

## Algorithmique et graphes : Devoir à la maison n°1

Rappel : Le tri par insertion d'un tableau de n entiers consiste à insérer un à un les éléments d'un tableau dans le sous-tableau trié des éléments déjà insérés.

Exemple :

6	12	1	5	8
Phase 1 : insertion de 12 dans le sous-tableau (première case)				
6	12	1	5	8
Phase 2 : insertion de 1 dans le sous-tableau des deux premières cases :				
1	6	12	5	8
Phase 3 : insertion de 5 dans le sous-tableau des trois premières cases :				
1	5	6	12	8
Phase 4 : insertion de 8 dans le sous-tableau des quatre premières cases :				
1	5	6	8	12

### Exercice 1

Voici quelques exemples de codes glanés dans vos copies qui soit ne fonctionnent pas, soit qui fonctionnent mal.

Merci de commenter ces codes en expliquant ce qui vous semble incorrect. Vous pouvez tâcher de les faire tourner sur des exemples. Vous analyserez également la complexité de ces codes.

#### Question 1 : Pour l'insertion dans un tableau trié :

```
int insertion(int*t,int nbel, int x)
{ int aux ,i ;
for(i=0 ;i<nbel ;i++)
    { if (x<t[i])
        { aux=t[i];
          t[i]=x;
          t[i+1]=aux;
          nbel++;
        }
    }
return (nbel);
}
```

Parmi les éléments à noter:

Boucle for infinie

Ecrasement des valeurs du tableau dans la boucle

Le cas où x est plus grand que tous n'est pas traité (même mal).

La complexité d'un algorithme qui ne termine pas ne peut pas être évaluée.

```
int inserer(int*tab, int n-1, int val)
{ int temp, i,j;
for(i=0;i<n;i++)
{ if (tab[i]>val)
    { for (j=i;j<n;j++)
        { tab[i]=tab[j];
          tab[j]=tab[j+1];
          tab[j+1]=temp;
        }
    }
}
```

```

    }
}
return(n-1);
}

```

Parmi les éléments à noter:

- paramètre n-1: on ne peut avoir une expression en paramètre temp non initialisée
- écrasement des valeurs, à cause de temp et du décalage à gauche au lieu de droite.
- valeur de retour absurde.
- Le cas où l'élément est plus grand que tous n'est pas traité.

Pour la complexité, la boucle sur j, à i fixé, comporte n-i itérations, et effectue un nb constant (5) d'affectations et de comparaisons. Et la boucle externe sur i fait varier i de 0 à n-1 et effectue au pire 2 comparaisons, une affectation et la boucle interne sur j.

Par conséquent, si n est le nb d'éléments du tableau,  $C_{insérer}(n) \leq 1 + 3n + \sum_{i=0}^{n-1} 5(n-i) = 1 + 3n + 5 \sum_{i=1}^n (n-i) = 1 + 3n + 5n(n+1)/2 \leq 5n^2$ . D'où une complexité en  $O(n^2)$

```

int inserer(int*tab, int n, int val)
{ int i,j;
for(i=0;i<n;i++)
    { if (tab[i]>val)
        { for (j=n+1;j>i;j++)
            { tab[j+1]=tab[j];
            tab[i]=val;
            } else t[n+1]=val;
        }
}
return(n+1);
}

```

Parmi les éléments à noter:

boucle interne infinie, probablement à cause d'un j++ au lieu de j--. Mais même dans ce cas, si le décalage est dans le bon sens cette fois, il est réalisé à chaque fois que val est plus petit que la suite du tableau.

else très mal placé (en plus du manque d'accolade).

tab[i] à factoriser : l'instruction n'a pas besoin d'être répétée dans la boucle pour j puisqu'on ne change pas val.

Complexité impossible à calculer à cause de la boucle infinie.

### Question 2 : Pour le tri par insertion :

```

Int triinsertion( int *tab, int nbelement)
{ int i, p ; int val=tab[nbelement+1] ;
for (i=0 ;i<nbelement;i++)
    { if (tab[i]>val)
        { p=tab[i];
        tab[i]=val;
        tab[i+1]=p;
        } else { val=tab[i]
        }
}
triinsertion (tab;nbelement-1);
return 0;
}

```

Parmi les éléments à noter:

Ce code comporte des erreurs de syntaxe mais aussi des erreurs de conception algorithmique, qui se mélangent pour donner plutôt le pire :

Déjà, dans l'aspect tri récursif : normalement, l'insertion est réalisée sur un tableau trié. Il faut donc d'abord trier récursivement puis essayer d'insérer (ce qu'est censé faire la première partie du code).

non terminaison : il n'y a pas de condition d'arrêt, le programme ne peut pas s'arrêter (hormis pour des erreurs de compil).

Val est initialisé avec un élément du tableau qui a priori se trouve hors du champ de l'étude, puisque le tri trie les éléments 0 à nbelement du tableau.

L'instruction val=tab pose un pb de type.

Le prototype de la fonction stipule que celle-ci renvoie un entier, mais l'appel récursif est fait sans en récupérer la valeur dans une variable entière.

l'insertion du début n'en est pas une, il y a écrasement de valeur.

Comme indiqué précédemment, la complexité ne peut être évaluée.

```
void triiteratif(int*t,nb,nbmax)
int i, nb2=0; int *t2[nbmax];
for(i=0; i<nb;i++)
    nb2=insere(*t2,nb2,t[i]);
t=t2;
}
```

Parmi les éléments à noter (en dehors des pb de parenthèses) :

déclaration tableau inexacte

incohérence dans l'appel sur le paramètre \*t2 : si c'est un tableau, il faut l'appeler insere avec t2.

l'instruction t=t2 pose plein de problèmes : elle ne copie pas le tableau t2 dans t, mais fait pointer t sur une zone allouée juste pour l'exécution de triiteratif, qui sera donc perdue après la fin de la fonction.

Cela aura également pour conséquence de perdre les valeurs du tableau t initial.

Pour la complexité, la boucle for comprend n itérations, et à chaque itération on insère dans un tableau dont la taille augmente de 1. On a donc une complexité  $C_{\text{triiteratif}}(n) \leq 1+3n+\sum_{i=0}^{n-1}$

$C_{\text{insere}}(i)$ . Si l'insertion est bien faite, elle est en  $O(n)$ , c'est-à-dire que  $C_{\text{insere}}(i) \leq a.i$ , où a est une constante.

D'où  $C_{\text{triiteratif}}(n) \leq 1+3n+an(n+1)/2 \leq (a+3)n^2$ . D'où une complexité en  $O(n^2)$

```
int tri_iter(int*tab, int n, int elem, nbcases)
{ int i,j,tmp;
for(i=0;i<nbcases;i++)
    for((j=i;j<nbcases;j++)
        { if( tab[j]>elem)
            { tmp=tab[j];
              tab[j]=elem;
              elem=tmp;
            }
        }
return nbcases;
}
```

Parmi les éléments à noter :

paramètre n inutilisé.

Pour un tri, le paramètre elem n'a aucune raison d'être

c'est un code faux pour l'insertion (avec tout ce qu'on a vu avant, écrasement de valeurs, pas de décalages, valeur de retour stupide,etc)

Pour la complexité, on a deux boucles imbriquées, la première a n itérations et la deuxième n-i pour i fixé. On peut donc tenir exactement le même raisonnement que pour le code n°2, avec une complexité identique en  $O(n^2)$ .

## Exercice 2

Ecrire un code récursif et un code itératif permettant de réaliser un tri par insertion d'une liste simplement chaînée d'entiers. Analyser leur complexité.  
Commenter leur efficacité par rapport à l'utilisation d'un tableau.

Je vous propose un corrigé en deux parties : d'abord un code correct, puis un commentaire sur quelques codes incorrects que vous m'avez fournis. Le code suivant va avoir pour effet de réordonner les maillons d'une liste (et pas de créer une copie de la liste initiale).

```
typedef struct maillon{int val ; struct maillon *suiv}maillon, *liste ;
```

```
liste insere( liste l, liste nouv)
{liste p =l;
if ( ( !p) || (p->val>nouv->val ))
    {nouv->suiv=p; return nouv ;}
while ((p->suiv)&&(p->suiv->val < nouv->val))
    { p=p->suiv;}
nouv->suiv=p->suiv ;
p->suiv=nouv ;
return (l) ;
}
```

Analyse de la complexité de l'insertion: la boucle while comporte au plus  $n-1$  iterations, où  $n$  est le nb de maillons dans  $l$ , puisqu'à chaque itération, on avance  $p$  d'un maillon. Le reste des calculs ne dépend pas de la taille de la liste. D'où une complexité en  $O(n)$ .

Au passage, un autre code (récursif celui-ci) pour l'insertion que je vous conseille de bien comprendre :

```
liste insere_rec( liste l, liste nouv)
{
if ( ( !l) || (l->val>nouv->val ))
    {nouv->suiv=l; return nouv ;}
else {l->suiv=insere_rec(l->suiv, nouv) ; return(l) ;}
}
```

tri par insertion récursif :

```
liste tri_rec(liste l)
{liste p=l->suiv;
if( !p) return (l) ;
p=tric_rec(p);
return(insere(p,l));
}
```

tri par insertion iteratif:

```
liste tri_ite(liste l)
{liste elem,q= l, p=NULL;
while (q)
    {elem=q;
q=q->suiv;
p=insere(p,elem);
}
return(p);
}
```

Complexité:  $O(n^2)$ , même raisonnement que pour l'interro (voir corrigé).

Voici maintenant vos codes commentés :

1) extrait des copies d'au moins 8 personnes :

**Version itérative :**

```
void tri_ite(liste *l)
{
    liste l2=NULL;
    liste p=*l;
    while(p)
    {
        l2=insertion(l2,p->val);
        p=p->suiv;
    }
    *l=l2;
}
```

Voici un code qui fonctionne (à condition que la fonction d'insertion soit correctement programmée). Mais en y regardant de plus près il comporte tout de même certains soucis : en effet, la fonction d'insertion a pour rôle d'insérer une valeur dans une liste (et non un maillon dans la liste). Par conséquent, l2 va être, à la fin de la boucle while, une liste de nouveaux maillons, copiés à partir de la liste de départ (\*l). Donc le fait d'écrire \*l=l2 va effectivement permettre à la fonction de pointer sur une liste triée, mais sans désallouer l'espace mémoire qui était dévolu à la liste \*l, qui ne sera plus accessible de nulle part.

**Complexité :**

- 1 comparaisons faites n fois
- 3 affectations faites 1 fois
- 1 rappel d'insertion fait n-k fois.

**D'où  $C(n)=3+3*n*\sum C_{insertion}(n-k) \leq O(n^2)$**

Cinsertion(n) est de l'ordre de  $O(n)$  Donc  $C(n)$  est de l'ordre de  $O(n^2)$

C'est typiquement une expression incomplète. Quand on écrit une somme, il est indispensable d'écrire les bornes de variations de la somme et sur quelle variable elle porte. De plus, le calcul doit alors être un peu développé pour aboutir à l'inégalité.

Tri dans une liste chaîné itératif :

```
Liste liste_triée(liste l)
{
    liste listetriée= null ;
    while(l)
    {
        listetriée=insert(listetriée, l->val) ;
        l= l->val ;
    }
    return listetriée ;
}
```

Ce code ressemble pas mal au précédent, sauf la forme fonction, mais a le même défaut de principe. Il comporte hélas également la grosse erreur l=l->val, qui aurait été pardonnable (étourderie ?) pour une personne fatiguée, mais comme vous vous y êtes mis à 3 au moins sans qu'aucun de vous ne s'en aperçoive...

### Version récursive :

```
void tri_recur(liste *l)
{
    liste p=*l;
    if(p)
    {
        tri_recur(&p->suiv);
        p=insertion(p->suiv,p->val);
    }
    *l=p;
}
```

```
liste tri_liste_recur(liste *l)
{
    if(l)
    {
        trier_liste(&l->suiv);
        l=insertion(l->suiv ;tmp->val);
    }
}
```

```
Void tri_liste(liste *l)
{
    if(l)
    {
        tri_liste(&l->suiv);
        l=insert(l->val, l->suiv);
    }
}
```

```
liste* triInsertionListe(liste *l){
    liste *lTrie;
    if (l->suiv){
        lTrie=inserer(l->suiv, l->val);
        triInsertionListe(l->suiv);
    }
    return lTrie;
}
```

On trouve ici le même genre de problème que dans le code précédent : les maillons sont déconnectés mais sans être désalloués.

La similitude avec le code précédent ne vous aura pas échappé... le problème c'est que l'on doit choisir entre l et \*l, fonction ou procédure et décider ce qu'est ce tmp de malheur...

Encore une presque copie conforme sauf que, et là aussi vous êtes trop nombreux pour que cela soit excusé, le test if(l) et le résultat : en effet, dans le cas d'un passage de paramètre comme vous l'avez choisi, le test pour savoir si la liste est vide est évidemment if(\*l) et le résultat doit être \*l=insert(...) On remarque au passage une incohérence entre l'ordre des paramètres de la version de insert de l'itératif et celle du récursif .

D'abord il n'est pas très logique d'appeler liste un type qui est manifestement plutôt un maillon( bien qu'il ne soit pas défini dans la copie). A part cela, le problème principal de ce code est de réaliser les opérations à l'envers : l'insertion ne peut normalement se faire que sur une liste triée, or ici on insère avant tout (dans quoi, d'ailleurs ? ce n'est pas très clair), et bien entendu la liste de départ n'est pas triée

**MORALE DE L'HISTOIRE :** si dans l'avenir, vous avez l'occasion de me rendre un autre devoir, si vous le faites collectivement, rendez-le collectivement !!! De plus, un travail collectif est normalement l'occasion de se relire les uns les autres. Je constate que pour ce devoir, certains se sont juste contentés de copier coller. Comme je ne sais pas qui, certains ont certainement eu plus de chances que d'autres.