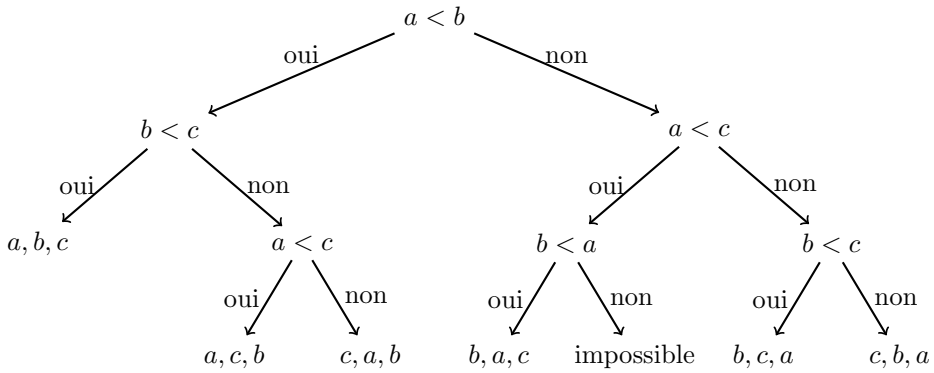
$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 < \log(n!) \leq cn \log n.$$

   Soient $c = 1$ et $n_0 = 0$. Supposons $n \geq n_0 = 0$. On a $n! \leq n^n$ donc par croissance du log, $\log(n!) \leq \log(n^n) = n \log n$. Ce qui montre bien que $\log(n!) = O(n \log n)$.

4. Réponse Oui. Le meilleur cas minore la moyenne donc un minorant asymptotique du meilleur cas est également un minorant asymptotique de la moyenne. De même le pire cas majore la moyenne donc un majorant asymptotique de la moyenne est également un majorant asymptotique de la moyenne. Ainsi la moyenne est en $\Omega(f(n))$ et en $\Gamma(f(n))$ c'est à dire bien en $\Theta(f(n))$.

Correction de l'exercice 3.

1.



2. Le meilleur cas du tri bulle se réalise lorsque la première passe est sans échange, c'est à dire sur un tableau déjà ordonné, par exemple : $0, 1, \ldots, N-1$. Ce tri fait alors $C(N) = N-1$ comparaisons (entre 0 et 1 puis 1 et 2, $\ldots$, $N-2$ et $N-1$).
3. Il n'est pas possible qu'un algorithme fasse moins de $N-1$ comparaisons en meilleur cas. En effet, on sait (cours) que pour trouver le maximum dans un tableau de N éléments, quel que soient les éléments du tableau, il faut faire au moins $N-1$ comparaisons. Or une fois que le tableau est trié on peut trouver ce maximum sans faire de comparaison supplémentaire. Il n'est donc pas possible qu'on ai fait moins de $N-1$ comparaisons au cours du tri.
4. Les tris par comparaison font en moyenne $\Omega(N \log N)$ comparaisons. .
5. Supposons qu'un algorithme de tri A fasse au plus $N-1$ comparaisons sur un tableau de taille N, quel que soit l'ordre initial des éléments. On considère son arbre de décision. Celui est par hypothèse de hauteur $N-1$. Il a donc 2^{N-1} feuilles. Mais toutes les permutations doivent apparaître comme des feuilles différentes de cet arbre et il y en a $N!$. Pour $N > 2$, on a

$$N! = \underbrace{N \times \ldots \times 3 \times 2}_{N-1 \text{ termes}} \times 1 > \underbrace{2 \times \ldots \times 2 \times 2}_{N-1 \text{ termes}} \times 1 = 2^{N-1}$$

Ce qui est une contradiction.

Correction de l'exercice 4.

1. L'algo en C :

```
int Minimum(int T[]){
  int i, x;
  x = T[0];
  for (i = 1; i < taille(T); i++){
    if (T[i] < x) {
      x = T[i];
    }
  }
  return x;
}
```

2. La fonction MINIMUM fera $N-1$ comparaisons, de même pour la fonction MAXIMUM. Donc la fonction MINETMAX en fera $2 \times (N-1) = 2N-2$.
3. Pour réaliser l'algorithme il faut plusieurs fois trouver le minimum et le maximum entre deux entiers a et b, et ceci se fait en une seule comparaison (si $a < b$ alors le minimum est a et le maximum est b sinon c'est l'inverse). On le fait une fois entre $T[0]$ et $T[1]$. Puis pour chaque paire suivante, on fait une comparaison pour trouver le minimum (MINLOCAL) et le maximum (MAXLOCAL) dans la paire plus une comparaison pour trouver le minimum entre MINIMUM et MINLOCAL et encore une autre comparaisons pour trouver le maximum entre MAXIMUM et MAXLOCAL. Au final cela fait une comparaison pour la première paire et trois comparaisons pour les chacune des $N/2-1$ paires suivantes, c'est à dire $3N/2-2$ comparaisons.

Correction de l'exercice 5.

1. La suite décroissante $\sigma = N-1, N-2, \ldots, 0$ donne le meilleur cas. Dans ce cas il n'y a qu'une seule pile. La suite croissante $\sigma = 0, \ldots, N-1$ donne le pire cas. Dans ce cas il y a autant de piles que de cartes c'est à dire N.
2. La suite des cartes en haut des piles est croissante. Pour trouver l'emplacement d'une nouvelle carte, il suffit donc de chercher la dernière pile dont la carte du dessus y a une valeur inférieure à x et de poser la carte x sur la pile suivante. Pour chercher cette pile, on cherche par dichotomie x dans le tableau des têtes de piles. Ceci prend un nombre de comparaisons en log du nombre de piles. Comme il y a au plus N piles, insérer une carte prend $O(\log N)$ comparaisons. Donc insérer N cartes prend $O(N \log N)$ comparaisons.
3. On remarque que si x vient d'être placé sur une pile $T[j]$ alors toute carte y qui suit x dans σ et qui est plus grande que x doit être placée après la pile $T[j]$. En effet au moment de placer x, toutes les piles avant $T[j]$ ont des cartes du dessus de valeur inférieure à x. Ces valeurs du dessus, jusqu'à $T[j]$ incluse ne peuvent que diminuer par ajout de nouvelles cartes. Donc au moment de placer y, il n'est pas possible de le poser sur une des piles avant $T[j+1]$.

   Supposons que $a_1 < \ldots < a_k$ est une sous-suite de σ. Une fois que a_i est placé sur une pile, a_{i+1} est nécessairement placé sur une des piles suivantes. Il faut donc au moins k piles.
4.

	0 1 2 6 5←7←**9**	0 1 2←**4** 6 5←7←9	0 **3** 1 2←4 6 5←7←9	0 3 1 2←4←**8** 6 5←7←9
...				

5. Si il y a une flêche d'un élément x vers élément y alors l'élément x est plus grand que l'élément y et x a été posé après y. Ainsi une suite obtenue en suivant les flêches donne, en ordre inverse, une suite croissante d'éléments de σ dans l'ordre de leur apparation dans σ, c'est à dire une sous-suite croissante de σ.
6. Tout élément qui n'est pas dans la première pile possède une flêche vers un élément dans la pile juste avant. Prenons n'importe quel élément de la dernière pile $T[p-1]$ et suivons les flêches. Nous aurons alors une sous-suite croissante de σ de longueur p.
7. La question précédente nous montre que le nombre de piles p formées par RÉUSSITE(σ) est égal à la longueur d'au moins une sous-suite croissante de σ, et donc que p minore $l(\sigma)$. Nous avons aussi montré (question 3) que quel que soit la stratégie le nombre de piles est toujours supérieur ou égal à $l(\sigma)$. On en déduit l'égalité $p = l(\sigma)$. D'autre part puisque toute stratégie forme au moins $l(\sigma)$ piles et que RÉUSSITE en forme exactement $l(\sigma)$, c'est que RÉUSSITE est optimal.
8. En prenant la suite $\sigma = 3, 1, 2$ on aura les deux piles : $\begin{array}{c} 1 \\ \hline 3 \;\; 2 \end{array}$ et après avoir enlevé 1 les têtes de piles ne vont plus croissante.
9. Une fois que la carte du dessus de $T[0]$ est enlevée les têtes des piles $T[1]$, ..., $T[p]$ sont toujours croissantes mais la tête de $T[0]$ n'a plus forcément une valeur inférieur à celle de la

tête de $T[1]$. Il faut alors procéder par insertion de $T[0]$ dans la suite du tableau de manière à retrouver la propriété.

10.
```
void  rassembler(pile_t T[], int nb_piles, elements_t res[], int nb_elts){
  int i;      /* indice de res (le tableau résultant)              */
  int j;      /* indice de T (le tableau de piles)                 */
  int k = 0;  /* indice dans T de la première pile non vide        */
  for (i = 0; i < nb_elts; i++) {
    res[i] = depiler(T[k]); /* On récupère le plus petit élément.   */
    if ( estvide(T[k]) ) {  /* Pile vide: passer à la suivante.     */
      k++;
    }
    else {                  /* Sinon on doit reclasser  les piles.  */
      j = k;
      while ( ( j + 1 < nb_piles )  &&
              ( tete(T[j]) > tete(T[j + 1]) ) ) {
        Echanger(T, j , j + 1)
        j++;
      }
    }
  }
}
```