CORRIGÉ DE L'ÉPREUVE D'INFORMATIQUE DE CENTRALE - 2001

Partie I - Algorithmes pour la sélection

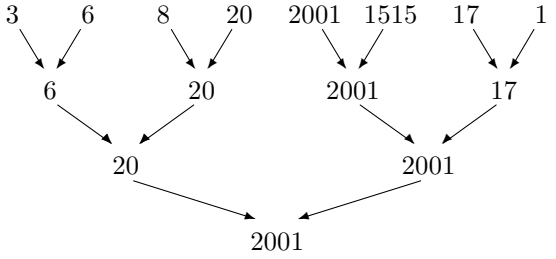
I.A - Recherche des deux plus grands éléments

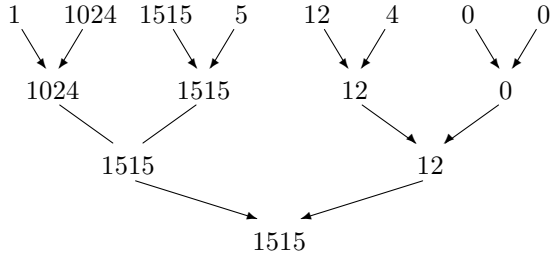
I.A.1) Utilisation d'un arbre pour réaliser un tournoi.

a) Il s'agit de calculer les étiquettes de tous les nœuds de hauteur $h \in [\![0; p-1]\!]$, chacun nécessitant une comparaison.
Il y a donc $1 + 2 + \cdots + 2^{p-1} = 2^p - 1 =$ $\boxed{n-1 \text{ comparaisons.}}$

b) Le maximum m se trouve bien sûr à la racine et son calcul nécessite $n-1$ comparaisons.
Pour trouver le second plus grand élément, on cherche le plus grand des fils autres que m de tous les nœuds étiquetés par m (ce qu'on peut démontrer par récurrence en raisonnant sur la hauteur de l'arbre). Pour cela, on doit donc descendre dans l'arbre jusqu'à une feuille et on effectue $p-1$ comparaisons.
Or $p = \ln_2 n$, donc au total, on a effectué moins de $n + \ln_2 n$ comparaisons.

c) - d) Dans le cas du deuxième tableau, on rajoute des 0 pour avoir un tableau ayant pout taille $2^3 = 8$, ce qui ne modifie pas les deux plus grands éléments, les e_i étant supposés > 0.
En cherchant le plus grand fils autre que le maximum de chaque nœud ayant même étiquette que la racine, on trouve 1515 et 1024 comme deuxième plus grand élément respectif de chaque tableau.





I.A.2) Nombre minimum de comparaisons pour déterminer le maximum.

a) Raisonnons par récurrence sur le nombre p d'arêtes du graphe.
. Si $p = 0$, comme (S, A) est connexe, on a nécessairement $|S| = 1$, donc $|A| \geq |S| - 1$.
. Supposons que tout graphe connexe ayant au plus $p-1$ arêtes a un nombre de sommets inférieur ou égal au nombre d'arêtes plus une.
Soit $G = (S, A)$ un graphe connexe ayant p arêtes. Choisissons une arête a joignant deux sommets i et j et considérons le graphe $G' = \big(S, A \setminus \{a\}\big)$.
. S'il est connexe, alors d'après l'hypothèse de récurrence, $|A'| \geq |S| - 1$, donc $|A| = |A'| + 1 \geq |S| > |S| - 1$.
. Sinon, G' a deux composantes connexes, une (S_1, A_1) contenant i, l'autre (S_2, A_2) contenant j. D'après l'hypothèse de récurrence, $|A_1| \geq |S_1| + 1$ et $|A_2| \geq |S_1| + 1$, d'où $|A'| \geq |S_1| + S|2| - 2 = |S| - 2$ et comme $|A'| = |A| - 1$, on obtient $|A| \geq |S| - 1$.
Autre solution : montrons la propriété par récurrence sur le nombre n de sommets.
. Si $n = 1$, alors $A = \emptyset$ et donc $|A| = 0 \geq |S| - 1$.
. Supposons la propriété vraie pour un graphe ayant $n-1$ sommets.
Appelons degré d'un sommet d'un graphe G le nombre d'arêtes ayant pour extrémité ce sommet.
Lorsque G est connexe, chaque sommet a un degré ≥ 1.
- S'il existe un sommet s de degré 1 et si a est l'arête arrivant en s, alors $G' = \big(S \setminus \{s\}, A \setminus \{a\}\big)$ est connexe et a $n-1$ sommets. D'après l'hypothèse de récurrence, $|A| - 1 \geq (|S| - 1) - 1$, donc $|A| \geq |S| - 1$.
- si tous les sommets ont un degré ≥ 2, alors on a directement $2|A| \geq 2|S|$, donc donc $|A| \geq |S| - 1$.

b) D'après l'hypothèse faite sur e, $\mathcal{A}$ trouve le plus grand élément e_{i_0} avec au plus $n-2$ comparaisons, ce qui signifie que le graphe $\big([\![1; n]\!], C\big)$ a au plus $n-2$ arêtes, donc il n'est pas connexe d'après le a).
Cependant e_{i_0} a pu être comparé avec tous les autres pour savoir que c'est le plus grand. Donc tous sommet e_i autre que e_{i_0} est relié e_{i_0} par un chemin. Il en résulte que deux sommets distincts e_i et e_j peuvent être reliés par un chemin passant par e_{i_0}, ce qui contredit la non-connexité du graphe.

c) Tout algorithme prenant en entrée n entiers et retournant le plus grand de ces entiers exécute au minimum $n-1$ comparaisons (ce qui semble être une évidence a priori).

I.B - Recherche systématique dans un tableau

I.B.1) La recherche du minimum nécessite $n-1$ comparaisons, celle du second plus grand élément en nécessite $n-2$, etc.., celle du i-ème en nécessite $n-i$.

Le nombre $C(i)$ de comparaisons effectuées est donc égal à $\displaystyle\sum_{k=1}^{i}(n-k) = n\,i - \frac{i\,(i+1)}{2} = \frac{2\,n\,i - i^2 - i}{2}$.

$\displaystyle\frac{dC(i)}{di} = n - i - \frac{1}{2}$, donc $C(i)$ est une fonction croissante de i pour $i \leq n/2$.

Si n est pair, le maximum est obtenu pour $i = n/2$ et vaut $\displaystyle\frac{3n^2-2n}{8}$.

Si n est impair, le maximum est obtenu pour $i = (n-1)/2$ et vaut $\displaystyle\frac{3n^2-4n+1}{8}$.

Remarque : si $i > n/2$, alors il est moins coûteux de chercher le $n-i+1$-ième plus grand élément.

I.B.2) Si on effectue un tri dichotomique du tableau T, dont le nombre de comparaisons est en $O(n\,\ln n)$, alors on obtient immédiatement le i-ème élément pour i quelconque dans la i-ème position du tableau T.

I.C - Tri rapide dans une liste chaînée

I.C.1) Fonction `partition`

```
let partition liste pivot =
   let rec aux l avant apres nb = match l with
        |  []  -> (avant,apres,nb)
        | t::q -> if t <= pivot then aux q (t::avant) apres (nb+1)
                               else aux q avant (t::apres) nb
   in aux liste [] [] 0 ;;
```

I.C.2) Fonction `recherche`

```
let rec recherche i liste =
   if liste = [] then failwith "Erreur"
   else
      let pivot = hd liste in
      let (avant,apres,p) = partition (tl liste) pivot in
         if p = i - 1 then pivot
                      else if p < i then recherche (i-p-1) apres
                                    else recherche i avant ;;
```

b) La terminaison de cette fonction est immédiate car les tailles des listes des appels successifs diminuent strictement.
Après l'appel de la fonction `partition`, le `pivot` (qui était le premier élément de la liste) a exactement p éléments qui lui sont inférieurs, donc a pour rang $p+1$.
. Si $i = p+1$, alors le i-ième plus petit élément de la liste initiale est égal au `pivot`.
. Si $p < i-1$, alors i-ième plus petit élément de la liste initiale est égal au $i-(p+1)$-ème élément situé dans la liste `apres` (on supprime les p éléments de la liste `avant` ainsi que le `pivot`).
. Si $p > i$, alors i-ième plus petit élément de la liste initiale est égal au i-ème élément situé dans la liste `avant`.

c) Le meilleur des cas est celui où le i-ème plus grand élément est placé au départ en tête de la liste : il n'y a alors qu'un seul appel de la fonction `partition` qui utilise $n-1$ comparaisons.
Les cas les plus coûteux sont ceux pour lesquels les tailles successives des sous-listes ne diminuent que d'une unité à la fois, c'est à dire lorsque qu'après chaque appel de la fonction `partition`, l'une des sous-listes `avant` ou `après` est vide, ce qui se produit en particulier si la liste initiale est déjà triée dans l'ordre croissant ou dans l'ordre décroissant.
Si par exemple $i = 1$ et la liste est triée dans l'ordre décroissant, alors chaque liste `apres` est vide et les tailles succssives des listes `avant` sont $n-1,\ n-2, \ldots, 1, 0$, ce qui conduit à $\displaystyle\sum_{k=1}^{n-1} k = \frac{n\,(n-1)}{2}$ comparaisons.

d) Si la liste est formée de tous les entiers d'un intervalle $[\![p;q]\!]$, on peut prendre comme pivot la partie entière de la moyenne de tous les éléments de la liste.

La fonction partition renvoie alors des listes `avant` et `après` elles-mêmes uniformes.
Le coût du calcul de cette moyenne est de même ordre de grandeur que celui de la fonction `partition`.
Ainsi chaque appel récursif s'applique sur une liste dont la taille est diminuée de moitié.
Mais cela ne fournit pas une amélioration dans tous les cas, seulement en moyenne vraisemblablement.

I-D - Recherche de l'élément médian

I.D.1) En utilisant une fonction de fusion.

a) On détermine d'abord les m premiers éléments du tableau fusionné qui à la sortie de la boucle sont (en désordre) : $T_0, \ldots, T_{i-1}, T_m, \ldots, T_{j-1}$. L'élément médian μ est alors le suivant, donc le minimum de t_i et t_j.

```
let median t =
   let m = vect_length t / 2 in
   let  i = ref 0 and j = ref m  in
     while  !i + !j < 2 * m  do
          if t.(!i) < t.(!j) then i := !i + 1 else j := !j + 1
     done;
   min t.(!i) t.(!j) ;;
```

b) Le nombre de comparaisons est égal à $\underline{m+1}$.

I.D.2) Recherche dichotomique.
Cherchons le médian μ des éléments de T dont l'indice $i \in [\![g_1; d_1]\!] \cup [\![g_2; d_2]\!]$ avec $d_1 - g_1 = d_2 - d_2 = 2^k - 1$, g_1 et g_2 pairs, les deux sous-tableaux $T[g_1..d_1]$ et $T[g_2..d_2]$ étant supposés rangés dans l'ordre croissant.
On pose $m = 2^{k-1}$. Ainsi la partie de T dont on cherche l'élément médian a $4m$ éléments, le médian étant le $(2m+1)$-ième plus petit.
On remaque d'abord que si $g_1 = d_1$ (et $g_2 = d_2$), alors μ est le plus grand des deux éléments T_{g_1} et T_{g_2}.
On pose $m_1 = \lfloor (g_1 + d_1)/2 \rfloor$, $m_2 = \lfloor (g_2 + d_2)/2 \rfloor$, $x = T_{m_1+1}$ et $y = T_{m_2+1}$.
Ainsi les 4 intervales $[\![g_1; m_1]\!]$, $[\![m_1+1; d_1]\!]$, $[\![g_2; m_2]\!]$, $[\![m_2+1; d_2]\!]$ ont tous le même nombre d'éléments égal à m.

• $\underline{\text{Si } x < y}$, alors :

∗ les m éléments d'indice $i \in [\![d_1, m_1]\!]$ sont tous $< x < y$.
les m éléments d'indice $i \in [\![d_2, m_2]\!]$ sont tous $< y$.
Il y a donc au moins $2m+1$ éléments inférieurs à y.
Or l'élément médian μ a exactement $2m$ éléments qui lui sont inférieurs, donc $\underline{\mu < y}$.

∗ les m éléments d'indice $i \in [\![m_1+1; d_1]\!]$ sont tous $\geq x$.
les m éléments d'indice $i \in [\![m_2+1; d_2]\!]$ sont tous $\geq y > x$.
Il y a donc au moins $2m$ éléments supérieurs ou égaux à x.
Or l'élément médian μ a exactement $2m$ éléments qui lui sont supérieur ou égal, donc $\underline{x \leq \mu}$.

Ainsi, si on supprime les éléments d'indice $i \in [\![g_1; m_1]\!] \cup [\![m_2+1; d_2]\!]$, alors on supprime autant d'éléments situés avant μ que d'éléments situés après μ. Donc μ est encore égal au médian des éléments d'indices $i \in [\![m_1+1; d_1]\!] \cup [\![g_2; m_2]\!]$.

• $\underline{\text{Si } x > y}$, alors, de même μ est le médian des éléments d'indice $i \in [\![g_1; m_1]\!] \cup [\![m_2+1; d_2]\!]$.

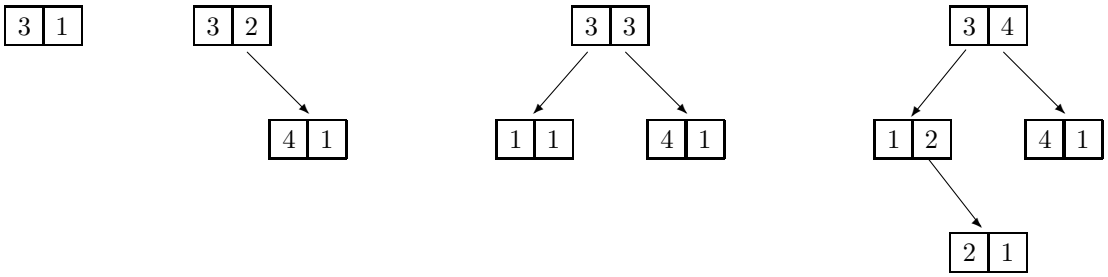
```
let median t =
   let n = vect_length t in
   let m = n / 2 in
   let rec recherche_dicho g1 d1 g2 d2 =
     if g1=d1 then max t.(g1) t.(g2)
     else
         let m1 = (g1+d1)/2 and m2 = (g2+d2)/2 in
         let x=t.(m1+1) and y=t.(m2+1) in
           if x < y then recherche_dicho (m1+1) d1 g2 m2
                    else recherche_dicho g1 m1 (m2+1) d2
   in recherche_dicho 0 (m-1) m (n-1) ;;
```

c) La validité du programme précédent a été prouvée au a).

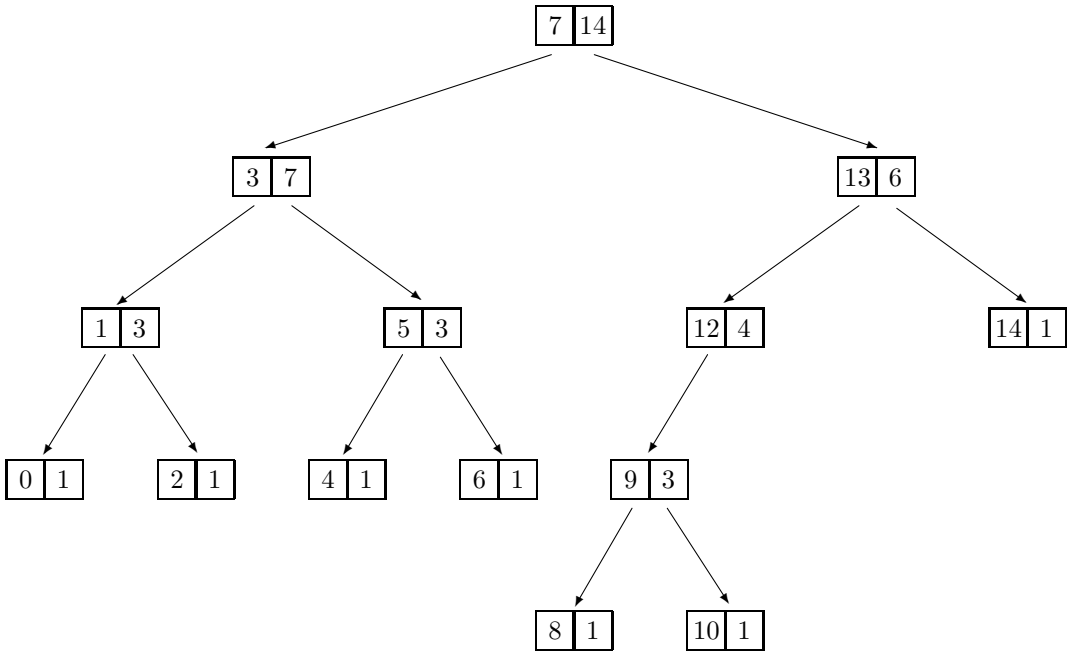
d) Chaque étape coûte une comparaison et diminue la taille par 2. Il y a donc $\underline{p = \ln n \text{ comparaisons}}$.

I-E - Utilisation d'arbres binaires de recherche

I-E.1) On obtient successivement :



b) L'élément 1 figure deux fois, ce qui contredit l'énoncé qui annonce des listes d'éléments distincts. Je ne tiendrais donc pas compte du second 1 de la liste.



I.E.2) La fonction d'insertion suivante suppose que l'élément à insérer n'est pas déjà dans l'arbre.

```
let rec insere x a = match a with
     |    vide -> Noeud {valeur = x; taille = 1; gauche = vide; droit = vide}
     | Noeud e -> e.taille <- e.taille + 1;
                  if x < e.valeur then e.gauche <- insere x e.gauche
                                  else e.droit  <- insere x e.droit;
                  a ;;
```

I.E.3)

a) Si l'arbre est bien équilibré au sens où toutes les feuilles sont à la hauteur h ou $h-1$, alors $h = \ln_2 n$ (où n désigne la taille de l'arbre) et la complexité de la fonction `insere` est alors en $O(h)$, donc en $O(\ln_2 n)$.

b) Si l'arbre est un "peigne", alors sa hauteur est égale à $n-1$ et la complexité de la fonction `insere` est alors en $O(n)$.

Ce cas se produit lorsqu'on construit l'arbre en insérant des entiers successifs rangés dans l'ordre croissant (peigne à droite) ou dans l'ordre décroissant (peigne à gauche).

I.E.4) La fonction de recherche dans l'arbre utilise le même principe que celle du I.C.2), la racine jouant le rôle de pivot.

```
let taille_arbre a = match a with
             |   vide  -> 0
             | Noeud b -> b.taille ;;

  let rec recherche i a = match a with
        |   vide  -> failwith "Erreur"
        | Noeud e -> let g = taille_arbre e.gauche in
                     if g = i - 1 then e.valeur
                     else if g >= i
                          then recherche i e.gauche
                          else recherche (i-g-1) e.droit ;;
```