CARRIERE Sébastien
MÉRÉ Aurélien

Projet d'algorithmique

Sommaire

Première partie

Introduction

Le but de ce projet est de réaliser un algorithme de recherche des deux plus grands éléments d'une liste. Nous étudierons dans ce dossier d'une part un algorithme de recherche séquentielle, et un algorithme de tournoi.

Algorithme de recherche séquentielle

Introduction

L'algorithme de recherche séquentielle balaye séquentiellement la liste de n éléments jusqu'à trouver les deux plus grands éléments. Deux balayages sont effectués : le premier pour détecter le maximum de la liste, et le deuxième pour déterminer le second maximum. Avant le deuxième balayage, on aura pris sans de retirer de la liste le maximum précédemment trouvé afin d'éviter sa réplication.

Tous les algorithmes présentés ci-après supposent que la liste contient au moins 2 éléments.

Nous présenterons cet algorithme avec une liste de type tableau. Nous aurions également pu utiliser une liste chaînée : il n'y a strictement aucun changement, si ce n'est dans la méthode de balayage. Toutefois, la complexité reste inchangée si on se base sur le nombre de comparaisons.

Algorithme

```
Procedure Recherche2Max_Seq (tableau t(1..n) entiers)
/* Recherche séquentiellement les deux maximum du tableau t */

/* max1 et max2 contiennent les maximum
   i est un index de boucle
   pmax est l'indice de la valeur maximum du tableau */

Entier max1, max2, i, pmax;

  /* Premier balayage */

i=2;
pmax=1;

Tant Que (i<n)
     Si (t[i] > t[pmax]) Alors
          pmax=i;
     Fin Si
Fin Tant Que

/* Deuxième balayage */

  max1=t[pmax];
t[pmax]=t[1];

max2=t[2];
i=3;
Tant Que (i<n) faire
     Si (t[i] > max2) alors
          max2 = t[i];
     Fin Si
Fin Tant Que

t[pmax]=max1;

Fin
```

Analyse de complexité

On prendra comme opération fondamentale la comparaison entre éléments dans notre analyse. Comme nous pouvons le voir dans le programme, il n'y a que deux instructions de comparaisons, chacune dans l'enceinte d'une boucle.

La première : « Si (t[i] > t[pmax]) Alors » est réalisée pour n variant de 2 à n.
La seconde : « Si (t[i] > max2) alors » est réalisée pour n variant de 3 à n.

En conséquence, la complexité théorique sera :

$$\sum_{i=2}^{n} 1 + \sum_{i=3}^{n} 1 = (n-1)+(n-2) = 2n-3$$

Jeux d'essais

Examinons le jeu d'essai suivant (tableau de 10 éléments) :

10	4	8	12	2	6	3	18	16	1

Avant le premier balayage, les variables sont initialisées aux valeur suivantes :

10	4	8	12	2	6	3	18	16	1
Pmax=1	i=2								

On compare Pmax à tous éléments du tableau entre i et n, soit 9 comparaisons (On a surligné les éléments qui ont été comparés). Après le premier balayage, les variables contiennent les valeurs suivantes :

10	4	8	12	2	6	3	18	16	1
							Pmax=8 Max1=18		

Avant le deuxième balayage, le tableau a été réinitialisé, de même que la variable i :

10	4	8	12	2	6	3	10	16	1
	Max2=4	I=3					T[10]=T[1]		

On compare Pmax à tous éléments du tableau entre i et n, soit 8 comparaisons (On a surligné les éléments qui ont été comparés). Après le second balayage, les variables contiennent les valeurs suivantes :

10	4	8	12	2	6	3	10	16	1
								Max2=16	

Après le second balayage, les valeurs initiales du tableau sont rétablies. Le nombre de comparaisons totales est donc de 9+8=17 comparaisons. Comme nous disposions de 10 éléments dans notre tableau, nous trouvons donc bien pratiquement 2n-3=17 comparaisons.

Autres essais pratiques et avantages

Pratiquement, sur un tableau de 1 million d'éléments de type entiers longs choisis au hasard, on trouve 1999997 comparaisons, ce qui correspond une fois de plus au résultat théorique. L'avantage de cet algorithme est qu'il n'occupe aucun espace mémoire supplémentaire de celui alloué pour la liste. Il faut toutefois que celle-ci soit en lecture/écriture pour fonctionner correctement. Pour pallier également à son manque de rapidité, nous allons voir le tri tournoi.

Algorithme de tournoi

Introduction

Une deuxième solution proposée pour résoudre le problème est d'utiliser un tournoi. On procède en plusieurs tours : Au premier tour, on compare tous les éléments deux à deux (comme dans un match où les joueurs sont les éléments à comparer), et on ne garde que le plus grand des deux pour le tour suivant (où il n'y aura plus que n/2 joueurs). Dans le cas éventuel où il y ait un nombre impair de joueurs à un tour, on déclare le dernier joueur vainqueur et il reste en lice pour le tour suivant. Au deuxième tour, on recommence, on fait jouer les joueurs deux par deux, et ainsi de suite. Au dernier tour, il ne reste plus que le vainqueur, à savoir le plus grand élément.

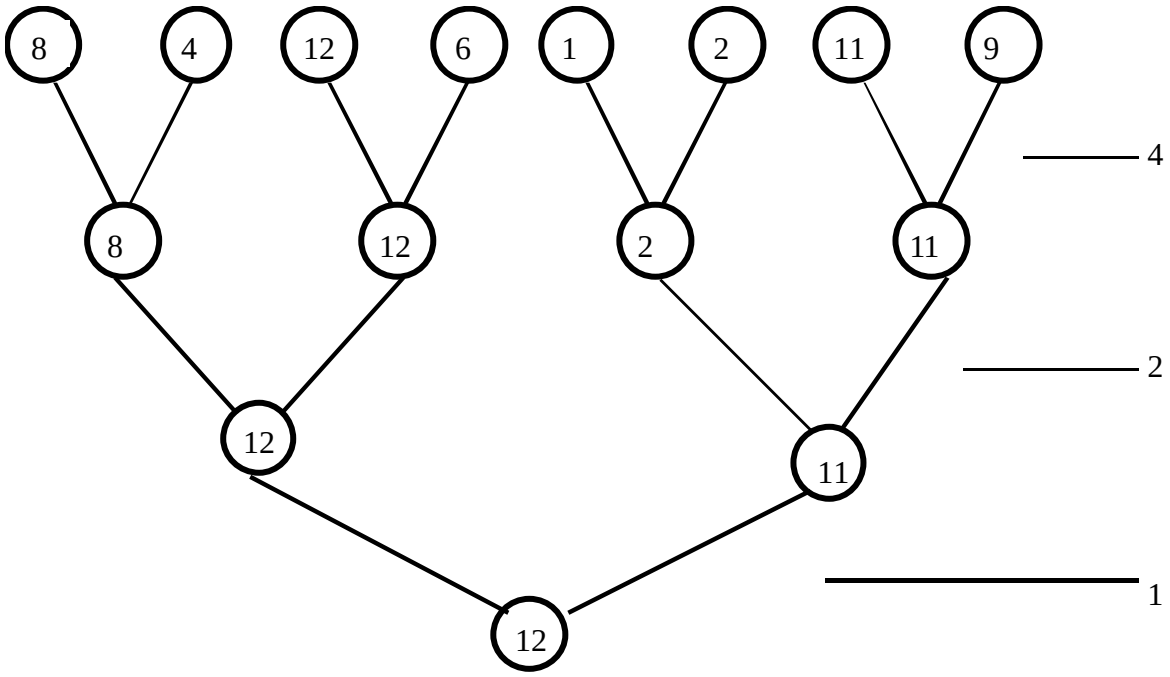
Réflexions préliminaires

Supposons que pour un tour t, il y ait n éléments. Si n est pair, il y aura (n div 2) éléments au tour suivant. Si n est impair, il y aura (n div 2)+1 éléments au tour suivant.

Pour un nombre n d'éléments au départ de l'algorithme, on aura $\lceil \log_2(n) \rceil$ tours.

Le deuxième plus grand élément a été obligatoirement « battu » par le maximum. En conséquence, celui-ci fera partie de la liste des vaincus immédiats du maximum.

Principe pratique



Le nombre de comparaisons réalisées pratiquement à chaque tour est indiqué à droite. Comme nous pouvons le voir, le maximum est bien sorti en dernier. Le second plus grand élément a été battu par le maximum à l'avant-dernier tour, ce qui implique qu'il se trouve immédiatement dans la liste des vaincus. Toutefois, il nous faudra parcourir l'ensemble de la liste des vaincus pour s'assurer que le second plus grand nombre n'a pas été éliminé plus tôt (par exemple entre la comparaison au premier tour entre max1 et max2).

Algorithme

Pour cet algorithme, on exploitera une liste chaînée, celle-ci permettant d'enregistrer aisément un pointeur vers l'élément suivant, vers la liste des vaincus de ce propre élément et sa valeur propre. La liste est donc enregistrée de cette manière :

```
Type Liste ;
Type Cellule = Enreg(val : élément, vaincus : liste, suivant : liste) ;
Type Liste = Pointeur(Cellule) ;
```

On considérera également que la liste a été préalablement remplie par un minimum de deux éléments, sans quoi la recherche va échouer.

```
Procedure Recherche2Max_Tournoi(liste tete)
Entiers Max1, Max2 ;

Liste P, P1, tmp ;
Liste prec_tete := Nouvelle cellule (0, NULL, tete) ;

/* On parcourt jusqu'à n'avoir plus qu'un seul élément */
Tant Que (tete^.suivant <> NULL) faire

     /* On parcourt jusqu'à avoir traité tous les éléments */
        P := prec_tete ;
     Tant Que (P <> NULL)
              P1 := P^.suivant ;
          Si (P1^.suivant <> NULL)
               P2 := P1^.suivant ;
               /* TMP est la cellule vaincue */
               Si (P1^.val > P2^.val)
                    TMP := P2 ;
                    P1^.suivant :=P2^.suivant ;
                    TMP^.suivant :=P1^.vaincus ;
                    P1^.vaincus := TMP ;
               Sinon
                    P^.suivant := P1^.suivant ;
                    TMP := P1 ;
                    TMP^.suivant := P2^.vaincus ;
                    P2^.vaincus := TMP ;
               Fin Si
          Fin Si
          P :=P^.suivant ;

     Fin Tant Que
Fin Tant Que
Max1 := tete^.val ;

Max2 := tete^.vaincus^.val;
 Tete^.vaincus = tete^.vaincus^.suivant

 /* On recherche le deuxième maximum dans la liste des vaincus */
 Tant Que (tete^.vaincus <> NULL)
     Si (tete^.vaincus^.val > Max2) alors Max2 = tete^.vaincus^.val ;
     Tete^.vaincus = tete^.vaincus^.suivant ;
 Fin Tant Que
Fin
```

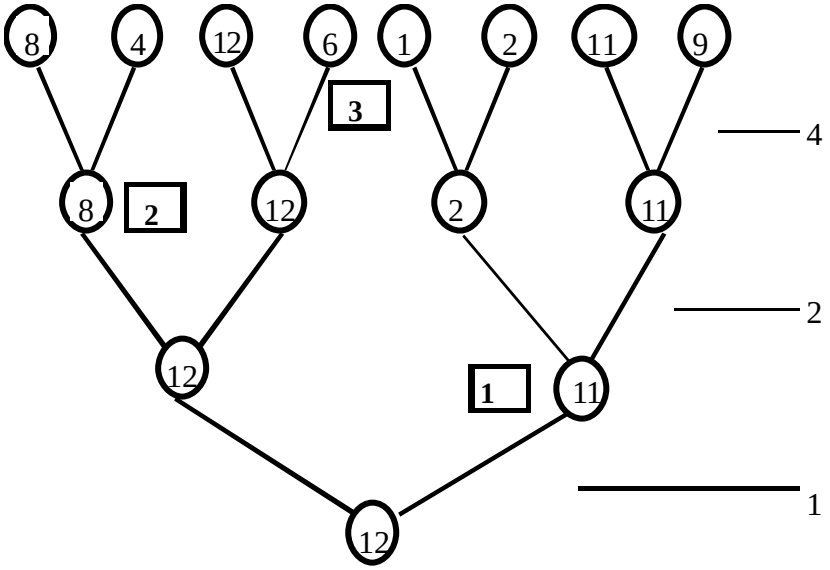
Complexité

Comme nous l'avons vu sur l'arbre précédent, la complexité en temps, sur le nombre de comparaisons, de la recherche du maximum est de n-1, ceci étant vérifié dans tous les cas. La recherche du second plus grand élément est également constante, du fait qu'il faille parcourir toute la liste des vaincus afin de trouver le plus grand parmi ceux-ci. Nous avons vu précédemment que le nombre de tours était égal à $\lceil \log_2(n) \rceil$ pour n éléments.

Le maximum aura donc vaincu autant de nombres que de nombre de tours. Il va donc falloir effectuer une recherche parmi les $\lfloor \log_2(n) \rfloor$ éléments. La complexité finale est donc de (n-1) + $\lfloor \log_2(n) \rfloor$. Le cas le plus favorable se présente pour $n = 2^p$ et le cas le plus défavorable lorsque n n'est pas une puissance de 2.

Nous aurons par exemple pour n=16 éléments une complexité C=(16-1)+log2(16)=15+4=19. En revanche, pour n=17 éléments, nous aurons une complexité C=(17-1)+log2(17)=16+5=21.

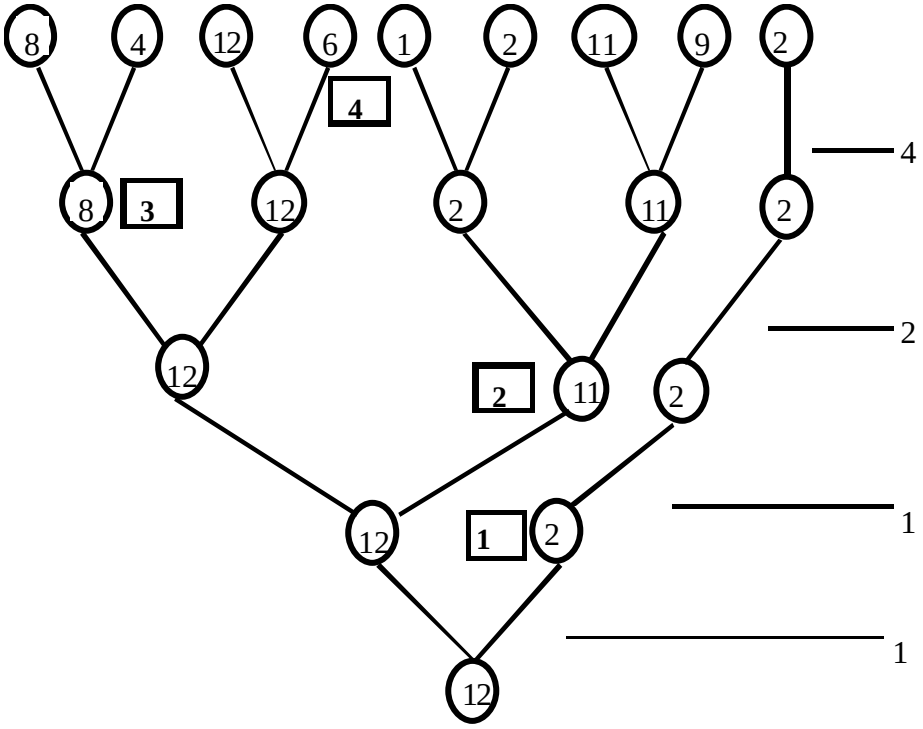
De même, nous pouvons voir sur les arbres suivants les rapports entre les complexités théoriques pour n=8 éléments (C=10) et n=9 éléments (C=12) :



Nous pouvons examiner à droite le nombre de comparaisons pour la première recherche, ici 4+2+1 = 7 comparaisons.

Dans les carrés, on trouve également les lieux où s'est ajoutée une valeur vaincue par le maximum 12, et donc un endroit où s'effectue une recherche dans la boucle finale. Il y en a 3.

La complexité finale est donc de 7+3 = 10, conformément au modèle théorique.



Dans le cas où nous avons 9 éléments, nous voyons que le neuvième gagne à chaque fois son tour sans comparaison, sauf lors du dernier tournoi où il affronte 12. Il y a une comparaison de plus, soit 4+2+1+1=8 comparaisons pour la recherche initiale. Enfin, il y a un vaincu supplémentaire qui s'ajoute à la liste, et qu'il faut donc comparer. On a donc une complexité de 8+4 = 12 comparaisons, conforme au modèle théorique.

Conclusion

Comme nous avons pu le voir au cours des différentes expérimentations, le nombre de comparaisons du tri tournoi est au mieux deux fois inférieur au tri séquentiel. Toutefois, le second demande beaucoup plus d'affectations et de manipulations de données, et une analyse de complexité sur le nombre d'affectations pourrait révéler que le second est beaucoup plus gourmand en temps d'exécution total. C'est d'ailleurs ce qui a été remarqué sur les premières implémentations en langage C. Toutefois, les programmes n'étant pas optimisés, on ne peut considérés comme valides ces derniers résultats. Cela nous ouvre toutefois à un grand nombre d'études différentes.

Deuxième partie

Introduction

Le but de ce projet est de réaliser un algorithme de tri d'une liste d'éléments. Nous étudierons dans cette partie le principe 2 de l'algorithme du tri tournoi. Celui-ci s'effectuera donc de bas en haut.

Principe du tri tournoi

Le principe du tri tournoi est de comparer deux éléments entre eux. Le fonctionnement se fait en deux étapes : la construction du tournoi puis l'extraction du plus grand élément du tournoi.

La construction du tournoi est elle aussi divisée en deux opérations simples : on compare deux éléments puis on remonte le plus fort à l'étage supérieur, et sera remplacée par le plus grand de ses fils. Si l'on se situe sur une feuille, on définit une valeur inférieure à toutes les valeurs de la liste, que nous nommerons **élément_terminal** par la suite.

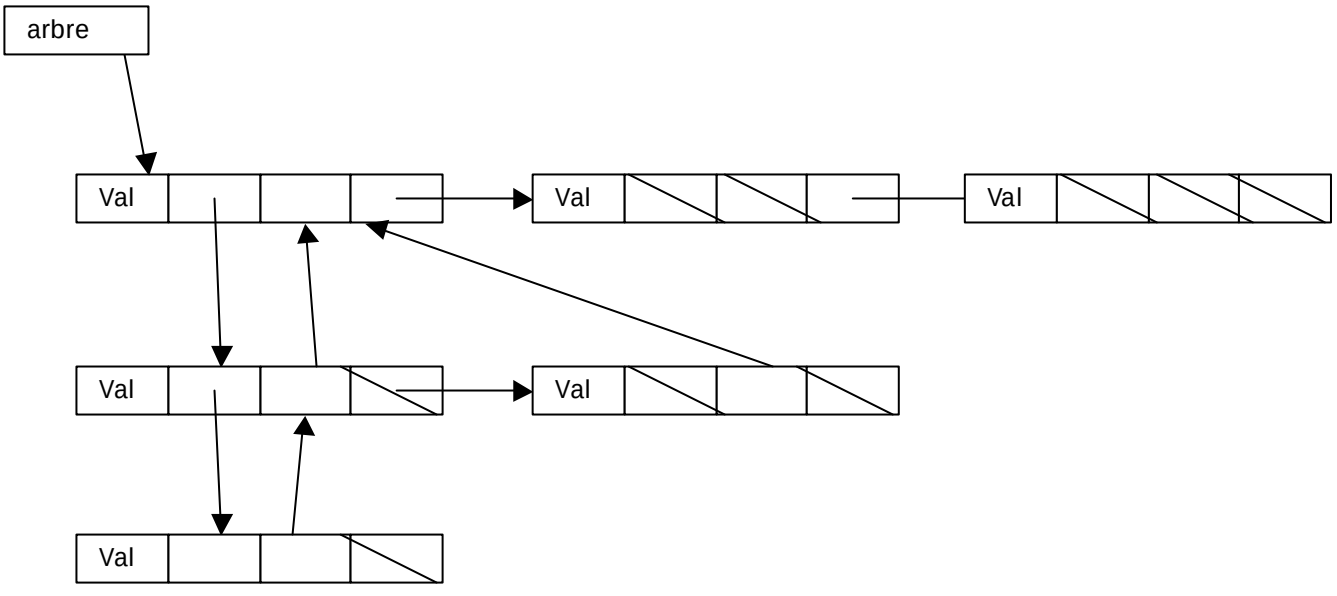
Le tournoi est effectué par tours. Au premier tour, on compare tous les éléments deux à deux (comme dans un match où les joueurs sont les éléments à comparer), et on ne garde que le plus grand des deux pour le tour suivant (où il n'y aura plus que n/2 joueurs). Dans le cas éventuel où il y ait un nombre impair de joueurs à un tour, on déclare le dernier joueur vainqueur et il reste en lice pour le tour suivant. Au deuxième tour, on recommence, on fait jouer les joueurs deux par deux, et ainsi de suite. Au dernier tour, il ne reste plus que le vainqueur, à savoir le plus grand élément.

La structure de données

Nous utiliserons, pour stocker les données, un arbre de type {père, frère, fils ainé} :

```
Type noeud = Enreg(
                val : element,
                pere : arbre,
                frere : arbre,
                fils : arbre} ;
type arbre = pointeur(noeud)
```

Schéma de la structure de donnée :.



L'algorithme

L'algorithme se décompose en deux parties. La première partie consiste en l'élaboration de l'arbre initial, c'est-à-dire que l'on place les éléments à trier aux feuilles et on les remonte jusqu'à la racine de l'arbre, selon la méthode du tournoi.

La deuxième partie consiste à remonter un par un les éléments à la racine, ceux-ci étant triés automatiquement par l'algorithme.

Dans notre algorithme, nous pouvons accéder à chaque nœud de l'arbre par l'utilisation d'un tableau de pointeurs à deux dimensions (numéro du tour, index du nœud). Ceci nous permet d'accélérer les traitements.

```
/* CREATION DE L'ARBRE INITIAL */

/* n est le nombre de joueurs du tour */
/* nb est le nombre total de joueurs */
/* k est le numéro du tour */
/* i et j sont des index du tableau de pointeurs */
/* t est le tableau de pointeurs */

k:=1 ; n:=nb ;

/* On poursuit jusqu'à atteindre la racine */
Tant que (n>1)
          J :=0 ;

          /* On effectue un tour */
     Pour i de 0 à n-1 par pas de 2faire

          /* On alloue le nœud du père, à savoir celui qui contiendra
          /* la plus grande valeur des deux éléments comparés

          T(k, j) :=nouveau nœud ;
          T(k, j)^.pere := NULL;

          /* On rétablit le lien avec ses fils */
          T(k, j)^.fils := t(k-1, i) ;
          T(k-1, i)^.pere := T(k, j);
          Si (i+1<n) alors T(k-1, i+1)^.pere := T(k, j) ;

          /* On rétablit le lien avec son frère si celui-ci */
          /* a déjà été créé */
          Si (j mod 2 = 0) faire
               T(k, j)^.frere := NULL;
          Sinon
               T(k, j)^.frere := t(k, j-1);
               T(k, j-1)^.frere := t(k, j);
          Fin Si

          /* On remonte la plus grande valeur de ses fils */
          T(k, j)^.val : = remonte(t(k-1, i));

          /* On passe au nœud suivant */
          j=j+1 ;
     Fin Pour

     /* On change de tour */
     n :=j ;
     k :=k+1 ;
Fin Tant Que
/* TRI */
```

```
/* Nous disposons d'un arbre de tournoi
/* et l'on sort tous les éléments triés

/* Si la racine contient l'élément terminal
/* Alors tous les éléments ont été traités

Tant que (t(k,0)^.val  <> element_terminal)

           Affiche T(k,0)  /* Affiche les éléments par ordre décroissant */

     T(k,0)^.val := remonte(t(k,0)^.fils);

Fin Tant Que


/* La fonction « remonte » monte récursivement les plus grands éléments parmi
/* l'arbre donné en paramètre et ses fils.

fonction remonte(arbre a) : element
     element tmp ;

     début
          /* On est aux feuilles : on renvoie l'élément terminal */
          si (a=NULL) alors
               retouner element_terminal ;
          sinon

               si a^.frere <> Null alors

                    /* On compare les deux frères et on remonte le plus fort
                    /* Pour le remplacer, on remonte le plus fort de ses vaincus
                    /* A noter que le traitement s'arrête lorsque l'on atteint
                    /* un élément terminal sur chacun des fils

                    si(a^.val=element_terminal et a^.frere^.val=element_terminal
                         tmp :=element_terminal ;
                    sinon
                         si a^.frere^.val > a^.val alors
                              tmp := a^.frere^.val
                              a^.frere^.val:= remonte(a^.frere^.fils)
                         sinon
                              tmp := a^.val
                              a^.val := remonte(a^.fils)
                         finsi
                    finsi
               sinon

                    /* Il n'y a pas de match car l'élément est seul
                    /* Il est donc vainqueur automatique

                    tmp := a^.val
                    si (a^.val <> element_terminal alors
                         a^.val := remonte(a^.fils)
                    finsi
                    retourner(tmp
               finsi
          finsi
     fin
```

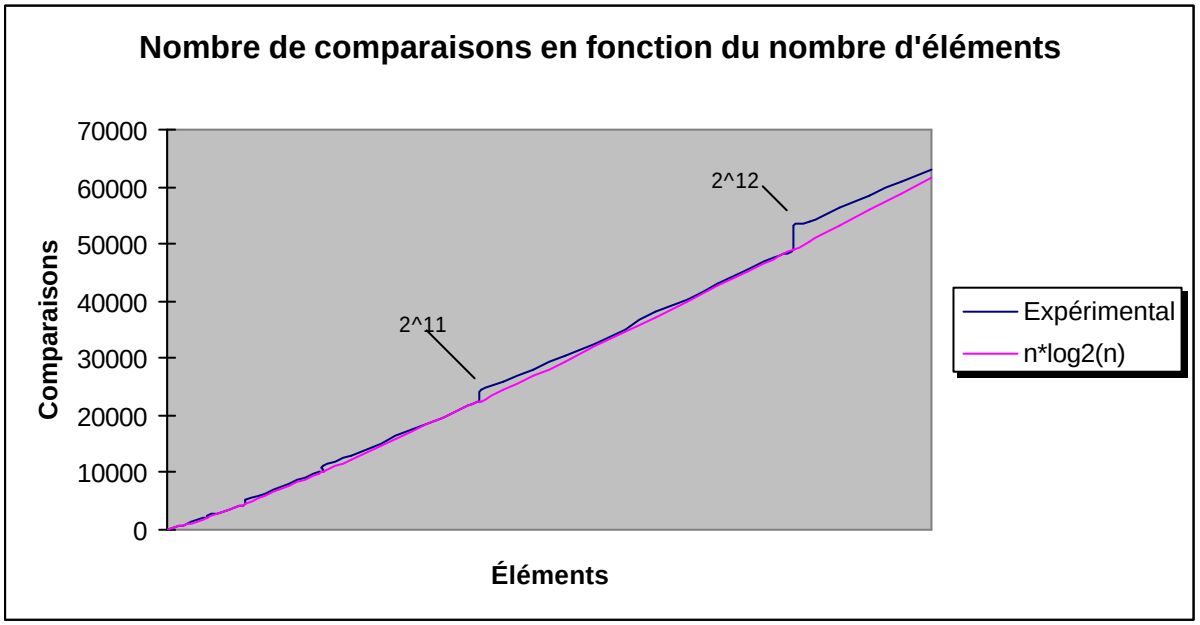
La complexité

Complexité en temps

L'opération fondamentale pour la complexité en temps est la comparaison entre éléments, celle-ci étant réalisée uniquement dans la fonction « remonte » présentée précédemment.
Etant donné la complexité de l'algorithme, nous n'avons pas été en mesure de calculer théoriquement sa complexité en temps. En effet, il est extrêmement difficile de calculer la complexité de la fonction « remonte » car à chaque appel, elle descendra à une profondeur indéterminée dans l'arbre, celle-ci variant avec le tour en cours ainsi que le nombre d'éléments qui ont déjà été traités. Cela fait intervenir des modèles probabilistes que nous ne sommes pas en mesure de déterminer avec nos capacités actuelles. Toutefois, les expériences pratiques sur le programme de test ont montré que :

- La complexité au mieux (dans le cas où n=2^p) est exactement de n*log(n) comparaisons.
- La complexité au pire (dans les cas où n=(2^p)+1) est légèrement supérieur à n*log(n) comparaisons.

D'après nos expériences pratiques, pour des valeurs de n supérieures à 2^10, la complexité en temps est environ 10% plus importante dans le cas pire, que dans le meilleur des cas. Cependant, cette valeur décroît progressivement avec un n qui devient de plus en plus grand.



Nous pouvons remarquer un pic lorsque l'on a un nombre d'éléments égal à n=(2^p)+1, ce qui correspond à la complexité au pire de l'algorithme.

Complexité en espace

L'espace mémoire est occupé par l'allocation dynamique des nœuds de l'arbre au cours de sa création. A chaque tour, nous avons $\lfloor n/2 \rfloor$ éléments.où n est le nombre d'éléments du tour en cours.

Or, nous avons $\lfloor \log_2(n) \rfloor$ tours au total. On en déduit que la complexité en espace est la suivante :

$$C = \sum_{i=0}^{\lfloor \log_2(n) \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor = n + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \ldots + \left\lfloor \frac{n}{2^i} \right\rfloor$$

Dans le cas au mieux, à savoir lorsque n=2^p, ceci est égal à :

$$C?\sum_{i?0}^{p} 2^i ?\frac{1?2^{p?1}}{1?2}?2^{p?1}?1?2n?1$$

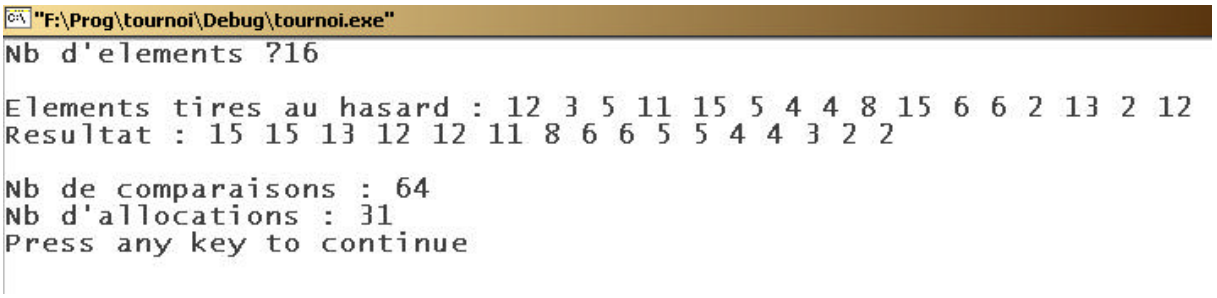
Effectivement, si nous effectuons un tri sur 8 éléments, nous avons 4 éléments au deuxième tour, 2 éléments au troisième tour et enfin le vainqueur, ce qui fait : 8+4+2+1=15 éléments, ce qui est effectivement égal à la valeur théorique de 2n-1 trouvée précédemment.

Dans le cas au pire , il est nécessaire de créer autant de nœuds que la hauteur de l'arbre. Ce cas se produit lorsque n=(2^p)+1. Il est donc nécessaire de rajouter au calcul précédemment réalisé la valeur de la hauteur de l'arbre, à savoir $?\log 2(n)?$.

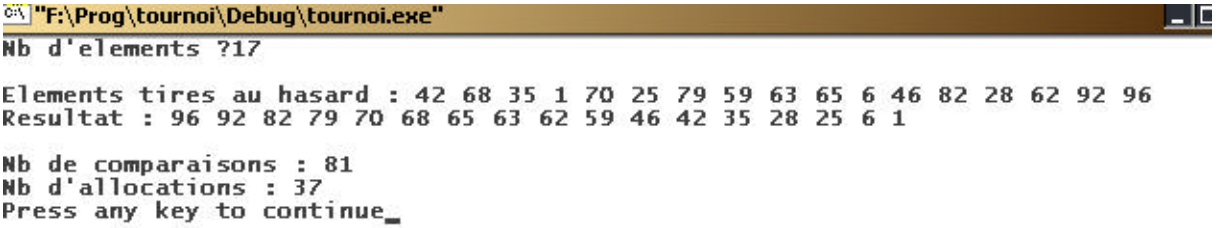
On a ainsi : 2n-1+ $?\log 2(n)?$

Les jeux de tests

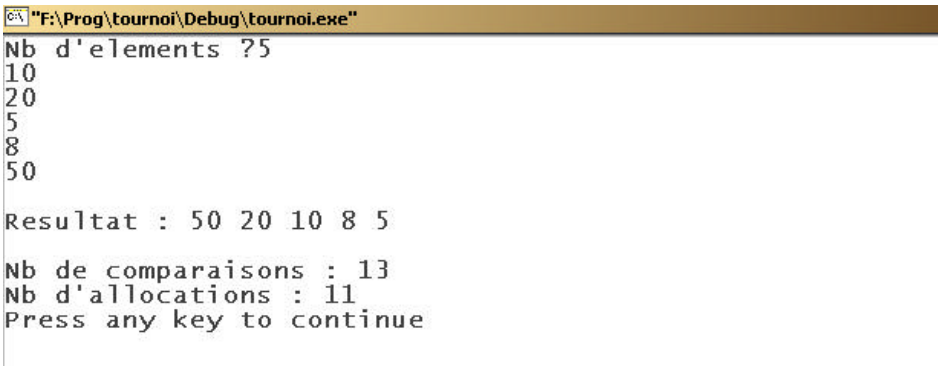
Voici les résultats pour 16 éléments, cas au mieux. Nous pouvons constater que le nombre de comparaisons est bien de n.log2(n)=16*log2(16)=64. Le nombre d'allocations est également bien de 2n-1=2*16-1=31. On pourra également remarquer que l'algorithme gère des nombres entrés en double.

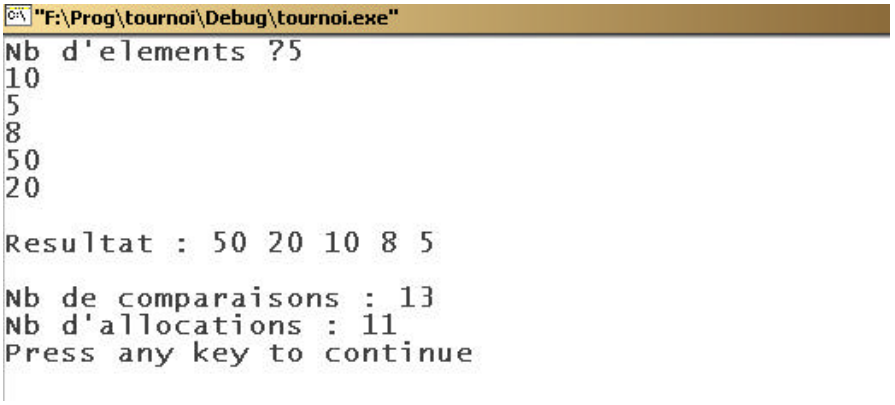


Voici les résultats pour 17 éléments, cas au pire. Le nombre d'allocations est bien vérifié, avec 2n-1+log2(n)=2*17-1+4=37. Le nombre de comparaisons est 26% supérieur à n.log2(n).

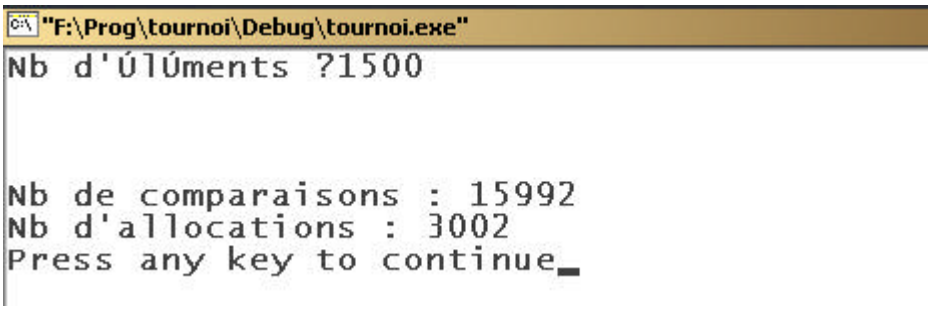


Voici les résultats pour 5 éléments rentrés au clavier (10,20,5,8 et 50), avec deux entrées différentes :

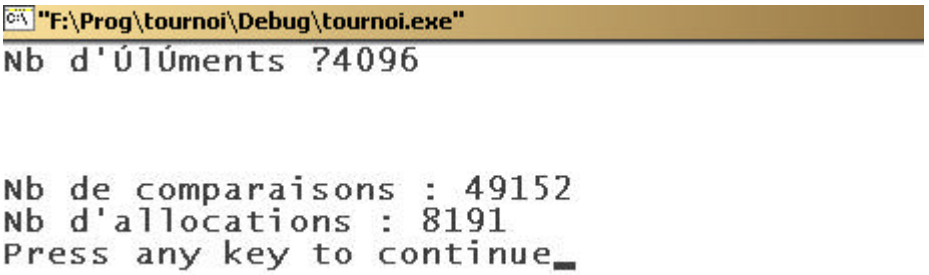




Nous testons ici pour des valeurs plus importantes de n :



Et nous testons ici pour n=2^12. On constate que la mesure expérimentale correspond bien à n*log2(n), à savoir 49152 comparaisons. De même le nombre d'allocations est bien égal à 2n-1=8191.



Comparaison avec le principe 1

Ordre	Nombre d'éléments	Principe 1	Principe 2
2^8	256		2048
	257		2305
	281		2451
2^10	1024		10240
	1025		11265
	1070		11570
2^12	4096		49152
	4098		53252
	4127		53433

Code source du programme final

```
/* tri tournoi

      tournoi.cpp

  (C) Aurélien Méré, 2001

  */
#include <stdio.h>
#include <stdlib.h>

#define element_terminal -1
#define NB_ETAGES 30
#define NB_FEUILLES 5000

typedef int t_element;

typedef struct s_noeud
{
      t_element val;
      struct s_noeud* pere;
      struct s_noeud* frere;
      struct s_noeud* fils;
} t_noeud;

typedef t_noeud* arbre;

int nbComp=0;
int nbAlloc=0;


t_element remonte (arbre a)
{
      t_element tmp;

      if (a == NULL)
      {
            return element_terminal;
      }
      else
      {
            if (a->frere != NULL)
            {
                  if (a->val==element_terminal && a->frere->val ==element_terminal)
                  {
                        tmp = element_terminal;
                  }
```

```
                else
                {
                        nbComp++;

                        if (a->frere->val > a->val)
                        {
                                tmp = a->frere->val;
                                a->frere->val = remonte(a->frere->fils);
                        }
                        else
                        {
                                tmp = a->val;
                                a->val = remonte(a->fils);
                        }
                }

        }
        else
        {
                tmp = a->val;

                if (a->val != element_terminal)
                {
                        a->val = remonte(a->fils);
                }

        }
        return(tmp);
    }
}

void AfficheNiveau(arbre t[NB_ETAGES][NB_FEUILLES], int niveau, int nbelts)
{
    int i;

    for (i=0; i<nbelts; i++)
    {
        printf("%3d ", t[niveau][i]->val );
    }
    printf("\n");
}


void main()
{

    int i, j, k, n, nb, val=0;
    int l,m;

    arbre t[NB_ETAGES][NB_FEUILLES];

    printf("Nb d'éléments ?");
    scanf("%d", &nb);

    for (i=0; i<nb; i++)
    {
        nbAlloc++;
        t[0][i]=(arbre)malloc(sizeof(s_noeud));
        t[0][i]->fils = NULL;
        if (i%2 == 1)
        {
                t[0][i]->frere = t[0][i-1];
                t[0][i-1]->frere = t[0][i];
        }
        else
        {
                t[0][i]->frere = NULL;
        }
```

```
            t[0][i]->pere = NULL;


            /* Valeur au hasard */
            val=rand()%40000+1;

            /* Valeur entrée */
//          scanf("%d", &val);

            t[0][i]->val =val;
      }

      //AfficheNiveau(t, 0, nb);

      /* Création de l'arbre */
      k=1;
      n=nb;
      while (n>1)
      {

            for (i=0, j=0; i<n; i+=2, j++)
            {
                  t[k][j]=(arbre)malloc(sizeof(s_noeud));
                  nbAlloc++;
                  t[k][j]->fils=t[k-1][i];
                  t[k][j]->pere=NULL;
                  if (j%2 == 0)
                  {
                        t[k][j]->frere=NULL;
                  }
                  else
                  {
                        t[k][j]->frere=t[k][j-1];
                        t[k][j-1]->frere=t[k][j];
                  }
                  t[k-1][i]->pere = t[k][j];
                  if (i+1<n) t[k-1][i+1]->pere = t[k][j];
                  t[k][j]->val=remonte(t[k-1][i]);

            }
            //printf("niveau %d -----\n", k);

            /*
            for (l=0, m=nb; l<=k; l++, m=(m/2)+(m%2))
            {

                  AfficheNiveau(t, l, m);
            }
            */

            n=j;
            k++;

      }
      printf("\n");
      /* Affichage */

      k--;
      while (t[k][0]->val  != element_terminal)
      {
//          printf("%d ", t[k][0]->val );

            t[k][0]->val = remonte(t[k][0]->fils);
      }

      printf("\n\nNb de comparaisons : %d\n", nbComp);
      printf("Nb d'allocations : %d\n", nbAlloc);
}
```

Conclusion

L'analyse du graphique du nombre de comparaisons entre éléments nous a permis d'en déduire que la complexité en moyenne de l'algorithme était de l'ordre de n.log2(n), ce qui en fait un algorithme particulièrement performant pour le tri d'un grand nombre d'éléments. Toutefois, il reste peu intéressant pour des petites zones à trier.