TD No1

Exercice 1: recherche du nombre d'occurence d'un élément dans un tableau

Proposez un algorithme permettant de trouver le nombre d'occurences d'un élément dans un tableau. Vous devrez prouver que votre algorithme termine et fournit le bon résultat. Déterminez la complexité de votre algorithme.

Exercice 2: recherche du second plus grand élément d'un tableau

Proposez un algorithme qui recherche le second plus grand élément d'un tableau. Déterminez la complexité de votre algorithme..

Exercice 3: calcul de x^n

On propose l'algorithme suivant de calcul de x^n :

- si n=0 alors 1
- si n est pair alors retourner $(x^{\frac{n}{2}})^2$
- si n est impair alors retourner $x^{n-1} * x$

Ecrire la fonction C correspondante et déterminer la complexité de cet algorithme.

Exercice 4: calcul des sommes partielles

Soit T un tableau d'entiers. On note S(i,j) la somme des éléments de T d'indices compris entre i et j:

$$S(i,j) = \sum_{k=i}^{k=j} T[i]$$

- Ecrire un algorithme calculant la plus grandes de ces sommes partielles (les calculer toutes et garder la plus grande)
- Quelle relation y a-t-il entre S(i,j) et S(i,j+1) ? Déduisez en un algorithme optimisé.
- Comparez la complexité de vos deux algorithmes.

Exercice 5: inversion des éléments d'un tableau

Ecrire un algorithme qui inverse les éléments d'un tableau: le premier devient le dernier, le second devient l'avant-dernier, ...

Exercice 6: positifs puis négatifs dans un tableau

- Proposez un algorithme permettant d'obtenir à partir d'un tableau T un tableau T' tel que
  - T' contient tous les éléments de T
  - Dans T', les nombres négatifs sont avant les nombres positifs
- On souhaite maintenant modifier directement le tableau T sans utiliser de tableau auxiliaire. Proposez l'algorithme correspondant

Vous calculerez la complexité de vos algorithmes.

Exercice 7: n-uplets de 0 et de 1

Proposez un algorithme permettant d'afficher tous les n-uplets composés de 0 et de 1. Ainsi, si n vaut 2, on souhaite obtenir les n-uplets suivants: 00 01 10 11 (dans l'ordre de votre choix).

Déterminez la complexité de votre algorithme.

TD No1: éléments de correction

Exercice 1: recherche du nombre d'occurence d'un élément dans un tableau

Proposez un algorithme permettant de trouver le nombre d'occurences d'un élément dans un tableau. Vous devrez prouver que votre algorithme termine et fournit le bon résultat. Déterminez la complexité de votre algorithme.

Algorithme:

```
// recherche du nombre d'occurences de l'élément X dans le tableau t
// nbElem: nombre d'élément de t. nbElem >= 0
// dans le cas où le tableau est vide, on retourne 0
int nbOccurence(Element x, tableau t, int nbElem){
   int cb, cmpt;
   assert(nbElem>=0);
   cmpt=0;
   for (cb=0; cb < nbElem; cb ++)
     if (t[cb]==x)
       cmpt++;
   return cmpt;
}
```

C'est un algorithme classique que de chercher le nombre d'éléments d'une structure vérifiant une propriété. Il s'écrit de façon générale de la façon suivante en utilisant une variable initialisée à 0 qui va être incrémentée à chaque découverte d'un élément vérifiant la propriété :

```
cmpt=0;
parcours de la structure
   si l'élément courant vérifie la propriété
   elors cmpt = cmpt + 1;
return cmpt;
```

Correction :

La **terminaison** ne pose pas problème car l'algorithme est constitué d'une boucle for simple dont le nombre d'itérations (nbElem) ne change pas lors de l'exécution de la boucle.

Posons l'**invariant de boucle** suivant: « x apparaît exactement cmpt fois dans la partie de tableau d'indices compris entre zéro et cb-1 ».

Initialisation : cet invariant de boucle est valable au début de la première boucle (la partie de tableau est vide car cb==0)

Conservation: si l'invariant de boucle est vrai à la ième itération et s'il y a une (i+1)ème itération:

- lors de la ième itération, cb vaut i-1
- si t[cb]==x alors cmpt est incrémenté. Or, le nombre d'occurences de x dans partie de t d'indices compris entre 0 et cb est égalà 1+ « le nombre d'occurences de x dans la partie du tableau d'indice compris entre 0 et cb-1 »=la valeur actuelle de cmpt.
- Si t[cb] != x alors la valeur de cmpt ne change pas. Or, le nombre d'occurences de x dans partie de t d'indices compris entre 0 et cb est égal au « nombre d'occurences de x dans la partie du tableau d'indice compris entre 0 et cb-1 « =la valeur actuelle de cmpt.
- À la fin de la ième boucle et donc au début de la (i+1)ème, l'invariant de boucle est donc

vérifié dans les deux cas.

La valeur de l'invariant de boucle lors de la dernière itération (cb==nbElem) nous permet de conclure sur la correction de l'algorithme car dans ce cas, la partie de tableau dont parle l'invariant de boucle correspond à tout le tableau.

Complexité:

Détermination des opérations fondamentales: au pif comme d'habitude. Si on se trompe d'opérations fondamentales, on énonce un résultat juste (l'algo nécessite tant de telles opérations) mais sans intérêt pratique car ce nombre d'opérations n'est pas alors pas proportionnel au temps d'exécution.

On choisit la comparaison **d'éléments** comme opération fondamentale. Ne pas confondre avec les comparaison d'entiers comme celles que l'on trouve dans « cb < nbElem ». L'une des raisons et notamment que, dans la vraie vie, les éléments sont des choses complexes dont la comparaison nécessite beaucoup plus de temps que la comparaison de deux entiers.

Il y a une comparaison dans la boucle for. La complexité en nombre de comparaisons d'éléments est donc égale à $\sum_{cb=0}^{nbElem-1} 1 = nbElem$

Exercice 2: recherche du second plus grand élément d'un tableau

Proposez un algorithme qui recherche le second plus grand élément d'un tableau. Déterminez la complexité de votre algorithme..

Correction:

L'énonce de l'exercice est imprécis. Exemple: Si le tableau contient les nombres suivants : 1;5;4;3;7;1;10;6;8;10;7. Le second plus grand élément du tableau est-il 10 (qui apparaît 2 fois donc une fois en première place et une fois en seconde) ou bien 8 (qui est la seconde plus grande valeur du tableau). Dit plus rigoureusement, laquelle des deux affirmations suivantes est-elle la bonne :

- on trie les éléments du tableau dans l'ordre décroissant (10,10,8,7,7,6,5,4,3,1,1) et on prend le deuxième éléments. Dans ce cas, on retourne 10. Tout ceci suppose qu'il y a au moins 2 éléments dans le tableau.
- on détermine la plus grande valeur, on ôte toutes ses occurences (1;5;4;3;7;1;6;8;7)et on prend le plus grand élément. Dans ce cas, on retourne 8. Tout ceci suppose qu'il y a au moins deux valeurs distinctes dans le tableau.

C'est au rédacteur de l'exercice de la préciser. :-) Comme on ne sait pas ce qu'il en est, on fera les deux.

```
// fonction vue en cours
void echange(int * a, int * b){
  int tmp;
  tmp=*a;
  *a=*b;
  *b=tmp;
}
// retourne le second plus grand élément du tableau t,
// c'est à dire le second élément d'une version triée de t
// t doit avoir au moins 2 éléments (nbElem >=2)
// nbElem: nombre d'éléments du tableau.
```

```
int max2_m1(tableau t, int nbElem)
{
  int i, max, max2,tmp;
  assert(nbElem >= 2);
  max=t[0];
  max2=t[1];
  if (max2 > max)
    echange(&max, &max2);
  for (i=2; i< nbElem; i++){
    tmp=t[i];
    if (tmp > max2){
      max2=tmp;
      if (max2 > max)
        echange (&max, &max2);
    }
  }
 return max2;
}
```

Pour expliquer l'algo ci-dessus, j'explique que max1 contient le plus grand élément de la partie de tableau déjà examinée et max2 contient le deuxième. Je fais remarquer que cette remarque naturelle revient à définir un invariant de boucle. Comme quoi les invariants de boucle font partie de la vie courante même si on en a pas concience. :-) Un problème classique est de savoir quoi prendre comme valeurs initiales pour nos variables (ici max1 et max2). Dans le premier algo, on peut simplement prendre t[0] et t[1] (le plus grand des deux dans max1 et l'autre dans max2). Dans le second algo (ci-dessous), ce n'est plus possible car il faut que max1 != max2. On va donc rechercher la première valeur v différente de t[0]. On mettra v et t[0] dans max1 (la plus grande) et dans max2 (l'autre).

La seconde valeur du tableau:

```
// retourne la seconde plus grande valeur de t : le max du
// tableau obtenu en enlevant son max à t
// t doit contenir au moins deux valeurs différentes
// nbElem: nombre d'éléments du tableau
int max2_m2(tableau t)
{
  int i, max, max2,tmp;
  max=t[0];
// on cherche une valeur != max dans t :
  i=1;
  while ((i<nbElem) && (t[i]==max))
    i++;
  assert(i < nbElem); // equivaut à l'existence de 2
                      // valeurs != dans le tableau
  max2=t[i];
  if (max2 > max)
    echange(&max, &max2);
  // arrivé là, max contient la plus grande valeur des celules d'indice <= i
  // max2 contient la seconde plus grande valeur (!= max) des celules
```

```
  // d'indice <= i
  // il reste à traiter le reste du tableau
  for (;i< nbElem; i++){ // on garde la valeur précédente de cb
    tmp=t[i];
    if ((tmp > max2) && (tmp != max)){
      max2=tmp;
      if (max2 > max)
        echange (&max, &max2);
    }
  }
  return max2;
}
```

Exercice 3: calcul de x^n

On propose l'algorithme suivant de calcul de x^n :

- si n=0 alors 1
- si n est pair alors retourner $(x^{\frac{n}{2}})^2$
- si n est impair alors retourner $x^{n-1} * x$

Ecrire la fonction C correspondante et déterminer la complexité de cet algorithme.

Corrigé:

```
// calcul de x^n
// n entier positif ou nul
// x entier
int puissance(int x, int n){
      assert(n>=0);
      if (n%2==0)
        return carre(puissance(x,n/2));
      else return x*puissance(x,n-1));
}
// détermine le carré d'un entier
// (pour éviter de calculer deux fois puissance(x, n/2) dans puissance)
int carre(int a){
      return a*a;
}
```

Complexité: opération fondamentales: opération arithmétique (/, -)

On note c(n) la complexité de puissance(x,n).

- si n est pair, $c(n)=2+c(\frac{n}{2})$
- si n est impair, $c(n)=2+c(n-1)=2+(2+c(\frac{n-1}{2}))$ car n-1 est pair

En admettant que c(n) est une fonction croissante de n, on en déduit

$c(n)\leqslant 4+c(\frac{n}{2})\leqslant 4i+c(\frac{n}{2^i})$ si $n>0$

Soit k tel que $2^{k-1} \leqslant n < 2^k$ soit $\frac{1}{2} \leqslant \frac{n}{2^k} < 1$ On en déduit que k est le plus petit k tel que

$\frac{n}{2^k} = 0$ et $c(n) \leqslant 4k$

De la définition de k, on déduit que k=partie entière de $\log_2(n)$. On a donc affaire à une complexité qui est au plus logarithmique (O(log n)).

Exercice 4: calcul des sommes partielles

Soit T un tableau d'entiers. On note S(i,j) la somme des éléments de T d'indices compris entre i et j:

$$S(i,j) = \sum_{k=i}^{k=j} T[i]$$

- Ecrire un algorithme calculant la plus grandes de ces sommes partielles (les calculer toutes et garder la plus grande)
- Quelle relation y a-t-il entre S(i,j) et S(i,j+1) ? Déduisez en un algorithme optimisé.
- Comparez la complexité de vos deux algorithmes.

Correction:

Remarque: le problème n'a d'intérêt que si le tableau contient des nombres négatifs et des nombres positifs.

L'algorithme qui calcule la somme des éléments d'une structure est un algorithme classique qui peut s'écrire schématiquement de la façon suivante en utilisant une variable s dans laquelle vont venir d'accumuler les valeurs des éléments de la structure:

```
s=0;
on parcourt tous les éléments de la structure
  s=s+élément courant
```

On va appliquer cet algorithme pour calculer la somme des éléments d'indice compris entre a et a de notre tableau:

```
// retourne la somme des éléments du tableau d'indices compris entre a et b
//  inclus.
// on suppose que a et b sont des indices valides et que a < b :
//    - a positif ou nul
//    - b strictement inférieur au nombre d'élément du tableaux
//   - a < b
int somme(int a, int b, tableau t, int nbElem){
  int res;
  assert(a >= 0);
  assert(b < nbElem);
  assert(a < b);
  res=0;
  for (cb=a; cb <= b; cb++)
    res=res+t[cb];
  return res;
}
```

On peut maintenant utiliser la fonction somme pour répondre au problème posé :

Le calcul de la plus grande des valeurs d'un ensemble d'objet est un algorithme classique qui peut

s'écrire de la façon suivante :

```
m=valeur initiale plus petite que le max des éléments de l'ensemble
on parcourt l'ensemble
  si la valeur courante est supérieure à m
  alors m = valeur courante
on retourne m
```

Le seul problème consiste à trouver une valeur initiale intelligente inférieure à la plus grande valeur de l'ensemble. Exemple: pour déterminer la plus grande valeur stockée dans un tableau, on prendra la première valeur t[0] comme valeur initiale car le max est forcément supérieur ou égal à t[0].

Appliquons tout ça à notre problème :

```
// calcul de la plus grande somme partielle
// des éléments d'un ensemble
// le tableau ne doit pas être vide (ça se discute mais on prend ça pour
// éviter de parler de somme des éléments d'un tableau vide
int plusGrandeSomme(tableau t, int nbElem){
  int i,j,res,tmp;
  assert(nbElem >0);
  s=somme(0,0,t, nbelem);
  for(i=0; i< nbElem; i++)
    for(j=i; j < nbElem; j++){
      tmp=somme(i,j,t,nbElem); // pour éviter de calculer 2 fois la somme
      if (s<tmp)
        s=tmp;
    }
  return s;
}
```

Alog optimisé:

```
// calcul de la plus grande somme partielle
// des éléments d'un ensemble
// le tableau ne doit pas être vide (ça se discute mais on prend ça pour
// éviter de parler de somme des éléments d'un tableau vide
int plusGrandeSomme(tableau t, int nbElem){
  int i,j,res,tmp;
  assert(nbElem >0);
  s=t[0];
  for(i=0; i< nbElem; i++){
    tmp=0;
    for(j=i; j < nbElem; j++){
      tmp=tmp+t[j];
      if (s<tmp)
        s=tmp;
    }
  return s;
}
```

Complexite: on admet que $\sum_{i=0}^{n} x^i = O(x^{n+1})$ On peut alors montrer que le premier algo est un $O(n^3)$ tandis que le second est un O(n²).

Exercice 5: inversion des éléments d'un tableau

Ecrire un algorithme qui inverse les éléments d'un tableau: le premier devient le dernier, le second devient l'avant-dernier, ...

Correction:

Les erreurs à ne pas faire :

Avant d'écraser une valeur (t[i] ici), il faut être sur de ne plus en avoir besoin :

```
for (i=0, i < nbElem; i++)
  t[i]=t[nbElem-i-1];
```

Le tableau {1,5,3,4}devient {4,3,3,4}

Autre erreur classique: erreur de borne

```
for (i=0, i < nbElem; i++)
  echange(&(t[i]), &(t[nbElem-i-1]));
```

Il fallait se limiter à (i < nbElem/2).

Une solution (pédagogiquement intéressante parce qu'elle utilise deux « mémorisateurs de position »: i et j):

```
i=0;j=nbElem-1;
while (i < j){
  echange(&(t[i]), &(t[j]));
  i++;
  j--;
}
```

Exercice 6: positifs puis négatifs dans un tableau

- Proposez un algorithme permettant d'obtenir à partir d'un tableau T un tableau T' tel que
  - T' contient tous les éléments de T
  - Dans T', les nombres strictement négatifs sont avant les nombres positifs ou nuls
- On souhaite maintenant modifier directement le tableau T sans utiliser de tableau auxiliaire. Proposez l'algorithme correspondant

Vous calculerez la complexité de vos algorithmes.

Correction:

Premier algo :

```
tableau posneg(tableau t, int nbElem){
  int i,j,cb;
  tableau res;
  assert(nbElem >= 0);
  i=0;
  j=nbElem-1;
```

```
  for (cb=0; cb < nbElem; cb++)
    if (t[cb]<0) {
       res[i]=t[cb];
       i++;
    } else {
     res[j]=t[cb];
     j--;
    }
  return res;
}
```

Algo sur place :

comme je n'ai pas envie de parler de passage de tableaux en paramètres, je ne présente pas l'algo sous la forme d'une fonction.

```
  int i,j,cb;
  assert(nbElem >= 0);
  i=0;
  j=nbElem-1;
  while (i<j){
    // on cherche un positif trop à gauche
    while ((i<j) && (t[i]<0))
      i++;
    // on cherche un négatif trop à droite
    while ((i<j) && (t[j]<0))
      j--;
    // si i<j, on les échange et on repart pour un tour
    if (i<j)
       echange(&(t[i]), &(t[j]));
  }
}
```

Dans les deux cas, complexité linéaire en nombre de comparaisons.

Exercice 7: n-uplets de 0 et de 1

Proposez un algorithme permettant de d'afficher tous les n-uplets composés de 0 et de 1. Ainsi, si n vaut 2, on souhaite obtenir les n-uplets suivants: 00 01 10 11 (dans l'ordre de votre choix). Déterminez la complexité de votre algorithme.

Correction:

En le faisant au tableau à la main, on voit que l'on est obligé d'utiliser la ligne du dessus pour savoir où l'on en est. Appliquer un algorithme similaire nécessitera un tableau pour mémoriser l'état courant.

```
#define NMAX 200
typedef struct ttableau {
   int tab [NAMX];
   int taille;
```

```
} tableau;
void affiche(tableau t){
  int cb;
  assert(t.taille >=0);
  for(cb=0; cb<t.taille; cb++)
    printf("%d",t.tab[cb]);
  printf("\n");
}
void nuplets(int n, tableau * pt){
  if (n == 0)
    affiche(*pt);
  else {
    pt->tab[n]=0;
    nuplets(n-1,pt);
    pt->tab[n]=1;
    nuplets(n-1,pt);
  }
}
```

complexité en nombre d'affectations au tableau: C(n)=2+2C(n-1)=2+2(2+2C(n-2))=2+2^2+2^4+ ... +2^n (de tête, à vérifier)