TD 3

Rechercher et trier

Exercice 1 (Tri sélection)**.**
Soit un tableau indexé à partir de 0 contenant des éléments deux à deux comparables. Par exemple des objets que l'on compare par leurs masses. On dispose pour cela d'une fonction

$$\text{Comparer}(T, j, k) \text{ qui rend : } \begin{cases} -1 \text{ si } T[j] > T[k] \\ 1 \text{ si } T[j] < T[k] \\ 0 \text{ lorsque } T[j] = T[k] \text{ (même masses).} \end{cases}$$

1. Écrire un algorithme Minimum qui rend le premier indice auquel apparaît le plus petit élément du tableau T.
2. Combien d'appels à la comparaison effectue votre fonction sur un tableau de taille N ?

On dispose également d'une fonction Échanger(T, j, k) qui échange $T[j]$ et $T[k]$. On se donne aussi la possibilité de sélectionner des sous-tableaux d'un tableau T à l'aide d'une fonction Sous-Tableau. Par exemple $T' =$ Sous-Tableau(T, j, k) signifie que T' est le sous-tableau de T de taille k commençant en j : $T'[0] = T[j], \ldots, T'[k-1] = T[j+k-1]$.

3. Imaginer un algorithme de tri des tableaux qui utilise la recherche du minimum du tableau. L'écrire sous forme itérative et sous forme récursive.
4. Démontrer à l'aide d'un invariant de boucle que votre algorithme itératif de tri est correct.
5. Démontrer que votre algorithme récursif est correct. Quelle forme de raisonnement très courante en mathématiques utilisez-vous à la place de la notion d'invariant de boucle ?
6. Combien d'appels à la fonction Minimum effectuent votre algorithme itératif et votre algorithme récursif sur un tableau de taille N ? Combien d'appels à la fonction Comparer cela représente t-il ? Combien d'appels à Échanger ? Donner un encadrement et décrire un tableau réalisant le meilleur cas et un tableau réalisant le pire cas.
7. Vos algorithmes fonctionnent-ils dans le cas où plusieurs éléments du tableau sont égaux ?

Exercice 2 (Interclassement)**.**
Soient deux tableaux d'éléments comparables `t1` et `t2` de tailles respectives n et m, tous les deux triés dans l'ordre croissant.

1. Écrire un algorithme d'interclassement des tableaux `t1` et `t2` qui rend le tableau trié de leurs éléments (de taille $n + m$).

On note $N = n + m$ le nombre total d'éléments à interclasser. En considérant le nombre de comparaisons, en fonction de N, discuter l'optimalité de votre algorithme en pire cas et en meilleur cas à l'aide des questions suivantes (démontrer vos résultats).

2. À votre avis, sans considérer un algorithme en particulier, dans quel cas peut-il être nécessaire de faire le plus de comparaisons : n grand et m petit ou bien n et m à peu près égaux ?

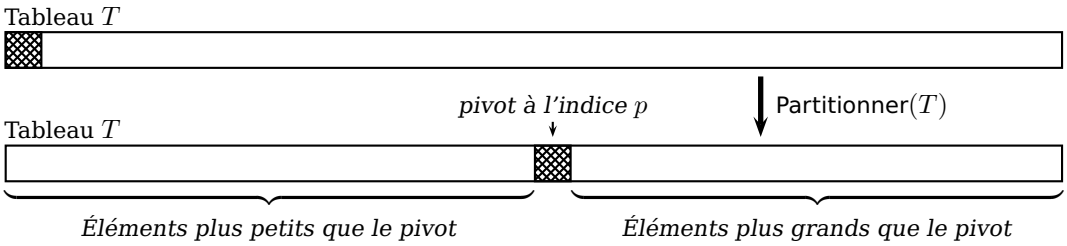
Dans la suite, on suppose que n et m sont égaux (donc $N = 2n$).

3. Dans le pire des cas, combien de comparaisons faut-il faire pour réussir l'interclassement ? Cela correspond t-il au nombre de comparaisons effectuées par votre algorithme dans ce cas ?
4. Toujours sous l'hypothèse $n = m$, quel est le meilleur cas ? En combien de comparaisons peut-on le résoudre ? Comparer avec votre algorithme.

Exercice 3.
Montrer que :

$$\log(n!) = \Omega(n \log n). \tag{1}$$

Exercice 4.
Soit un tableau T de N éléments deux à deux comparables. On souhaite partitionner le tableau autour de son premier élément $T[0]$, appelé pivot. Après partition, le pivot est à l'indice p, les éléments plus petits que le pivot sont (dans le désordre) aux indices inférieurs à p et les éléments strictement plus grands que le pivot sont (dans le désordre) aux indices supérieurs à p.



Écrire l'algorithme de partitionnement de manière à ce qu'il effectue $N-1$ (ou N) comparaisons.

2 pt
18 min

Exercice 5 (Min et max (partiel 2007))**.**
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.*

tot: 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
9 min

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
9 min

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
9 min

Exercice 6.
Montrer que dans un arbre binaire, si la profondeur maximale des feuilles est k alors le nombre de feuilles est au plus $2^k - 1$.

Corrigé

Correction de l'exercice 1.

1. On écrit une fonction itérative, qui parcourt le tableau de gauche à droite et maintient l'indice du minimum parmi les éléments parcourus.

```
int iminimum(tableau_t t){ /* t ne doit pas être vide */
  int j, imin;
  imin = 0;
  for (j = 1; j < taille(t); j++){
    /* Si T[imin] > T[j] alors le nouveau minimum est T[j] */
    if ( 0 > comparer(t, imin, j) ) imin = j;
  }
  return imin;
}
```

2. On fait exactement taille(tab) $-1 = N - 1$ appels.

3. On cherche le minimum du tableau et si il n'est pas déjà à la première case, on échange sa place avec le premier élément. On recommence avec le sous-tableau commençant au deuxième élément et de longueur la taille du tableau de départ moins un. ça s'écrit en itératif comme ceci :

```
1  void triselection(tableau_t tab){
2    int n, j;
3    for (n = 0; n < taille(tab) - 1; n++) {
4      j = iminimum(sous_tableau(tab, n, taille(tab) - n));
5      if (j > 0) echanger(tab, n + j, n);
6    }
7  }
```

et en récursif comme ceci :

```
1  void triselectionrec(tableau_t tab){
2    int j;
3    if ( taille(tab) > 1 ){
4      j = iminimum(tab);
5      if (j > 0) echanger(tab, j, 0);
6      triselectionrec(soustableau(tab, 1, taille(tab) - 1));
7    }
8  }
```

4. Traitons le cas itératif.

On pose l'invariant : le tableau a toujours le même ensemble d'éléments mais ceux indexés de 0 à $n-1$ sont les n plus petits éléments dans le bon ordre et les autres sont indexés de n à $N-1$ (où on note N pour `taille(tab)`).

Initialisation. Avant la première étape de boucle $n = 0$ et la propriété est trivialement vraie (il n'y a pas d'élément entre 0 et $n - 1 = -1$).

Conservation. Supposons que l'invariant est vrai au début d'une étape quelconque. Il reste à trier les élements de n à la fin. On considère le sous-tableau de ces éléments. À la ligne 4 on trouve le plus petit d'entre eux et j prend la valeur de son plus petit indice dans le sous-tableau (il peut apparaître à plusieurs indices). L'indice de cet élément e dans le

tableau de départ est $n + j$. Sa place dans le tableau trié final sera à l'indice n puisque les autres éléments du sous-tableau sont plus grands et que dans le tableau général ceux avant l'indice n sont plus petits. À la ligne 5 on place l'élément e d'indice $n + j$ à l'indice n (si j vaut zéro il y est déjà on ne fait donc pas d'échange). L'élément e' qui était à cette place est mis à la place désormais vide de e. Ainsi, puisqu'on procède par échange, les éléments du tableau restent inchangés globalement. Seul leur ordre change. À la fin de l'étape n est incrémenté. Comme l'élément que l'on vient de placer à l'indice n est plus grand que les éléments précédents et plus petits que les suivants, l'invariant de boucle est bien vérifié à l'étape suivante.

Terminaison. La boucle termine lorsque on vient d'incrémenter n à $N - 1$. Dans ce cas l'invariant nous dit que : (i) les éléments indexés de 0 à $N - 2$ sont à leur place, (ii) que l'élément indexé $N - 1$ est plus grand que tout ceux là, (iii) que nous avons là tous les éléments du tableau de départ. C'est donc que notre algorithme résout bien le problème du tri.

Pour le cas récursif on raisonne par récurrence (facile). On travaille dans l'autre sens que pour l'invariant : on suppose que le tri fonctionne sur les tableaux de taille $n - 1$ et on montre qu'il marche sur les tableaux de taille n.

5. Pour le tri itératif : on appelle la fonction `iminimum` autant de fois qu'est exécutée la boucle (3-6), c'est à dire $N - 1$ fois. Le premier appel à `iminimum` se fait sur un tableau de taille N, puis les appels suivant se font en décrémentant de 1 à chaque fois la taille du tableau, le dernier se faisant donc sur un tableau de taille 2. Sur un tableau de taille K `iminimum` effectue $K - 1$ appels à des comparaisons `cmptab`. On ne fait pas de comparaison ailleurs que dans `iminimum`. Il y a donc au total $\sum_{k=2}^{N} k - 1 = \sum_{k=1}^{N-1} k = \frac{N(N-1)}{2}$ appels à `cmptab`.
   Chaque exécution de la boucle (3-6) peut donner lieu à un appel à `echangetab`. Ceci fait a priori entre 0 et $N - 1$ échanges. Le pire cas se réalise, par exemple, lorsque l'entrée est un tableau dont l'ordre a été inversé. Dans ce cas on a toujours `j > 0` puisque, à la ligne 4, le minimum n'est jamais le premier élément du sous tableau passé en paramètre. Le meilleur cas ne survient que si le tableau passé en paramètre est déjà trié (dans ce cas `j` vaut toujours 0).
   La version récursive fournit une formule de récurrence immédiate donnant le nombre de comparaisons u_n pour une entrée de taille n, qui se réduit immédiatement :
   $$\begin{cases} u_1 = 0 \\ u_n = u_{n-1} + 1 \end{cases} \iff u_n = n - 1.$$
   Donc $N - 1$ comparaisons pourun tableau de taille N. On fait au plus un échange à chaque appel et le pire et le meilleur cas sont réalisés comme précedemment. Cela donne entre 0 et $N - 1$ échanges.
6. Réponse oui (il faut relire les démonstrations de correction). De plus on remarque qu'avec la manière dont a été écrite notre recherche du minimum, le tri est *stable*. Un tri est dit stable lorsque deux éléments ayant la même clé de tri (ie égaux par comparaison) se retrouvent dans le même ordre dans le tableau trié que dans le tableau de départ. Au besoin on peut toujours rendre un tri stable en augmentant la clé de tri avec l'indice de départ. Par exemple en modifiant la fonction de comparaison de manière à ce que dans les cas où les deux clés sont égales on rende le signe de $k - j$.

Correction de l'exercice 2.

1. On écrit en C. On ne s'occupe pas de l'allocation mémoire, on suppose que le tableau dans lequel écrire le résultat a été alloué et qu'il est passé en paramètre.

```
/* ------------------------------------------------------------------------ */
/* Interclassement de deux tableaux avec écriture dans un troisième tableau */
/* ------------------------------------------------------------------------ */
void interclassement(tableau_t t1, tableau_t t2, tableau_t t){
  int i, j, k;
  i = 0;
  j = 0;
  k = 0;
  for (k = 0; k < taille(t1) + taille(t2); k++){
    if ( j == taille(t2) ){/* on a fini de parcourir t2 */
      for (; k < taille(t1) + taille(t2); k++){
        t[k] = t1[i];
i++;
      }
      break; /* <-------- sortie de boucle */
    }
    if ( i == taille(t1) ){/* on a fini de parcourir t1 */
      for (; k < taille(t1) + taille(t2); k++){
        t[k] = t2[j];
j++;
      }
      break; /* <-------- sortie de boucle */
    }
    if ( t1[i] <= t2[j] ){/* choix de l'élément suivant de t : */
      t[k] = t1[i];                    /* - dans t1;           */
      i++;
    }
    else {
      t[k] = t2[i];                    /* - dans t2.           */
      j++;
    }
  }
}
```

2. En pire cas, notre algorithme effectue $n + m - 1$ comparaisons.

   Pour trouver un minorant du nombre de comparaisons pour n'importe quel algorithme, on raisonne sur le tableau trié t obtenu en sortie. Dans le cas où $n = m$, il contient $2n$ éléments. On considère l'origine respective de chacun de ses éléments relativement aux deux tableaux donnés en entrée. Pour visualiser ça, on peut imaginer que les éléments ont une couleur, noir s'ils proviennent du tableau `t1`, blanc s'ils proviennent du tableau `t2`. On se restreint aux entrées telles que dans `t` l'ordre entre deux éléments est toujours strict (pas de répétitions des clés de tri).

   Lemme 1. *Quel que soit l'algorithme, si deux éléments consécutifs* `t[i]`, `t[i+1]` *de* t *ont des provenances différentes alors ils ont nécessairement été comparés.*

   Preuve. Supposons que ce soit faux pour un certain algorithme A. Alors, sans perte de généralités (quitte à échanger `t1` et `t2`), il existe un indice `i` tel que : `t[i]` est un élément du tableau `t1`, disons d'indice `j` dans `t1` ; `t[i+1]` est un élément du tableau `t2`, disons d'indice `k` dans `t2` ; ces deux éléments ne sont pas comparés au cours de l'interclassement. (Remarque : `i` est égal à `j + k`). On modifie les tableaux `t1` et `t2` en échangeant `t[i]` et `t[i+1]` entre ces deux tableaux. Ainsi `t1[j]` est maintenant égal à `t[i+1]` et `t2[k]` est égal à `t[i]`. Que fait A sur cette nouvelle entrée ? Toute comparaison autre qu'une comparaison entre `t1[j]` et `t2[k]` donnera le même résultat que pour l'entrée précédente (raisonnement

par cas), idem pour les comparaisons à l'intérieur du tableau `t`. Ainsi l'exécution de A sur cette nouvelle entrée sera identique à l'exécution sur l'entrée précédente. Et `t1[j]` sera placé en `t[i]` tandis que `t2[k]` sera placé en `t[i + 1]`. Puisque maintenant `t1[j]` est plus grand que `t2[k]`, A est incorrect. Contradiction.

Ce lemme donne un minorant pour le nombre de comparaisons égal au nombre d'alternance entre les deux tableaux dans le résultat. En prenant un tableau `t` trié de taille $2n$ on construit des tableaux en entrée comme suit. Dans `t1` on met tous les éléments de `t` d'indices pairs et dans `t2` on met tous les éléments d'indices impairs. Cette entrée maximise le nombre d'alternance, qui est alors égal à $2n-1$. Par le lemme, n'importe quel algorithme fera alors au minimum $2n-1$ comparaisons sur cette entrée (et produira `t`). Notre algorithme aussi. Donc du point de vue du pire cas et pour $n = m$ notre algorithme est optimal. Des résultats en moyenne ou pour les autres cas que $n = m$ sont plus difficiles à obtenir pour le nombre de comparaisons. On peut remarquer que pour $n = 1$ et m quelconque notre algorithme n'est pas optimal en nombre de comparaisons (une recherche dichotomique de la place de l'élément de `t1` serait par exemple plus efficace).

Par contre, il est clair que le nombre minimal d'affectations sera toujours $n + m$, ce qui correspond à notre algorithme.

Correction de l'exercice 3.

Solution 1 Soit $n > 0$. Si n est pair alors $n = 2k$ avec $k > 0$. Dans ce cas :

$$(2k)! \geq (2k) \times \ldots \times k \geq \underbrace{k \times \ldots \times k}_{k+1 \text{ termes}} \geq k^k$$

Donc $\log((2k)!) \geq k \log k$, c'est à dire $\log(n!) \geq \frac{1}{2} n \log(n/2)$.

Si n est impair, alors $n = 2k+1$ avec $k \geq 0$. Dans ce cas :

$$(2k+1)! \geq (2k+1) \times \ldots \times (k+1) \geq \underbrace{(k+1) \times \ldots \times (k+1)}_{k+1 \text{ termes}} = (k+1)^{k+1}$$

Donc $\log((2k+1)!) \geq (k+1)\log(k+1)$ et donc $\log(n!) \geq \frac{1}{2} n \log(n/2)$.

Ainsi pour $n > 0$ on a

$$\log(n!) \geq \frac{1}{2} n \log(n/2).$$

Pour $n \geq 4$, $n/2 \geq \sqrt{n}$, et ainsi $\log(n/2) \geq \log(\sqrt{n}) = \frac{1}{2} \log n$.

Finalement, pour $n \geq 4$,

$$\log(n!) \geq \frac{1}{4} n \log n.$$

Ce qui montre bien que $\log(n!) = \Omega(n \log n)$.

Solution 2 On a (faire un dessin)

$$\log(n!) = \sum_{k=2}^{n} \log k \geq \int_1^n \log t dt.$$

Donc $\log(n!) \geq [t \log t]_1^n = n \log n - n + 1$. Pour $n \geq 4$, $\log n - 1 \geq \frac{1}{2} \log n$. Ainsi pour $n \geq 4$, $\log(n!) \geq \frac{1}{2} n \log$. Ce qui conclue.

Solution 3 On suppose que $n > 0$. On écrit

$$\log(n!) = \sum_{k=1}^{n} \log k$$

Et comme pour la sommation $\sum_{k=1}^{n} k$ on somme deux fois :

$$\begin{aligned} 2\log(n!) &= \sum_{k=1}^{n} \log k + \sum_{k=1}^{n} \log(n+1-k) \\ &= \sum_{k=1}^{n} \log(k(n+1-k)) \end{aligned}$$

Mais lorsque $1 \leq k \leq n$, $k(n+1-k)$ est maximal pour $k = \frac{n+1}{2}$ et minimal pour $k = 1$ et $k = n$ (on raisonne sur $(a+b)(a-b)$ avec $a = \frac{n+1}{2}$ et $b = k - a$).

Ainsi pour tout $1 \leq k \leq n$, $\log(k(n+1-k)) \geq \log(n)$.

On en déduit qu'à partir du rang $n = 1$:

$$\log(n!) \geq \frac{n \log n}{2}.$$

Ce qui montre bien que $\log(n!) = \Omega(n \log n)$.

Correction de l'exercice 4.

L'algorithme est donné dans le poly du cours.

Correction de l'exercice 5.

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 6.

À écrire (récurrence facile).