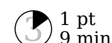
Examen du 8 juin 2009

Le barême est uniquement indicatif.

Vous pouvez écrire les algorithmes en C ou en pseudo-code. Si cela vous pose trop de difficultés n'hésitez pas à répondre en décrivant un algorithme par des phrases ! Vous pouvez faire appel à des fonctions auxilliaires vues en cours (comparaison, tris, sous-tableau, etc.).

Première partie — 11 points

Exercice 1 (Notation asymptotique)**.**
Rappeler les définitions utilisées et justifier (démontrer) vos réponses à partir de ces définitions.


1 pt 9 min

1. Est-ce que $n \log n = O(n^2)$?

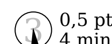
1 pt 9 min

2. Est-ce que $\log(n!) = O(n \log n)$?

1 pt 9 min

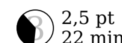
3. Est-ce que $\log(n!) = \Theta(n^2)$

Exercice 2 (Piles)**.**


0,5 pt 4 min

On forme une nouvelle pile en empilant successivement 3, 2, 1 puis on dépile deux éléments. Quel élément reste-t-il dans la pile ? (Facile)

Exercice 3.


2,5 pt 22 min

Soit un tableau t contenant les éléments $10, 5, 1, 9, 11, 3, 4, 2, 7$.

1. Former un tas max en insérant un à un et dans l'ordre les éléments de t. Répondre en représentant le tas obtenu. Supprimer l'élément maximum. Répondre en représentant le nouveau tas.
2. Dans le cours, il y a deux façons de former un tas max à partir d'un tableau d'éléments. On peut insérer un à un les éléments du tableau dans un tas (initialement vide) comme à la première question. Quelle est l'autre façon et quel tas obtient on pour l'entrée t ? Décrire l'algorithme brièvement (donner en particulier le type de maintien utilisé) et donner le tas obtenu.

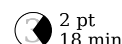
Exercice 4 (Insertion / suppression ABR)**.**

1.5 pt 13 min

Répondre à chaque question en représentant seulement l'arbre obtenu sans justification et, lorsqu'il y a deux réponses correctes possibles, ne donner qu'une seule réponse :

1. Former un arbre binaire de recherche en insérant successivement, et dans cet ordre, les éléments : $10, 8, 3, 9, 12, 4, 11, 13$.
2. Supprimer l'élément 3.
3. En repartant de l'arbre obtenu à la première question, supprimer l'élément 8.

Exercice 5.


2 pt 18 min

Dans cet exercice, l'argument x est un arbre binaire de recherche dont les éléments sont des entiers :

- Écrire une fonction récursive Taille(x) donnant le nombre d'éléments contenus dans l'arbre.
- Écrire une fonction moyenne(x) calculant la moyenne des éléments de l'arbre (supposé non-vide). Pour cela vous aurez besoin de la fonction Taille ainsi que d'une autre fonction récursive qu'il vous appartient de définir.

Exercice 6.

1.5 pt 13 min

Classer les fonctions de complexité $n^2, n, 2^n, n \log n$ par ordre croissant. Pour les complexités $n \log n$ et n^2 donner l'exemple d'un algorithme (du cours ou des TD) qui a asymptotiquement cette complexité en pire cas, en temps.

Seconde partie : problèmes 9 points

Exercice 7.
Étant donné un tableau T de N entiers et un entier x, on veut déterminer s'il existe deux éléments de T dont la somme est égale à x.

3 pt
27 min

1. Donner un algorithme le plus simple possible, basé sur la comparaison, sans faire appel à des algorithmes du cours. Quel est le pire cas ? Donner un équivalent asymptotique du nombre de comparaisons dans le pire cas. (Justifier)
2. Pouvez-vous donner un algorithme en $O(N \log N)$ comparaisons en pire cas ? (Justifier) Vous pouvez utiliser des algorithmes vus en cours et les résultats de complexité sur ces algorithmes.

Exercice 8.
Écrire une fonction PeignerDroite(x) prenant en entrée un arbre binaire et le transformant *en utilisant uniquement des rotations* en un arbre binaire dont aucun nœud n'a de fils gauche. (Remarque : cette fonction n'a aucun intérêt pratique). Les rotations dans les arbres binaires sont rappelées dans la figure 1.

1,5 pt
13 min

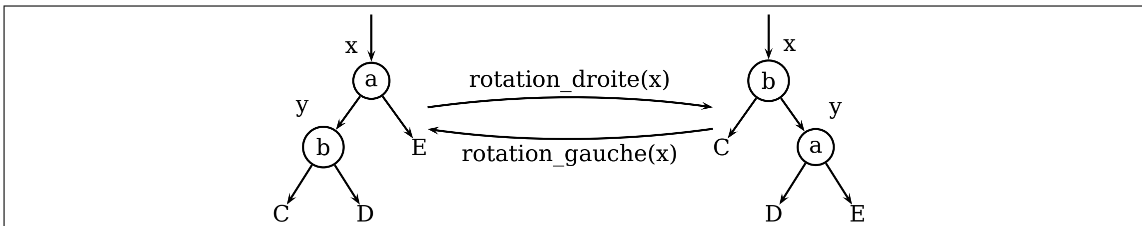


Fig. 1: Rotations gauche et droite. Dans cette version les éléments a et b sont échangés entre les nœuds x et y mais x garde sa place dans l'arbre.

Problème du rang (complément du devoir)

Dans cette partie on s'intéresse au problème suivant : étant donné un tableau T de N éléments deux à deux comparables, déterminer quel est l'élément de rang k (le k-ième plus petit élément), c'est à dire l'élément x de T tel que exactement $k-1$ éléments de T sont plus petits que x. Bien entendu, on peut supposer que k est choisi tel que $1 \leq k \leq N$. On pourra considérer que les éléments de T sont distincts (la comparaison ne les déclare jamais égaux). Si on a par exemple les éléments 23, 62, 67, 56, 34, 90, 17 (N vaut 7) alors l'élément de rang 3 est 34.

Exercice 9.
Le moyen le plus simple d'écrire une fonction Rang(T, k) résolvant ce problème est de trier le tableau et de renvoyer l'élément d'indice $k-1$ du tableau obtenu (les tableaux commencent à l'indice 0). En choisissant au mieux l'algorithme de tri, quel sera en pire cas le temps d'exécution de cette fonction ?

0.5 pt
4 min

On souhaite évaluer la complexité d'une autre solution à ce problème, dérivée du fonctionnement du tri rapide. Pour calculer Rang(T, k) on utilise l'algorithme suivant :

Fonction Rang(T, k)

$p =$ Partitionner(T);
si $p + 1 = k$ **alors**
  retourner $T[p]$;
si $k < p + 1$ **alors**
  retourner Rang ($T[0 \ldots p-1]$, k);
si $k > p + 1$ **alors**
  retourner Rang ($T[p+1 \ldots$ Taille$(T) - 1]$, $k - p - 1$);

Il s'agit de partitionner le tableau T autour de la valeur x contenue à l'indice 0, appelée *pivot*, après quoi le tableau contient : les éléments plus petits que x (dans le désordre), puis x à un certain indice p, puis les éléments plus grands que x (dans le désordre). La fonction de partitionnement travaille en place et renvoie le nouvel indice p de l'élément qui a servi de pivot (x).

Si l'indice p coincide avec le rang recherché (c'est à dire si $p + 1 = k$) alors c'est terminé et on renvoie le pivot de ce partitionnement $x = T[p]$ qui est bien l'élément de rang k. Sinon, il y a deux cas selon s'il faut chercher à gauche ou à droite de x : si k est plus petit que $p+1$ on cherche l'élément de rang k dans le sous-tableau des éléments plus petits que x, et, si k est plus grand que $p+1$ on cherche dans le sous tableau des éléments plus grands que x l'élément de rang $k-p-1$ (puisque $p+1$ éléments sont déjà plus petits).

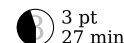
Exercice 10.

On suppose que le partitionnement d'un tableau T replace les éléments plus petits que le pivot dans le même ordre qu'ils étaient avant partitionnement et de même pour les éléments plus grands. En représentant les tableaux et sous-tableaux obtenus successivement, exécuter à la main $\mathsf{Rang}(T, 4)$ où T contient dans cet ordre les éléments : 3, 2, 8, 6, 9, 1, 5.

1 pt
9 min

Exercice 11.

En supposant que partitionner un tableau de n éléments prend un temps n, on étudie le temps d'exécution global (appels récursifs compris) de $\mathsf{Rang}(T, k)$ en fonction de la taille N de T, en notation asymptotique.


3 pt
27 min

1. Quel est le meilleur cas et quel temps obtient-on ? (Donner un exemple générique de tableau et de valeur de k réalisant ce meilleur cas et un équivalent asymptotique du temps d'exécution).
2. Quel est le pire cas et quel temps obtient-on ? (Donner un exemple générique de tableau et de valeur de k réalisant ce pire cas et un équivalent asymptotique du temps d'exécution).
3. Supposons que T est de taille $N = 2^m$ et qu'au cours de la recherche de l'élément de rang 1, chaque partitionnement d'un tableau de taille n donne $\frac{n}{2}$ éléments plus petits que le pivot (c'est à dire que la fonction de partionnement renvoie l'indice $p = \frac{n}{2}$) jusqu'à arriver à la taille 1 dans la récursion. Quel est alors le temps d'exécution en notation asymptotique ? (Ne pas chercher d'exemple de tableau réalisant ce cas).
4. Que penser du temps d'exécution dans le cas plus général où, après chaque partition d'un tableau de taille n, l'appel récursif a lieu sur un tableau de taille $r \times n$ où $r < 1$ est une proportion fixée globalement (à la question précédente r valait $\frac{1}{2}$) ?

Corrigé

Première partie **11 points**

Correction de l'exercice 1.

Rappel des définitions (c'était demandé) :

– $f = O(g)$ ssi
$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq cg(n).$$

– $f = \Omega(g)$ ssi
$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) \leq f(n).$$

– $f = \Theta(g)$ ssi $f = O(g)$ et $f = \Omega(g)$

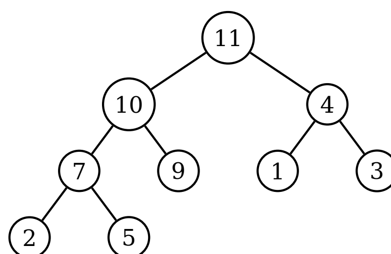
1. Oui $n \log n = O(n^2)$. En effet, pour tout $n \geq 1$, on a $\log n \leq n$ d'où encore $n \log n \leq n^2$. Ainsi, prendre $n_0 = 1$ et $c = 1$ convient.
2. Oui. On a $\log(n!) = \sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n$ pour tout $n \in \mathbb{N}$. Ainsi, prendre $c = 1$ et $n_0 = 0$ convient.
3. Non. Il faudrait que $\log(n!) = O(n^2)$ (ce qui est vrai) et que $\log(n!) = \Omega(n^2)$. Mais on n'a pas $\log(n!) = \Omega(n^2)$. En effet, supposons que ce soit le cas pour un certain n_0 et un certain c et montrons qu'il y a contradiction. À partir du rang n_0 on a $cn^2 \leq \log(n!) = \sum_{i=1}^{n} \log i \leq n \log n$. D'où $cn \leq \log n$ pour tout $n \leq n_0$ avec $c > 0$. Ce qui est impossible puisque $\lim_{n->+\infty} \frac{\log n}{n} = 0$.

Correction de l'exercice 2.

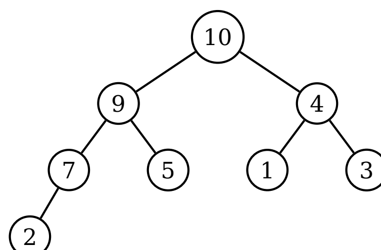
Il reste *pince-moi je rêve ce sujet est trop facile*.

Correction de l'exercice 3.

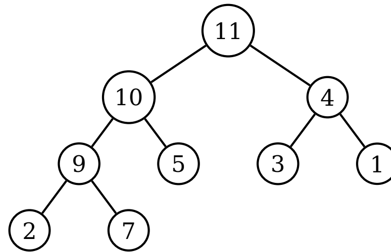
1. Tas obtenu par insertions :



Après suppression de l'élément maximum :

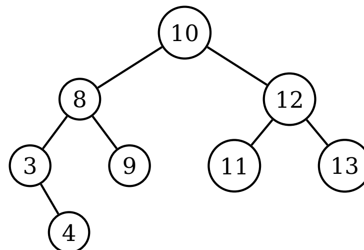


2. Pour former le tas on peut également prendre l'arbre binaire quasi-parfait des éléments du tableau et rétablir la propriété de dominance des tas en appliquant maintienBas successivement sur tous les éléments à partir du dernier élément ayant au moins un fils et en passant à chaque fois au précédent dans l'ordre du tableau.

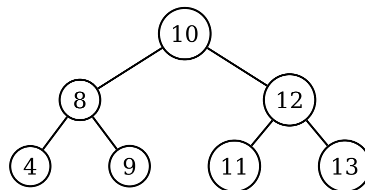


Correction de l’exercice 4.

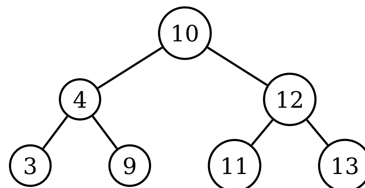
1. L’arbre de recherche obtenu :



2. L’arbre de recherche obtenu :



3. L’arbre de recherche obtenu :



Correction de l’exercice 5.

1. ```
int taille(ab_t x) {
  if (estVide(x)) {
    return 0;
  }
  return taille(gauche(x)) + taille(droite(x)) + 1;
}
```

2. On se donne également la fonction `somme` calculant la somme des éléments de l’arbre :

```
int somme(ab_t x) {
  if (estVide(x)) {
    return 0;
  }
  return somme(gauche(x)) + somme(droite(x)) + x->e;
}
```

Ainsi la moyenne est donnée par

```
double moyenne (ab_t x) {
  return ((double) somme(x)) / ((double) taille(x));
}
```
```

Correction de l'exercice 6.

Par ordre croissant : $\log n, n, n\log n, n^2$.
Exemples de réponses correctes :
Le tri fusion a une complexité en pire cas de $n\log n$.
Le tri rapide a une complexité en pire cas de n^2.

Seconde partie : problèmes 9 points

Correction de l'exercice 7.

1. On cherche i et j différents tels que $t[i]+t[j]=x$. Par symétrie on peut ne s'intéresser qu'à trouver de tels i et j avec $i<j$. Pour chaque indice i allant de 0 à $N-2$, pour chaque j entre $i+1$ et $N-1$ on teste si $t[i]+t[j]=x$, si c'est vrai on s'arrête et on renvoie vrai. Si à la fin de cette double boucle on n'a rien renvoyé on renvoie faux. Le pire cas advient lorsque on ne trouve pas i et j. Ceci va nécessiter $N-1-i$ comparaisons pour chaque i, c'est à dire $\sum_{i=0}^{N-2} N-1-i = \sum_{i=1}^{N-1} N-i = \sum_{k=1}^{N-1} k = \frac{N(N-1)}{2} = \Theta(N^2)$
2. On peut commencer par trier le tableau avec un tri de complexité $N\log N$ en pire cas, par exemple un tri par tas, puis pour chaque indice i (donc N fois), calculer $y=x-t[i]$ et chercher par dichotomie si y est dans le tableau (ailleurs qu'à l'indice i), ce qui coûtera $\log(N)$ en pire cas. Si on trouve y dans le tableau pour un certain i alors on s'arrête car on a deux éléments dont la somme est x sinon il n'existe pas deux éléments différents dont la somme est x. Le temps total d'exécution en pire cas est celui du tri plus N fois celui de la recherche dichotomique ($\log N$) donc en $O(N\log N)$ en pire cas. Remarque : si pour un indice i, la recherche dichotomique de $y=x-t[i]$ renvoie le même indice i alors il faut regarder si $t[i-1]$ ou $t[i+1]$ ne contiennent pas la même valeur y (mais sans précision particulière on pouvait supposer que t contient des valeurs toutes distinctes).

   En voici une version en C, qui tient compte de la dernière remarque :

```
int testsomme(int t[], int taille, int x) {
 int i, j, k;
 int y;
 for (i = 0; i < taille; i++) {
   y = x - t[i];
   j = recherche_dichotomique(t, taille, y); /* indice de y dans t, -1 sinon */
   if (0 <= j) {/* y a été trouvé à l'indice j */
     if ((i != j)                              /* i, j sont déjà distincts */
         || ((0 < i) && (t[i - 1] == y))   /* j = i  mais j = i - 1 convient */
         || ((i + 1 < taille) && (t[i + 1] == y)))  /* ou j = i + 1 convient */
     {
       return TRUE;
     }
   }
 }
 return FALSE;
}
```

Correction de l'exercice 8.

```
void peigner_droite(ab_t x) {
  while (x) {
    while (x->gauche) {
      rotation_droite(x);
    }
```

```
    x = x->droite;
  }
}
```

Problème du rang (complément du devoir)

Correction de l'exercice 9.

Les meilleurs algorithmes de tri, tel le tri fusion et le tri par tas, font en pire $N \log N$ comparaisons. Le fait de renvoyer l'élément d'indice $k-1$ a un coût constant quelle que soit la taille du tableau. Le temps total d'exécution sera donc asymptotiquement $N \log N$ en pire cas.

Correction de l'exercice 10.

Le premier partitionnement se fait autour de 3 qui est l'élément d'indice 0. Le tableau devient : 2, 1, 3, 8, 6, 9, 5 et p vaut 2.

Comme $k = 4 > p + 1 = 3$, il y a un appel récursif sur le sous-tableau 8, 6, 9, 5 avec pour k la valeur $4 - 3 = 1$.

Le partitionnement se fait autour de 8, le tableau devient 6, 5, 8, 9 et p vaut 2.

Comme $k = 1 < p + 1 = 3$, il y a un appel récursif sur le sous-tableau 6, 5 avec $k = 1$.

Le partitionnement se fait autour de 6, le tableau devient 5, 6 et p vaut 1.

Comme $k = 1 < p + 1 = 2$, il y a un appel récursif sur le sous-tableau 5 avec $k = 1$.

Le partitionnement se fait autour de 5, laisse le tableau inchangé et renvoie $p = 0$.

Finalement, comme $k = 1 = p + 1$ la valeur 5 (à l'indice p) est renvoyée par l'algorithme.

Correction de l'exercice 11.

1. Dans tous les cas il faudra procèder au moins à un partitionnement. Le meilleur cas se produit lorsque le pivot est l'élément recherché. Par exemple, si le tableau est déjà trié et que c'est l'élément de rang 1 qui est recherché, après le partitionnement on aura $p = 0$ et le premier élément du tableau sera renvoyé. Ceci prend le temps du partitionnement plus un temps constant (ne dépendant pas de N). Ainsi le meilleur cas est en $\Theta(N)$ où N est la taille du tableau.
2. Chaque partitionnement réduit d'au moins 1 la taille du tableau. Au pire, l'élément n'est pas trouvé avant d'atteindre dans la récusion une taille de tableau égale à 1 et chaque appel se fait sur un tableau dont la taille diminue de 1 par rapport à l'appel précédent. Ce cas peut tout à fait se produire : par exemple si le tableau est trié et qu'on cherche l'élément de rang N (où N est la taille du tableau initial), alors chaque partitionnement laisse inchangé le tableau et élimine de la zone de recherche un élément à chaque étape (le pivot). Le temps d'exécution est alors la somme des temps des partitionnements : $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$ plus N fois un temps ne dépendant pas de N, c'est à dire finalement un temps en $\Theta(N^2)$.
3. Si T est de taille $N = 2^m$, qu'il faut attendre la taille 1 pour terminer et que pour chaque appel récursif la taille du tableau est divisée par deux alors la somme des temps des partitionnement est : $\sum_{i=0}^{m} 2^i = 2^{m+1} - 1 = 2N - 1$ à laquelle il faut ajouter un temps constant autant de fois qu'il y a de termes dans cette somme ($m + 1$ fois, c'est à dire $\log N$). Ainsi le temps d'exécution dans ce cas est $2N - 1 + \log N = \Theta(N)$.
4. En généralisant le raisonnement précédent, sur un tableau de taille N le temps d'exécution des partitionnement sera : $P(N) = N + rN + r^2N + \ldots + r^mN$ plus ou moins 1 où m est le nombre de fois où il faut multiplier N par r pour trouver 1 : $m = \frac{\ln N}{-\ln r}$ (qui est en $\Theta(\log N)$). Pour simplifier on peut supposer que $N = \left(\frac{1}{r}\right)^m$. Ainsi on obtient un temps d'exécution égal à

$$E(N) = N \left(\sum_{i=0}^{m} r^i \right) + c \log N$$

où c est une constante positive, ce qui se simplifie en

$$E(N) = N\left(\frac{1 - r^{m+1}}{1 - r}\right) + c \log N$$

Comme $0 < r < 1$, $\frac{1-r^{m+1}}{1-r}$ tend vers une constante et $E(N)$ est en $\Theta(N)$.