Ecole Supérieure en Génie Electrique et Energétique d'Oran

Algorithmique 2 et Structures de Données Avancées

SUPPORT D'EXERCICES

Classes Préparatoires

(Ancien Programme)

par

Fatima Zohra LEBBAH

À ma mère

TABLE DES MATIÈRES

INTRODUCTION GÉNÉRALE

Ce document est un récapitulatif des exercices et problèmes traités en travaux dirigés, en travaux pratiques et en examens avec nos étudiants de la deuxième année des classes préparatoires au sein de l'école.

Les exercices sont classés par chapitre conformément au contenu du support de cours. Nous donnons une suite de problèmes, dont chacun des énoncés permet de s'exercer sur une bonne partie du programme pédagogique.

Les cinq premiers chapitres comportent des exercices avec leurs solutions. Des exercices que nous avons proposés dans les fiches de travaux dirigés, les sujets des épreuves de devoirs surveillés ou d'examens de synthèse.

Le chapitre six comporte des exercices supplémentaires. Nous avons évité de donner les solutions correspondantes, il est préférable à ce que ça soit traité en collaboration avec l'enseignant en travaux pratiques.

Dans le dernier chapitre, nous proposons une suite de problèmes avec solutions. Cette partie représente les énoncés de problèmes que nous avons proposés dans certaines épreuves et dans les sujets conçus pour le concours de passage aux grandes écoles.

1

RAPPEL DE COURS

1.1 EXERCICES

Exercice 1 : (tableaux)

1. Écrivez un algorithme qui permet de :
   a) définir deux vecteurs **Vect1** d'entiers positifs et **Vect2** de caractères de même taille 50,
   b) saisir les valeurs de **Vect1** et **Vect2**, sachant que **Vect1**$[i]$ correspond au code *ASCII* de l'élément **Vect2**$[i]$ saisi,
   c) donner le nombre d'occurrences du caractère 'a' dans **Vect2**.
2. Donnez l'équivalent de l'algorithme en C++.

Exercice 2 : (matrices)

1. Écrivez un algorithme qui remplit un tableau **T** à deux dimensions 100×100 de valeurs booléennes en mettant 1 dans $\mathbf{T}[i, j]$ si $i + j$ est pair, 0 sinon.
2. Donnez le programme C++ équivalent.

Exercice 3 : (enregistrements)

Un nombre complexe donné sous la forme $a + \mathbf{i}b$ est représenté en machine par une variable de type enregistrement (article).

1. Donnez dans le langage algorithmique :
   a) la définition de l'enregistrement **complexe**,
   b) la définition du vecteur **C** de valeurs complexes, de taille 20,
   c) un algorithme qui réalise et affiche la somme des 20 valeurs.
2. Donnez l'équivalent en C++.

Exercice 4 : (fonctions)

1. Donnez dans le langage algorithmique :
   a) la fonction *puissance* qui calcule X^n, tel que $X_i \in \mathbb{R}$ et $n_i \in \mathbb{N}$.

b) la fonction *somme* qui fournit la somme $X_1^{n1} + X_2^{n2} + X_3^{n3} + \cdots$ pour une suite de paires (X_i, n_i) données (telles que : $X_i \in \mathbb{R}$ et $n_i \in \mathbb{N}$) qui se termine par la paire $(0, 0)$[Bensaoud-Senhadji, 2005].

c) l'algorithme qui fait appel à la fonction *somme*.

2. Donnez l'équivalent des fonctions et l'algorithme en C++.

Exercice 5 : (procédures)[Bensaoud-Senhadji, 2005]

Donnez la procédure *nombre* qui n'a aucun paramètre en entrée. Cette procédure demande à l'utilisateur de saisir un nombre entre 10 et 20, jusqu'à ce que la réponse convienne. En cas de réponse supérieure à 20, on affiche un message : "Plus petit !" et inversement "Plus grand !" si le nombre est inférieur à 10.

Exercice 6 : (notions de base)[Bensaoud-Senhadji, 2005]

Un grand magasin offre à ses clients la possibilité de bénéficier d'une réduction sur le total des achats effectués. Le taux de réduction est lié au résultat obtenu suite au lancement du dé. Il est déterminé comme suit :

— résultat obtenu 1 ou 6 : réduction de 5%,
— résultat obtenu 2 ou 5 : réduction de 8%,
— résultat obtenu 3 ou 4 : réduction de 10%.

Lire le total de la facture d'un client ainsi que le résultat obtenu, suite au lancement du dé et afficher le total à payer.

Exercice 7 : (instruction de contrôle)

Soit le programme ci-dessous écrit en *C* :

```
#include <stdio.h>
int main()
{
  int i;
  printf("Tapez un entier entre 1 et 3 bornes incluses : ");
  scanf("%d",i);
  switch(i)
  {
    case 1: printf("GAGNÉ\n");
            i=i+99;
            break;
    case 2: printf("PERDU num 2\n");
            i=0;
            break;
    case 3: printf("PERDU num 3\n");
            i=0;
            break;
    default : break;
  }
  printf("%d est la valeur finale de i ", i);
  return 0;
}
```

1. Réécrire ce programme en utilisant les mots clé *if* et *else*.
2. Réécrire ce dernier, en ne faisant appel qu'aux nouvelles possibilités d'entrées-sorties de C++, donc en remplaçant les appels à *pritf* et *scanf*.

Exercice 8 : (enregistrements et tableaux)

Pour l'inscription en deuxième année au sein de l'école, on a pour chaque étudiant les renseignements suivants : le numéro d'inscription, le nom, le prénom, la date et le lieu de naissance, l'adresse, le numéro de téléphone, la moyenne obtenue au baccalauréat et la moyenne obtenue en première année. Ces données sont regroupées dans une structure *enregistrement* dont on doit définir le type. Donner dans le langage C++ :

1. le type *TEtudiant*, qui regroupe tous les renseignements de l'étudiant,
2. le type *TvEtud* qui définit un vecteur de valeurs de type *TEtudiant* et de taille 100,
3. la fonction *Nom* qui fournit le nom de l'étudiant ayant la plus grande moyenne obtenue en première année.
4. la fonction *Moyenne* qui fournit la moyenne des moyennes obtenues au baccalauréat.

1.2 SOLUTIONS

Exercice 1 : (tableaux)

1. L'algorithme :

```
var Vect1 : tableau[1...50] de entier positif;
    Vect2 : tableau[1...50] de caractère;
    i,occur : entier;
Algorithme-principal Vect1-Vect2
début
occur← 0;
pour i de 1 à  50 faire
        écrire("Vect1[",i,"]=");
        lire(Vect1[i]);
        Vect2[i]← chr(Vect[i]);
        si Vect2[i]='a' alors occur← occur+1;
        finsi
finfaire
écrire("Le nombre d'occurrence de 'a' est : ", occur);
fin
```

2. Le programme en C++ :

```
#include<iostream>
using namespace std;
main()
{
  unsigned int  Vect1[50];
  char  Vect2[50];
  int occur=0;
  for(int i=0;i<50;i++)
  {
        cout << "Vect1[" << i <<"]=";
        cin >>Vect1[i];
        Vect2[i]=chr(Vect[i]);
        if (Vect2[i]='a') occur++;
  }
  cout << "Le nombre d'occurrence de 'a' est : " << occur << endl;

}
```

Exercice 2 : (matrices)

1. L'algorithme :

```
var T : tableau[1...100,1...100] de booléen;
    i,j : entier;
Algorithme-principal MatriceT
début
  pour i de 1 à  50 faire
        pour j de 1 à  50 faire
                si i+j mod 2=0 alors T[i,j]← 1;
                       sinon T[i,j]← 0;
                finsi
        finfaire
  finfaire
fin
```

2. Le programme en C++ :

```
#include<iostream>
using namespace std;
main()
{
  bool  T[100][100];
  for(int i=0;i<100;i++)
        for(int j=0;j<100;j++)
                if (i+j % 2==0) T[i][j]=1;
                else T[i][j]=0;
}
```

Exercice 3 : (enregistrements)

1. L'algorithme :

```
type article complexe
        {
                re : réel;
                im : réel;
        };
var C : tableau[1...20] de complexe;
    sre,sim : réel;
Algorithme-principal ComplexAlgo
début
        sre← 0;
        sim← 0;
        pour i de 1 à  20 faire
                écrire("Donnez la partie réel du nombre
                complexe ",i," : ");
                lire(C[i].re);
                écrire("Donnez la partie imaginaire du nombre
                complexe ",i," : ");
                lire(C[i].im);
                sre←sre+C[i].re;
                sim←sim+C[i].im;

        finfaire
        écrire("La somme = ", sre,"+i",sim);
fin
```

2. Le programme en C++ :

```
#include<iostream>
using namespace std;
typedef article complexe
        {
                re : float;
                im : float;
        };
complexe  C[20];

main()
{
        float sre=sim=0;
        for(int i(0);i<20;i++)
        {
                cout <<"Donnez la partie réel du nombre
                complexe "<< i << " : ";
                cin >> C[i].re;
                cout << "Donnez la partie imaginaire du nombre
                complexe " << i << " : ";
                cin >> C[i].im;
                sre=+C[i].re;
                sim=+C[i].im;
        }
        cout << "La somme = " << sre << "+i" << sim << endl;
}
```

Exercice 4 : (fonctions)

1. La fonction *puissance* :

```
fonction puissance(X:réel,n:entier):réel
var i : entier naturel;
        p : réel;
début
        p← 1;
        pour i de 1 à  n faire
                p←p*X;
        finfaire
        puissance←p;
fin
```

2. La fonction *somme* :

```
fonction somme():réel
var ni:entier naturel;
        s,Xi:réel;
début
        s← 0;
        écrire("Xi=");
        lire(X);
        écrire("ni=");
        lire(ni);
        tant que ((Xi≠0 ou ni≠0))
        faire
                écrire("Xi=");
                lire(X);
                écrire("ni=");
                lire(ni);
                s←s+puissance(Xi,ni);
        finfaire
        somme←s;
fin
```

3. L'algorithme :

```
Algorithme-principal PuissSommeAlgo
début
        écrire("Le résultat de la fonction somme = ",somme());
fin
```

4. Le programme en C++ :

```
#include<iostream>
using namespace std;

float puissance(float,int);
float somme();
main()
{
        cout << "Le résultat de la fonction somme = "
        << somme() << endl;
}

float puissance(float X,int n)
{
        unsigned int i;
        float p=1,;
        for(int i(0);i<=n;i++)
                p=*X;
        return p;
}
float somme()
{
        unsigned int ni;
        float s(0),Xi;
        cout << "Xi=";
        cin >> X;
        cout << "ni=";
        cin >> ni;
        while ((Xi!=0 || ni!=0))
        {
                cout << "Xi=";
                cin >> X;
                cout << "ni=";
                cin >> ni;
                s=+puissance(Xi,ni);
        }
        return s;
}
```

Exercice 5 : (procédures)

```
procedure()
variable N : entier;
début
    écrire("Entrez un nombre compris entre 10 et 20 : ");
    lire(N);
    tant que (N < 10 ou N >20)
    faire
        si (N < 10) alors écrire("Plus grand !")
                    sinon écrire("Plus petit !")
        finsi
        lire(N);
    fintq
fin
```

Exercice 6 : (notions de base)

```
#include <iostream>
using namespace std;
main()
{
        float fact; // total de la facture
        cout << "Tapez le total de la facture SVP : ";
        cin >> fact;
        unsigned int de;
        cout << "Tapez le résultat du lancement du dé SVP : ";
        cin >> de;
        float red (0); // taux de la réduction
        switch (de){
                case 1 :
                case 6 : red=0.05; break;
                case 2 :
                case 5 : red=0.08; break;
                case 3 :
                case 4 : red=0.1; break;
                default : break;
        }
 fact = fact-(fact*red);
 cout << "la somme à payer est : " << fact << endl;
}
```

Exercice 7 : (instructions de contrôle)

Le programme en $C++$:

```
#include <iostream>
using namespace std;
int main()
{
        int i;
        cout << "Tapez un entier entre 1 et 3 bornes incluses : ";
        cin >> i;
        if (i==1){
                  cout << "GAGNÉ\n";
                  i=i+99;
        }
        else
                if (i==2){
                        cout << "PERDU num 2\n";
                        i=0;
                }
                else
                        if (i==3){
                                cout << "PERDU num 3\n";
                                i=0;
                        }
        cout << i << " est la valeur finale de i \n";
        return 0;
}
```

Exercice 8 : (enregistrements et tableaux)

1. Le type *TEtudiant* :

```
struct date_nais{
        unsigned int jj, mm, aa;
};
struct TEtudiant{
        unsigned int insc; //numéro d'nscripiton
        string nom, prenom, lieu_nais, tel, adres;
        date_nais date;
        float moy_bac, moy_a1;
};
```

2. Le type *TvEtud* :

```
typedef TEtudiant TvEtud[100];
```

3. La fonction *Nom* :

```
string nom(TvEtud t)
{
        float max=-1; string nm=" ";
        for(int i=0; i<nbe; i++)
                if (t[i].moy_a1>max) {
                        max = t[i].moy_a1;
                        nm=t[i].nom;
                };
        return nm;
};
```

4. La fonction *Moyenne* :

```
float moyenne(TvEtud t)
{
        float moy(0);
        for(int i=0; i<nbe; i++)
                moy = moy+t[i].moy_bac;
        moy=moy/nbe;
        return moy;
};
```

5. Le *main* :

```
#include <iostream>
using namespace std;

int nbe; //le nombre d'étudiants
// prototypes des fonctions
string nom(TvEtud);
float moyenne(TvEtud);

int main()
{
        TvEtud et;
        cout << "Donnez le nombre d'étudiants : "; cin >> nbe;
  //saisie des renseignements des étudiants
        for (int i=0; i<nbe;i++)
        {
                cout << "L'ETUDIANT NUM " << i+1 << " :" << endl;
                cout << "Le numéro d'inscription = ";
                cin >>  et[i].insc;
                cout << "Le nom de l'étudiant = ";
                cin >>  et[i].nom;
                cout << "Le prénom de l'étudiant = ";
                cin >>  et[i].prenom;
                cout << "La date de naissance : " << endl;
                cout << "jour : "; cin >>  et[i].date.jj;
                cout << "mois : "; cin >>  et[i].date.mm;
                cout << "année : "; cin >>  et[i].date.aa;
                cout << "Le lieu de naissance = ";
                cin >>  et[i].lieu_nais;
                cout << "Le numéro de téléphone = ";
                cin >>  et[i].tel;
                cout << "L'adresse = "; cin >>  et[i].adres;
                cout << "La moyenne au baccalauréat = ";
                cin >>  et[i].moy_bac;
                cout << "La moyenne de la première année = ";
                cin >>  et[i].moy_a1;
        }
        //Appel des fonctions
        cout << "*** APPLICATION ****"<< endl;
        cout << "Le nom de l'étudiant ayant la meilleure moyenne
         en 1ère année : " << nom(et) << endl;
        cout << "La moyenne des moyennes obtenues au baccalauréat
        : " <<  moyenne(et)<<endl;
}
```

2

TECHNIQUES DE PROGRAMMATION C++

2.1 EXERCICES

Exercice 1 : (notions de base)

1. Soit le le bloc d'instructions suivant :

```
int x=3;
int * px;
int y=-10;
int * py=&y;
px = &x;
*py = y + *px;
x = *py-x;
cout << "x=" << x << endl;
```

   Q : Quelle est la valeur de x ?

2. Trouvez et corrigez les erreurs dans les blocs d'instructions suivants :

```
a)
int* p;
int x=35;
*p=x;

b)
int x=11;
int * p=x;
*p=11

c)
double * z;
int y=12;
int * py=&y;
z=py;

d)
int x;
int * p;
&x=p;

e)
char terme[10];
char c='B';
char * pc=&c;
terme=pc;
```

3. Soit le bloc d'instructions suivant :

```
int tab[10]={5,8,4,3,9,6,5,4,3,8};
printf("%d \n",*tab);
printf("%d \n",tab[0]);
int *pt = tab;
pt++;
printf("%d \n",pt[0]);
```

Q : Qu'affiche-t-il ?

4. Soit le prototype de la fonction *g* :

```
void g(int j, bool b=true, float y=1.1)
```

Q : Donnez les différents cas d'appels corrects de cette fonction ?

Exercice 2 : (tableaux et pointeurs)

1. Nous supposons que les valeurs d'un tableau T sont comprises entre 0 et max (avec max pas trop grand), et que l'on peut trier les valeurs en comptant tout d'abord le nombre de 0, le nombre de 1, le nombre de 2, ... le nombre de max en entrée. Ensuite, il suffit de parcourir le tableau à nouveau en indiquant la bonne quantité de chaque valeur [1].

   Donnez en C++ le programme qui :

   a) saisit la taille puis les valeurs du tableau dynamique T,

   b) calcule le *max* des valeurs saisies,

   c) ordonne les valeurs du tableau T dans l'ordre croissant, en utilisant un tableau annexe *count*, dont l'espace mémoire est réservé dynamiquement et où $count[i]$ indique le nombre de i dans le tableau initial en question. A ce niveau, l'accès aux éléments des deux tableaux doit être réalisé sans utiliser l'opérateur « [ ] ».

2. Donnez en C++ les fonctions :

   a) *RS* qui permet de trouver l'indice d'un entier positif x donné dans le tableau T non ordonné, en utilisant une recherche séquentielle,

   b) *RD* qui permet de trouver l'indice d'un entier positif x donné dans le tableau T trié dans l'ordre croissant, en utilisant une recherche dichotomique. Sachant que le principe de la recherche par dichotomie (du grec dikhotomia, division en deux parties égales) ou recherche binaire.

   Testez dans le *main*, la fonction *RS* avant le tri et la fonction *RD* après le tri.

Exercice 3 : (déroulement d'une fonction)

Soit la fonction suivante :

1. Le tri par dénombrement est proposé par Seward en 1954

```
const int N=20;
typedef bool MC[N][N];
bool fct(MC T, int taille = N)
{
    for(int i=0; i<taille-1;i++)
    {
        for(int j=i+1; j<taille; j++)
        {
            if (i != j)
            {
                int k=0;
                while (T[i][k] == T[j][k])
                {
                    if (k == taille-1) return true;
                    else k = k+1;
                }
            }
        }
    }
    return false;
}
```

Que fait la fonction *fct* ? justifiez.

Exercice 4 : (fonctions et portée des variables)

1. Donnez en C++ les fonctions suivantes :
   a) *minmax* qui a comme paramètres un tableau d'entiers, la taille du tableau et 2 pointeurs vers des entiers min et max. Elle doit renvoyer dans les entiers pointés par *min* et *max* respectivement le plus petit et le plus grand entier du tableau.
   b) *occurrence* qui a en paramètre une chaîne de caractères et qui renvoie le nombre d'occurrences de la lettre 'A'. Cette fonction devra parcourir la chaîne en utilisant un pointeur.
   c) *copy* qui a comme prototype *copy(char * ch1, char * & ch2)*. Cette fonction a comme paramètres d'entrée une chaîne de caractères *ch1* et une référence vers un pointeur *ch2*. Avant l'appel, *ch2* est un pointeur non initialisé. Après l'appel, *ch2* doit pointer sur un nouveau tableau de caractères qui contient une copie de la chaîne *ch1*. Ce nouveau tableau de caractères aura la taille minimale nécessaire [Delannoy, 1995].
2. Testez les trois fonctions dans un *main* (la fonction *occurrence* doit être testée sur un nombre quelconque de chaînes de caractères.).

Exercice 5 : (allocation dynamique)[Delannoy, 1999]

1. Donnez en C++ les instructions suivantes, en utilisant les opérateurs new et delete :

```
int* adi;
double* add;
    ...
adi = malloc(sizeof(int));
add = malloc(sizeof(double)*100);
```

2. Donnez un programme allouant des emplacements pour des tableaux d'entiers dont la taille se fournit en donnée. Les allocations se poursuivront jusqu'à ce que l'on aboutisse à un débordement mémoire. L'exécution se présente ainsi.

```
Taille souhaitée ? 600000
Allocation bloc numéro : 1
Allocation bloc numéro : 2
Allocation bloc numéro : 3
Allocation bloc numéro : 4
Allocation bloc numéro : 5
Mémoire insuffisante - arrêt exécution
```

Exercice 6 : (tableaux et fonctions)

Soit un tableau T d'entiers :

1. Donnez en C++ les fonctions ci-dessous :
   a) *void multiple(int *T, int n, ..., ...)* qui reçoit en entrée le tableau et retourne la longueur et l'indice du premier composant de la plus longue série d'éléments consécutifs égaux. Par exemple, la série la plus longue dans le tableau :

5	-1	-1	0	1	0	4	**5**	**5**	**5**

   est celle en gras de longueur 3 et elle commence à l'indice 7.
   b) *void monotone(int *T, int n, ...)* qui reçoit en entrée le tableau et retourne la longueur de la plus longue suite strictement monotone (croissante ou décroissante). Par exemple, la suite la plus longue dans le tableau :

5	-1	-2	0	3	10	**11**	**5**	**3**	**-1**

   est celle en gras de longueur 4.
   c) *void eliminate(int *T, int n, ..., ...)* qui permet d'allouer l'espace mémoire nécessaire pour un nouveau tableau, et d'y charger tous les éléments du tableau reçu en entrée après élimination des doublons.
2. En supposant que T est un tableau dynamique, donnez le *main* qui permet de :

— allouer l'espace mémoire nécessaire et lire les valeurs du tableau,
— tester les fonctions *multiple*, *monotone* puis *eliminate* et afficher les résultats respectifs.
— libérer l'espace mémoire alloué pour T.

NB : L'accès aux éléments du tableau doit se faire via l'opérateur "".*

Exercice 7 : (tableaux dynamiques)

En utilisant les pointeurs, donnez un programme en C++ qui permet de :

1. saisir la taille N d'un tableau T d'éléments de type *double*,
2. allouer l'espace mémoire suffisant pour le tableau T,
3. saisir les éléments du tableau T,
4. mettre dans un pointeur p l'adresse de la $5^{\text{ème}}$ case du tableau,
5. afficher le contenu de ce tableau à partir de la $5^{\text{ème}}$ case.

2.2 SOLUTIONS

Exercice 1 : (notions de base)

1. La valeur de x : $x = -10$.
2. Correction des erreurs dans les blocs d'instructions :

Bloc d'instructions	erreur	type de l'erreur	correction
a)	-	pas d'erreur	-
b)	*p=x	incompatibilité de types	*p=&x
c)	z=py	incompatibilité de types	int *z
d)	&x=p	incompatibilité de types	*p=x
e)	-	pas d'erreur	-

3. Les trois instructions affichent :

   5

   5

   8

4. Les différents cas d'appels corrects :
   a) g(1) ;
   b) g(1,false) ;
   c) g(1,false,3.0) ;

Exercice 2 : (tableaux et les pointeurs)

1. Le programme C++ :

```
#include<iostream>
using namespace std;

unsigned int n,mx(0),*T,*count;
main()
{
        cout << "La taille du tableau = ";
        cin >> n;
        T=new unsigned int[n];
        for(int i(0);i<n;i++)
        {
                cout << "T[" << i << "]=";
                cin >> T[i];
                if (mx<T[i]) mx=T[i];
        }
        count=new unsigned int[mx+1];
        for(int i(0);i<=mx;i++)
                count[i]=0;
        for(int i(0);i<n;i++)
                count[T[i]]++;
        int k(0);
        for(int i(0);i<=mx;i++)
        {
                for(int j(k);j<count[i]+k;j++)
                        T[j]=i;
                k+=count[i];
        }
}
```

2. Les fonctions en C++ :

   a) La fonction *RS* :

```
int RS(unsigned int x)
{
        for(int i(0);i<n;i++)
                if (T[i]==x) return i;
        return -1;
}
```

   b) La fonction *RD* :

```
int RD(unsigned int x)
{
  bool trouve;
  int id;  //indice de début
  int ifin;  //indice de fin
  int im;  //indice de "milieu"
  trouve = false;
  id = 0;
  ifin = n;
  im = (id + ifin)/2;
  while((T[im] != x) && (ifin >id))
  {
    if(T[im] > x) ifin = im;
    else id = im+1;
    im = (id + ifin)/2; //on détermine l'indice de milieu
  }
  if(T[id] == x) return(id);
  else return(-1);
}
```

Le test des fonctions dans le *main* :

```
#include<iostream>
using namespace std;
unsigned int n,mx(0),*T,*count;
int RS(unsigned int);
int RD(unsigned int);
main()
{
        cout << "La taille du tableau = ";
        cin >> n;
        T=new unsigned int[n];
        for(int i(0);i<n;i++)
        {
                cout << "T[" << i << "]=";
                cin >> T[i];
                if (mx<T[i]) mx=T[i];
        }
        count=new unsigned int[mx+1];
        for(int i(0);i<=mx;i++)
                count[i]=0;
        for(int i(0);i<n;i++)
                count[T[i]]++;
        int k(0);
        for(int i(0);i<=mx;i++)
        {
                for(int j(k);j<count[i]+k;j++)
                        T[j]=i;
                k+=count[i];
        }
        int X;
        cout << "X="; cin >> X;
        cout <<"L'indice de " << X << "dans T avec RS = "
        << RS(X) << endl;
        cout <<"L'indice de " << X << "dans T avec RD = "
        << RD(X) << endl;
}
```

Exercice 3 : (déroulement d'une fonction)

La fonction *fct* renvoie VRAI si et seulement si la matrice possède deux lignes identiques. Il faut justifier en donnant une trace de la fonction (ex : N=5).

Exercice 4 : (fonctions et la portée des variables)

1. Les fonctions en C++ :
   a) *minmax* :

```
void minmax(int* tab,int taille,int *min,int *max)
{
   *min=*max=tab[0];
   for (int i=1;i<taille;i++)
   {
       if (tab[i]>*max) *max=tab[i];
       if (tab[i]<*min) *min=tab[i];
   }
}
```

b) *occurrence* :

```
int occurrence(char* ch)
{
    int compteur=0;
    while(*ch)
    {
        if(*ch=='A') compteur++;
        ch++;
    }
    return compteur;
}
```

c) *copy* :

```
void copy(char * ch1, char * &ch2)
{
    int taille=strlen(ch1);
    ch2=new char[strlen(ch1)];
    for (int i=0;i<taille;i++)
    ch2[i]=ch1[i];
}
```

2. Test des trois fonctions :

```
#include <iostream>
using namespace std;
...
main()
{
    // test de la fonction minmax
    int n,min,max;
    cout <<"Taille du tableau?"; cin>>n;
    int *t=new int[n];
    for (int i=0;i<n;i++)
    {
        cout<<"t["<<i<<"]="; cin>>t[i];
    }
    minmax(t,n,&min,&max);
    cout<<"min="<<min<<",max="<<max;
    // test de la fonction occurrence
    int n,compteur=0;
    cout<<"Nombre de chaînes de caractères?";cin>>n;
    cin.ignore();
    char **chaines=new char* [n];
    for (int i=0;i<n;i++)
    {
        chaines[i]=new char[50];
        cout<<"Chaîne de caractères "<<i+1<<"-->";
        cin.getline(chaines[i],50);
        compteur+=occurrence(chaines[i]);
    }
    cout<<"Le nombre d'occurence de 'A' est:"<<compteur;
    //test de la fonction copy
    char *chaine1="Bonjour",*chaine2;
    copy (chaine1,chaine2);
    cout<<chaine2;
}
```

Exercice 5 : (allocation dynamique)

1. L'allocation en utilisant les opérateurs *new* et *delete* :

```
int * adr = new int [taille];
```

2. Le programme principal :

```
#include <stdlib.h>
#include <iostream>
#include <new>
using namespace std;

void deborde()
{
  cout << "Mémoire insuffisante - arrêt exécution \n";
  exit (1);
}

main()
{
  set_new_handler(deborde);
  long taille;
  int * adr;
  int nbloc;
  cout << "taille souhaitée ?";
  cin >> taille;
  for(nbloc=1;;nbloc++)
  {
        adr = new int [taille];
        cout << "Allocation bloc numéro : " << nbloc << "\n";
  }
}
```

Nous ajoutons l'instruction *set_new_handler(deborde)* afin d'éviter le phénomène de débordement [2].

Exercice 6 : (tableaux et fonctions)

1. Les fonctions :
   a) *void multiple(...)* :

2. À tester le programme sans l'instruction *set_new_handler(deborde)* afin d'analyser la sortie.

```
void multiple(int *T, int n, int& l, int& ind){
    int j, g; l=0; ind=0;
    for(int i(0); i<n-1; i++){
        if (*(T+i)==*(T+i+1)){
            g=2;
            for(j=i+1; j<n-1 && *(T+j)==*(T+j+1); j++)
                g++;
            if (g>=l)
            {
                l=g;
                ind=i;
            }
            i=j;
        }
    }
}
```

b) *void monotone(...)* :

```
void monotone(int *T, int n, int& l){
    int j, g; l=0;
       for(int i(0); i<n-1; i++)
        {
            if (*(T+i)<*(T+i+1))        {
                g=2;
                for(j=i+1; j<n-1 && *(T+j)<*(T+j+1); j++)
                    g++;
                if (g>=l) l=g;
                i=j;
            }else{
                    if (*(T+i)>*(T+i+1)){
                        g=2;
                        for(j=i+1; j<n-1 && *(T+j)>*(T+j+1); j++)
                            g++;
                        if (g>=l) l=g;
                        i=j;
                    }
                }
        }
}
```

c) *int eliminate(...)* :

```
void eliminate(int *T, int n, int* & SD, int & le){
    int j, g; le=0;
    for(int i(0); i<n-1; i++){
        le++;
        if (*(T+i)==*(T+i+1)){
            for(j=i+1; j<n-1 && *(T+j)==*(T+j+1); j++);
            i=j;
        }
    }
    if (*(T+n-1)!=*(T+n-2)) le++;
    SD=new int[le];
    int k=0;
    for(int i(0); i<n-1; i++){
        *(SD+k)=*(T+i); k++;
        if (*(T+i)==*(T+i+1)){
            for(j=i+1; j<n-1 && *(T+j)==*(T+j+1); j++);
            i=j;
        }
    }
    if (*(T+n-1)!=*(T+n-2)) *(SD+le-1)=*(T+n-1);
}
```

2. Le *main* :

```
main()
{
        int n, lg, d, le; int *SD;
        cout << "Donnez la taille du tableau : "; cin >> n;
        int * T = new int[n];
        for(int i(0); i<n; i++){
                cout << "T[" << i << "]="; cin >> *(T+i);
        }
        multiple(T,n,lg,d);
        cout << "La plus longue série d'éléments consécutifs
        égaux est de longueur : " << lg << " et commence à
        l'indice : " << d << endl;
        monotone(T,n,lg);
        cout << "La plus longue suite monotone est de longueur : "
        << lg << endl;
        eliminate(T,n,SD,le);
        cout << "Les valeurs du nouveau tableau : \n";
        for(int i(0); i<le; i++){
                cout << "SD[" << i << "]=" << *(SD+i) << endl;
        }
        delete[]T;
}
```

Exercice 7 : (tableaux dynamiques)

Le programme en C++ :

```
main(){
    int N;
    cout << "N=" ;
    cin >> N;
    double * T=new double[N];
    double *pt = T;
    for(int i=0; i<N; i++)
    {
        cout << "T[" << i << "]=" ; cin >> *pt;
        pt++;
    }
    double *p = T+5;
    for(int i=0; i<N-5; i++)
    {
        cout << "p[" << i << "]=" << *p << endl;
        p++;
    }
}
```

ou

```
main(){
    int N;
    cout << "N=" ;
    cin >> N;
    double * T=new double[N];
    double *pt = T;
    for(int i=0; i<N; i++)
    {
        cout << "T[" << i << "]=" ; cin >> *pt;
        pt++;
    }
    double *p = T+5;
    cout << "Affichage du contenu du tableau à partir
    de la case 5 : " << endl;
    while(p<T+N)
    {
        cout << *p<<" ";
        p++;
    }
}
```

3

RÉCURSIVITÉ

3.1 EXERCICES

Exercice 1 : (comparaison récursivité-itérativité)

Associez à chaque terme de la colonne A, une ou plusieurs expressions de la colonne B :

(A)	(B)
(1) fonction itérative (2) fonction récursive	(a) plus rapide (b) consomme plus en mémoire (c) plus lente (d) consomme moins en espace mémoire (e) facile à écrire (f) de complexité élevée

Exercice 2 : (récursivité)

Donner les fonctions récursives qui permettent respectivement de :

1. calculer les termes de chacune des suites suivantes :
   a) $U_{n+1} = \sqrt{(1 + U_n)}$, avec U_0 un réel donné.
   b) $U_{n+2} = a.U_{n+1} + b.U_n$, avec U_0, U_1, a et b des réels donnés.
   c) $U_n = 0 - 1 + 2 - 3 + 4 - 5 + 6 - 7 + \cdots$
2. afficher tout le contenu d'un tableau T de taille n.
3. vérifier si un nombre donné est palindrome.
4. tester si une chaîne de caractères donnée est un anagramme d'une autre chaîne de caractères. Par exemple : 'algorithme' est un anagramme de 'logarithme'.

Exercice 3 : (récursivité-itérativité)

1. Soit la fonction suivante :

```
void asterisque(int n)
{
   if (volume <= 0) return;
   for(int i=0; i<n; i++)
        cout << "*";
   cout << endl;
   asterisque(n-1);
}
```

a) que fait cette fonction ?

b) donnez la version itérative correspondante.

2. Soit la fonction suivante :

```
#include<math.h>
int f(int n)
{
    static int i=-1;
    if (n==0) return 0;
    else
    {
      i++;
      return (n%2*((int) pow(10,i)))+f(n/2);
    }
}
```

a) que fait cette fonction ?

b) donnez la version itérative correspondante.

Exercice 4 : (itérativité-récursivité)

1. Soit la fonction itérative suivante :

```
typedef int *TAB;
int f(TAB T, int N)
{
   int m = 0;
   for(int i=1; i<N; i++)
       if (T[m] < T[i]) m = i;
   return(m);
}
```

a) que fait cette fonction ?

b) transformez la fonction en sous-programme récursif.

2. Soit la fonction g, qui calcule le nombre d'occurrence (nombre d'apparitions) d'un entier X dans un tableau T de taille N.

```
typedef int * TAB;
int g(TAB T, int N,  int X)
{
    int p=0;
    for(int i=0; i<N; i++)
        if (T[i] == X) p++;
    return p;
}
```

Transformez en sous-programme récursif.

Exercice 5 : (tableaux et récursivité)

1. Donnez la fonction récursive *void image(int*,int)*, qui retourne l'image miroir d'un tableau d'entiers. Par exemple, l'image miroir du tableau | 5 | -1 | 10 | 0 | 11 | serait après exécution de la fonction | 11 | 0 | 10 | -1 | 5 |.
2. Testez la fonction dans un *main* et afficher le résultat. Ceci, en utilisant un tableau dynamique : la taille et les valeurs doivent être saisies par l'utilisateur et la mémoire allouée doit être libérée après usage.

NB : les deux questions peuvent être traitées indépendamment.

Exercice 6 : (fonction récursive)

Soit la fonction récursive suivante :

```
unsigned int c(0);
int f(char * X)
{
  if (*X=='\0') return c;
  else {
          if (*X>='0' && *X<='9') c++;
          f(X+1);
       }
}
```

1. Que fait la fonction f ? Justifier en traçant l'arbre d'exécution sur un exemple.
2. Donnez la version itérative.
3. Si on met la variable c comme paramètre d'entrée, donnez le nouvel entête de la fonction f.

Exercice 7 : (arbre d'exécution)

Soit la fonction g suivante :

```
int a,b;
int g(int n) {
  if (n==1 || n==2) return 1;
  else {
          a=g(n-1);
          b=g(n-2);
          return a+b;
        }
}
```

Évaluez *g(5)*, en donnant :

1. l'arbre d'exécution,
2. les différentes valeurs assignées aux variables *a* et *b*.

Si on déclare *a* et *b* comme variables locales à la fonction, donnez la valeur de *g(5)*.

Exercice 8 : (fonction récursive et arbre d'exécution)

1. Donnez la fonction récursive *int transform(char *, int)* qui permet de transformer une chaîne de caractères ne contenant que des chiffres en un entier.
2. Donnez l'arbre d'exécution de *transform* appliquée sur la chaîne de caractères "5269".

Exercice 9 : (fonction récursive et arbre d'exécution)

1. Donnez en C++ la fonction récursive *int sum(int)* qui reçoit en entrée un nombre entier positif et retourne la somme de ses chiffres.
2. Tracez l'arbre d'exécution de *sum(358)*.

Exercice 10 : (fonction de MORIS)

Soit la fonction de Morris :

```
int morris (int m, int n){
     if (m == 0) return 1;
     else return morris(m - 1, morris(m, n));
}
```

Expliquer pourquoi l'appel de *morris(1,0)* ne se termine pas !

Exercice 11 : (fonction de SYRACUSE)

Pour un entier $m > 0$ donné, la suite de *Syracuse* est définie par :

$$\begin{cases} u_0 = m \\ u_{n+1} = \begin{cases} \frac{u_n}{2} & si \quad u_n \quad est \quad pair \\ u_n \times 3 + 1 & si \quad u_n \quad est \quad impair \end{cases} \end{cases}$$

donnez la fonction récursive dont le prototype *int syracuse(int n, int m)* et qui calcule la suite définie ci-dessus.

Exercice 12 : (récursivité-itérativité)

1. Écrire la fonction récursive *puissance* qui reçoit en entrée un nombre entier positif n et fournit en sortie le nombre 10^k, avec k le nombre de chiffres contenus dans n.
2. Écrire, en utilisant la fonction *puissance*, la fonction récursive *miroir* qui, à partir d'un nombre entier positif, renvoie son image miroir. Par exemple, l'image miroir de 27113 est 31172.
3. Donner la version itérative de chacune des fonctions *puissance* et *miroir*.

Exercice 13 : (trace d'une fonction récursive)

Soit la fonction récursive suivante :

```
bool g(unsigned int x)
{
        if (x==0) return true;
        else if (x==1) return false;
         else return g(x-2);
}
```

Que fait la fonction g ? (à justifier)

3.2 SOLUTIONS

Exercice 1 : (comparaison récursivité-itérativité)

1. fonction itérative : (a), (d)
2. fonction récursive : (b), (c), (e), (f)

Exercice 2 : (récursivité)

Les fonctions récursives :

1. pour calculer les termes de chacune des suites :
   a) $U_{n+1} = \sqrt{(1+U_n)}$, avec U_0 un réel donné :

```
#include <math.h>
float U0;
...
float Ua(int n)
{
        if (n==0) return U0;
        else return sqrt(1+Ua(n-1));
}
```

   b) $U_{n+2} = a.U_{n+1} + b.U_n$, avec U_0, U_1, a et b des réels donnés :

```
float U0,U1,a,b;
...
float Ub(int n)
{
        if (n==0) return U0;
        else return a*Ub(n-1)+b*Ub(n-2);
}
```

   c) $U_n = 0-1+2-3+4-5+6-7+\cdots$:

```
#include <math.h>
float Uc(int n)
{
        if (n==0) return 0;
        else return pow(-1,n%2)*n+Uc(n-1);
}
```

2. pour afficher le contenu d'un tableau T de taille, nous avons deux possibilités :
   a) Affichage des éléments du tableau de l'indice 0 à l'indice $n-1$:

```
#include<iostream>
using namespace std;
void afficher(int * T,int n)
{
        if (n<0) return;
        else
        {
                afficher(T,n-1);
                cout << T[n] << endl;
        }
}
```

   b) Affichage des éléments du tableau de l'indice $n-1$ à l'indice 0 :

```
void afficher(int * T,int n)
{
         if (n<0) return;
         else
         {
                  cout << T[n] << endl;
                  afficher(T,n-1);
         }
}
```

3. la fonction palindrome :

```
#include<math.h>
int v;
bool palindrome(int nbre,int nbchiffre)
{
        v=nbre/(int)pow(10,nbchiffre-1);
        if (nbchiffre<=1) return true;
        else
          if (v==nbre%10)
            return palindrome((nbre-v*(int)pow(10,nbchiffre-1))
                                                  /10,nbchiffre-2);
          else return false;
}
```

4. la fonction anagramme :

```
bool anagramme(char*ch1,char*ch2,int nc1,int nc2)
{
  if (nc1!=nc2) return false;
  else
          if (nc1==0 && nc2==0) return true;
          else
                 {
                   for(int i(0);i<nc2;i++)
                   {
                         if (ch1[0]==ch2[i])
                         {
                           for(int k=0;k<nc1-1;k++)
                                 ch1[k]=ch1[k+1];
                           for(int k=i;k<nc2-1;k++)
                                 ch2[k]=ch2[k+1];
                           return anagramme(ch1,ch2,nc1-1,nc2-1);
                         }
                    }
                         return false;
                 }
}
```

Exercice 3 : (récursivité-itérativité)

1. La fonction *asterisque* :

   a) La fonction affiche un triangle d'astérisques. Par exemple l'appel *asterisque(10)* donne en sortie :

```
**********
*********
********
*******
******
*****
****
***
**
*
```

   b) La version itérative :

```
void asterisque(int n)
{
   while (n > 0)
   {
                for(int i=0; i<n; i++)
                        cout << "*";
                cout << endl;
                n--;
   }
}
```

2. la fonction f :

   a) La fonction *int f(int n)* retourne le nombre n en binaire. Par exemple, l'appel de la fonction $f(10)$ retourne la valeur 1010.

   b) La version itérative :

```
#include<math.h>
int f(int n)
{
        int i=-1,s(0);
        while(n!=0)
        {
          i++;
          s+=n%2*((int) pow(10,i));
          n=n/2;
        }
        return s;
}
```

Exercice 4 : (itérativité-récursivité)

1. La fonction itérative :
   a) La fonction f retourne l'indice de la plus grande valeur contenue dans le tableau en entrée.
   b) La version récursif :

```
typedef int *TAB;
int f(TAB T, int N)
{
   static int m = 0;
   if (N==1) return m;
   else
        {
                if (T[m] < T[N-1]) m = N-1;
                return f(T,N-1);
        }
}
```

2. La version récursive de la fonction g :

```
typedef int *TAB;
int g(TAB T, int N, int X)
{
   if (N==0) return 0;
   else
   {
     if (T[N-1] == X) return g(T,N-1,X)+1;
     else return g(T,N-1,X);
   }
}
```

Exercice 5 : (tableaux et récursivité)

1. La fonction *image* :

```
int i(0);
void image(int *t, int s){
  if (i>s/2) return;
  else { int tr;
         tr=t[s-i]; t[s-i]=t[i]; t[i]=tr;
         i++;
         image(t,s);}
}
```

2. Le *main* :

```
main(){
  int taille;
  cout << "taille="; cin>>taille;
  int *tab=new int[taille];
  for(int i(0); i<taille; i++)
  {
        cout << "tab[" <<i<<"]=";
        cin >>tab[i];
  }
  image(tab,taille-1);
  for(int i(0); i<taille; i++)
         cout << "tab[" <<i<<"]="<<tab[i]<< endl;
  delete []tab;
}
```

Exercice 6 : (fonction récursive)

1. La fonction permet de compter le nombre de chiffres qui figurent dans une chaîne de caractères.

Initialement c=0 :
f("0a1d3")
|
c=1
f("a1d3")
|
f("1d3")
|
c=2
f("d3")
|
f("3")
c=3
|
f("")
↓

2. La version itérative :

```
unsigned int c(0);
int f(char * X)
{
    while(*X!='\0') {
     if (*X>='0' && *X<='9') c++;
     X=X+1;
  }
  return c;
}
```

3. Le nouvel entête de la fonction *f* : *void f(char * X, int &c)*.

Exercice 7 : (arbre d'exécution)

1. l'arbre d'exécution de *g(5)* :

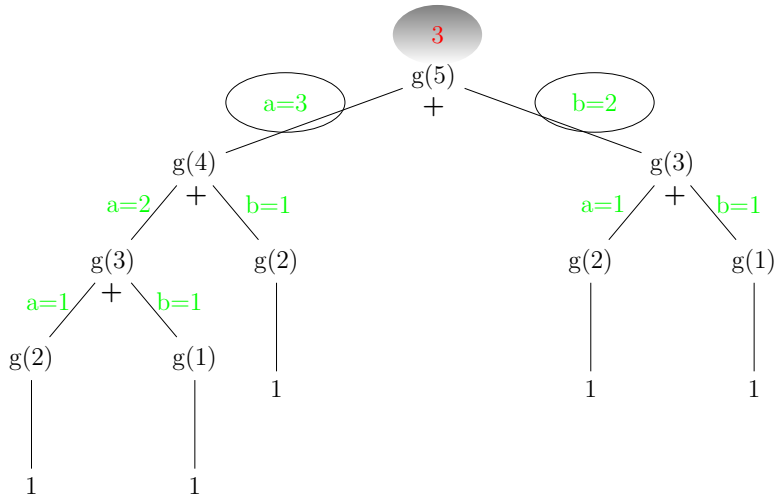


FIGURE 1 – Arbre d'exécution de *g(5)* avec *a* et *b* globales

2. les différentes valeurs assignées aux variables *a* et *b* :

   a=1 → 2 → 3 → 1

   b=1 → 1 → 1 → 2

   g(5)=1+2=3

Si on déclare *a* et *b* comme variables locales : *g(5)=5*.

Exercice 8 : (fonction récursive et arbre d'exécution)

1. La fonction récursive *int transform(char *, int)* :

```c
#include <math.h>
int transform(char * c, int l)
{
        if (l==1) return *c-'0';
        else return (*c-'0')*((int)pow(10,l-1))
                                +transform(c+1,l-1);
}
```

2. L'arbre d'exécution de *transform* appliquée sur la chaîne de caractères "5269" :

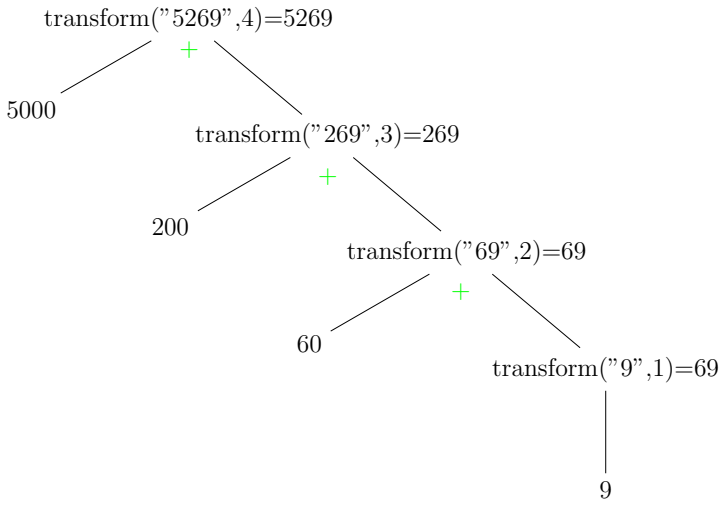


FIGURE 2 – Arbre d'exécution de *transform(5269,4)*

Exercice 9 : (fonction récursive et arbre d'exécution)

1. La fonction *sum* :

```c
int sum(int nbr)
{
    if (nbr==0) return 0;
    else
        return (nbr%10)+sum(nbr/10);
}
```

2. L'arbre d'exécution de *sum(358)* :

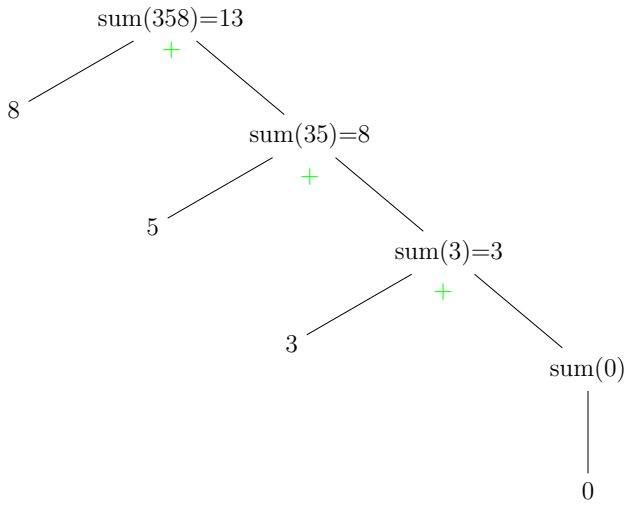


FIGURE 3 – Arbre d'exécution de *sum(358)*

Exercice 10 : (fonction de MORIS)

Nous donnons ci-dessous la figure 4 l'arbre d'exécution correspondant à l'appel de la fonction *moris(1,0)*.

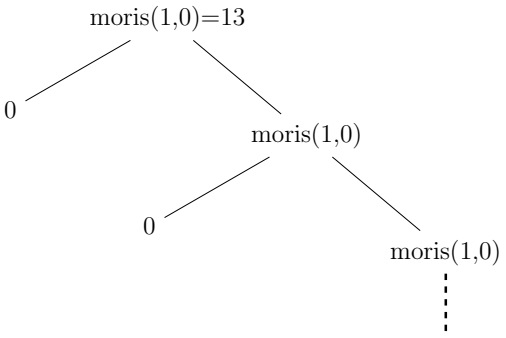


FIGURE 4 – Arbre d'exécution de *moris(1,0)*

A partir de l'arbre d'exécution, on peut conclure que *moris(1,0)* fait toujours appel à *moris(1,0)* d'une manière infinie ce qui montre que *moris(1,0)* ne se termine jamais.

Exercice 11 : (fonction de SYRACUSE)

```
int syracuse(int n, int m)
{
    if(n==0) return m;
    else
      if (syracuse(n-1,m)\%2) return (syracuse(n-1,m)*3+1);
      else return (syracuse(n-1,m)/2);
}
```

Exercice 12 : (récursivité-itérativité)

1. La fonction *puissance* :

```
int puissance(int n)
{
  if (n<10) return 10;
  else return (10*puissance(n/10));
}
```

2. La fonction *miroir* :

```
int miroir(int n)
{
  if (n<10) return n;
  else
  {
    return(puissance(n/10)*(n%10)+miroir(n/10));
  }
}
```

3. La version itérative de chacune des fonctions :

   La fonction *puissance* :

```
int p;
int puissance(int n)
{
  if (n<10) return 10;
  else {
         int p=10; n=(n/10);
         while(n>=10)
         {
          p=p*10;
          n=n/10;
         }
        return p*10;
       }
}
```

La fonction *miroir* :

```
int miroir(int n)
{
  if (n<10) return n;
  else
  {
    int p=puissance(n/10)*(n%10); n=n/10;
    while(n>=10)
    {
      p=p+puissance(n/10)*(n%10);
      n=n/10;
    }
    return(p+n);
  }
}
```

Exercice 13 : (trace d'une fonction récursive)

— La fonction permet de vérifier si x et pair ou impair.
— Pour justifier, on trace les arbres d'exécution suivants :

```
g(3)          g(2)
 |             |
g(1)    et    g(0)
 |             |
false         true
```

4

COMPLEXITÉ ALGORITHMIQUE

4.1 EXERCICES

Exercice 1 : (notions de base)[Baynat et al., 2007]

1. Démontrer que :
   $n^2 \in O(10^{-5}n^3)$
   $25n^4 - 19n^3 + 13n^2 \in O(n^4)$
   $2^{n+100} \in O(2^n)$
2. Soit un ordinateur pour lequel toute instruction possède une durée de 10^{-6} secondes. On exécute un algorithme qui utilise, pour une donnée de taille n, $f(n)$ instructions, $f(n)$ étant l'une des fonctions $(logn, nlogn, n, n^2, n^3$ et $2^n)$. Remplir un tableau qui donne, en fonction de la taille n = 10 et 60, et de la fonction $f(n)$, la durée d'exécution de l'algorithme.
3. Soient f, g, S et $T : \mathbb{N} \to \mathbb{N}$:
   a) Montrer que si $f(n) \in O(g(n))$, alors $f(n) + g(n) \in O(g(n))$.
   b) Montrer que $f(n) + g(n) \in \Theta(max(f(n), g(n)))$.

Exercice 2 : (comparaison entre complexités)

Considérons deux algorithmes A_1 et A_2 pour un même problème de taille N, avec $C_{A_1}(N) = 2N$ et $C_{A_2}(N) = 0.5N^2$.

1. Tracez les courbes correspondant à C_{A_1} et C_{A_2}.
2. Que peut-on dire sur ces deux complexités ?
3. Donnez la relation d'ordre existant entre C_{A_1} et C_{A_2} ? justifiez.

Exercice 3 : (calcul de la complexité)[Baynat et al., 2007]

Déterminer la complexité des algorithmes suivants (par rapport au nombre d'itérations effectuées), où m et n sont deux entiers positifs.

Algorithme 1 :

```
i := 1; j := 1;
tant que (i <= m) et ( j <= n) faire
          i := i + 1;
          j := j + 1;
fin tant que
```

Algorithme 2 :

```
i := 1; j := 1;
tant que (i <= m) ou ( j <= n) faire
          i := i + 1;
          j := j + 1;
fin tant que
```

Exercice 4 : (calcul de la complexité)

Soit la question 1) de l'exercice 2 du chapitre *Techniques de programmation* C++ 2.1.

1. Calculez la complexité asymptotique de cet algorithme.
2. Discutez cette complexité.

Exercice 5 : (trace et complexité d'une fonction récursive)[Baynat et al., 2007]

Soit la fonction récursive F d'un paramètre n suivante :

```
int F(int n){
    if (n==0) return 2;
    else return (F(n-1)*F(n-1));
}
```

1. Que calcule cette fonction ? Justifiez.
2. Déterminez la complexité de la fonction F. Comment améliorer cette complexité ?

Exercice 6 : (calcul de la complexité)

Soit le code source L suivant :

```
int x=0;
int p1,p2;
if (a>b)
    for(int i=0; i<=n; i++)
        x=x+p1;
else
     x=x+p1;
cout << x*x << endl;
```

Calculez $T_L(n)$ (le nombre d'opérations fondamentales), sachant que les opérations arithmétiques et les opérations logiques constituent les opérations fondamentales.

Exercice 7 : (calcul de la complexité)

1. Étudiez le nombre d'additions réalisées, au pire des cas, par le listing L suivant :

```
if (cond)
   for(int i=1; i<=n; i++)
       if (T[i]>a) s = s + T[i];
else
   if (a > b) for(int i=1;i<=n;i++) x=x+a;
   else x=x+b;
```

2. Répondez, sans justifier, par *vrai* ou *faux* :
   a) $C(n) = n!$ est une complexité exponentielle.
   b) $C(n) = n^4$ est une bonne complexité.
   c) $n^2 \in O(n.log(n))$.

Exercice 8 : (calcul des trois types de complexité)

Soit le programme A ci-dessous :

```
int a,b,n;
...
int s(0);
for(int i=2; i<n){
    if (a>b)     {s+=a-b; a=a/i; b=b/i;}
    else {
            if (a=b){s+=a*a; a=a/i;}
             }
        else { s+=(b-a)/i; a=a*i; b=b*i;}
}
```

Donnez les trois types de complexités de A, sachant que les opérations fondamentales à prendre en considération sont : les opérations arithmétiques et l'opération d'affectation.
NB : *On suppose que la distribution est uniforme.*

Exercice 9 : (calcul des trois types de complexité)

Soit le programme en C++ suivant :

```
if (a>b)
     for(int i=5; i<=n-5;i++)
       for(int j=i-5; j<=i+5; j++)
          x=x+3;
else
     for(int i=1; i<=n;i++)
        for(int j=1; j<=n; j++)
            for(int k=1; k<=n; k++)
               x=x+b;
```

Étudiez le nombre d'additions réalisées par le programme ci-dessus dans le meilleur cas, le pire cas, puis dans le cas moyen en supposant que les tests ont une probabilité de $\frac{1}{2}$ d'être vrai.

Exercice 10 : (calcul des trois types de complexité)

Soit le listing suivant :

```
int x=0;
int p1,p2;
if (a>b)
        for(int i=0; i<n; i++)
                for(int j=0; j<n; j++){
                        p1=i+j;
                        x=x+p1;
                }
else
        if (a==b)
          for(int i=7; i<=n-7; i++)
                for(int j=i-7; j<=i+7; j++){
                        p1=i*j;
                        x=x+p1;
                }
        else
                for(int i=1; i<=n; i++)
                        for(int j=1; j<=i; j++)
                                for(int k=1; k<=j; k++){
                                        p1=i*j;
                                        p2=p1/k;
                                        x=x+p2;
                                }
```

Calculez le nombre d'opérations arithmétiques réalisées par le listing ci-dessus :

1. au meilleur des cas,
2. au pire des cas,
3. au moyen des cas, en supposant que les tests ont une probabilité de $\frac{1}{3}$ d'être vrai.

Exercice 11 : (calcul des trois types de complexité)

Soit le programme P suivant :

```
int b,i=1,j=1;
cout << "b="; cin >> b;
if (b>0)
        while ((i <= m) && ( j <= n)){
                i = i + 1;
                j = j + 1;
        }
else
        while ((i <= m) || ( j <= n)){
                i = i + 1;
                j = j + 1;
        }
```

En ne prenant en considération que les opérations arithmétiques, donner le nombre d'opérations fondamentales exécutées par P :

1. dans le meilleur des cas,
2. dans le pire des cas,
3. dans le moyen des cas, avec une distribution uniforme de la probabilité des deux cas : $b > 0$ et $b \leqslant 0$.

4.2 SOLUTIONS

Exercice 1 : (notions de base)

1. Démonstrations :

$$\lim_{n\to\infty} \frac{n^2}{10^{-5}n^3} = \lim_{n\to\infty} \frac{1,}{10^{-5}n} = 0 \Rightarrow n^2 \in O(10^{-5}n^3)$$

$$\lim_{n\to\infty} \frac{25n^4 - 19n^3 + 13n^2}{n^4} = \lim_{n\to\infty} \frac{25n^4}{n^4} = 25 \Rightarrow 25n^4 - 19n^3 + 13n^2 \in O(n^4)$$

$$\lim_{n\to\infty} \frac{2^{n+100}}{2^n} = \lim_{n\to\infty} \frac{2^n.2^{100}}{2^n} = 2^{100} \Rightarrow 2^{n+100} \in O(2^n)$$

2. Le tableau correspondant à $f(n)$:

$f(n)$	10	20
$log\ n$	2.30×10^{-6}	4.09×10^{-6}
$n\ log\ n$	2.30×10^{-5}	2.46×10^{-4}
n	10^{-5}	6×10^{-5}
n^2	10^{-4}	3.60×10^{-3}
n^3	10^{-3}	2.16×10^{-1}
2^n	1.02×10^{-3}	1.15×10^{12}

3. Soient f, g, S et $T : \mathbb{N} \to \mathbb{N}$:
   a) si $f(n) \in O(g(n))$, alors $f(n) + g(n) \in O(g(n))$.

$$f(n) \in O(g(n)) \Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$\Rightarrow \frac{f(n)}{g(n)} + \frac{g(n)}{g(n)} = 1 \Rightarrow \lim_{n\to\infty} \frac{f(n)+g(n)}{g(n)} = 1$$

$$\Rightarrow f(n) + g(n) \in O(g(n))$$

   b) $f(n) + g(n) \in \Theta(max(f(n), g(n)))$

   <u>cas 1</u> : $max(f(n), g(n)) = f(n)$

$$\lim_{n\to\infty} \frac{f(n)+g(n)}{f(n)} = \lim_{n\to\infty} (1 + \frac{g(n)}{f(n)}) = 1 + c_1 = c_1' \quad (1)$$

   <u>cas 2</u> : $max(f(n), g(n)) = g(n)$

$$\lim_{n\to\infty} \frac{f(n)+g(n)}{g(n)} = \lim_{n\to\infty} (1 + \frac{f(n)}{g(n)}) = 1 + c_2 = c_2' \quad (2)$$

   $(1) \wedge (2) \Rightarrow f(n) + g(n) \in \Theta(max(f(n), g(n)))$

Exercice 2 : (comparaison entre complexités)

1. Courbes correspondant à C_{A_1} et C_{A_2} (voir la figure 5) :

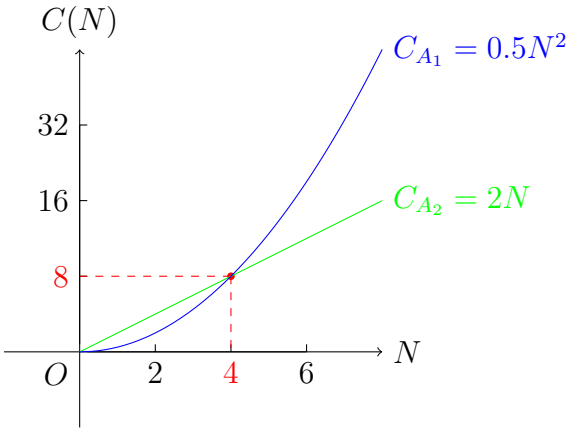


FIGURE 5 – Les courbes $C_{A_1}(N) = 2N$ et $C_{A_2}(N) = 0.5N^2$

2. Analyse des deux complexités :
   a) A_1 fait plus d'opérations que A_2 pour $N < 4$,
   b) dès que $N \geq 4$ A_1 fait moins d'opérations que A_2.

donc on peut dire que l'algorithme A_1 est meilleur que l'algorithme A_2.

3. Relation d'ordre entre C_{A_1} et C_{A_2} :
   $2N \in O(0.5N^2)$ puisque :

$$\lim_{N\to\infty} \frac{2N}{0.5N^2} = \lim_{N\to\infty} \frac{4}{N} = 0$$

Exercice 3 : (calcul de la complexité)

Algorithme 1 : $C(m,n) = O(min(m,n))$

Algorithme 2 : $C(m,n) = O(max(m,n))$

Exercice 4 : (calcul de la complexité)

1. La complexité asymptotique :

   La première boucle B_1 : $T_{B_1}(n) = n$

   La deuxième boucle B_2 : $T_{B_2}(n, max) = n$,

   En supposant que $a_i = count\,[i]$, on aura :

   $T_{B_2}(n, max) = a_1 + a_2 + \cdots a_{max}$
   $a_{max} = n - (a_1 + a_2 + \cdots a_{max-1})$
   $a_{max-1} = n - (a_1 + a_2 + \cdots a_{max-2} + a_{max})$
   $\cdots$
   $a_2 = n - (a_1 + a_3 + \cdots a_{max})$
   $a_1 = n - (a_2 + a_3 + \cdots a_{max})$

   Donc, si on développe les a_i, on obtient :

   $T_{B_2}(n, max) = max * n - (max - 1) * (a_1 + a_2 + \cdots a_{max}) = max * n - (max - 1) * n) = n$

   Ce qui implique : $C_{prog}(n) = \Theta(2n) = \Theta(n)$
2. La complexité est linéaire.

Exercice 5 : (trace et complexité d'une fonction récursive)

1. Le rôle de la fonction F :
   Nous avons :

   $F(0) = 2 = 2^{2^0}$
   $F(1) = F(0) \times F(0) = 2 \times 2 = 2^2 = 2^{2^1}$
   $F(2) = F(1) \times F(1) = 2^2 \times 2^2 = 2^4 = 2^{2^2}$
   $F(3) = F(2) \times F(2) = 2^4 \times 2^4 = 2^8 = 2^{2^3}$
   $F(4) = F(3) \times F(3) = 2^8 \times 2^8 = 2^{16} = 2^{2^4}$
   $\cdots$

$F(n) = F(n-1) \times F(n-1) = 2^{2^{n-1}} \times 2^{2^{n-1}} = 2^{2^n}$

Donc, de façon intuitive, $F(n) = 2^{2^n}$.

2. La complexité de la fonction F :

   Soit $T_F(i)$ le nombre d'opérations arithmétiques (le nombre de multiplications) réalisées par $F(i)$. Par récurrence, nous avons :

   $T_F(0) = 0$

   $T_F(1) = 1 + 2 \times T_F(0) = 1 = 2^0$

   $T_F(2) = 1 + 2 \times T_F(1) = 1 + 2 = 2^0 + 2^1$

   $T_F(3) = 1 + 2 \times T_F(2) = 1 + 2 \times (1 + 2) = 2^0 + 2^1 + 2^2$

   $T_F(4) = 1 + 2 \times T_F(3) = 1 + 2 \times (1 + 2 \times (1 + 2)) = 2^0 + 2^1 + 2^2 + 2^3$

   $\ldots$

   $$T_F(n) = \cdots = 2^0 + 2^1 + 2^2 + 2^3 + \cdots 2^{n-1}$$

   Ce qui donne

   $$T_F(n) = \sum_{i=0}^{n-1} 2^i = 2^n - 1$$

   Par conséquent, $C_F(n) \in \Theta(2^n)$, ce qui correspond à une complexité exponentielle.

   Pour améliorer la complexité de F, il faut éviter le double appel récursif de la fonction, et pourquoi pas opter pour une version itérative.

Exercice 6 : (calcul de la complexité)

```
int x=0;
int p1,p2;
if (a>b)                        ----> 1
    for(int i=0; i<=n; i++)     ----> boucle de 0 à n=>n+1 itérations
        x=x+p1;                 ----> 1
else
    x=x+p1;                     ----> 1
cout << x*x << endl;            ----> 1
```

$$T_C(n) = 1 + max(\sum_{i=0}^{n} 1, 1) + 1 = 1 + max(n+1, 1) + 1 = 1 + n + 1 + 1 = n + 3$$

Exercice 7 : (calcul de la complexité)

1. Le nombre d'additions réalisées, au pire des cas par L :

   $T_L(n) = max(n, n) = n$

2. Répondez, sans justifier, par *vrai* ou *faux* :
   a) *vrai*
   b) *vrai*.
   c) *faux*.

Exercice 8 : (calcul des trois types de complexité)

Soit le programme B ci-dessous :

```
int a,b,n;
...
int s(0);                                        ---> 1 op
for(int i=2; i<n){
    if (a>b)     {s+=a-b; a=a/i; b=b/i;}         ---> 7 ops
    else {
            if (a=b){s+=a*a; a=a/i;}             ---> 5 ops
        }
        else { s+=(b-a)/i; a=a*i; b=b*i;} ---> 8 ops
}
```

Par conséquent on aura :

1. la complexité au meilleur des cas : $Min_B(n) = 1 + 5(n-2) = 5n - 9$
2. la complexité au pire des cas : $Max_B(n) = 1 + 8(n-2) = 8n - 15$
3. la complexité en moyenne : $Moy_B(n) = 1 + \frac{n-2}{3}(7+5+8) = 1 + \frac{20(n-2)}{3} = \frac{20}{3}n - \frac{37}{3}$

Exercice 9 : (calcul des trois types de complexité)

si $a > b$ le nombre d'additions est comme suit :

$$C1(n) = \sum_{i=5}^{n-5} \sum_{j=i-5}^{i+5} 1 = \sum_{i=5}^{n-5} 11 = 11 \sum_{i=5}^{n-5}$$

$$= 11(\sum_{i=1}^{n-5} - \sum_{i=1}^{4}) = 11(\sum_{i=1}^{n-5} -4) = 11(n-5-4) = 11(n-9)$$

sinon le nombre d'additions est le suivant :

$$C2(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} 1 = n.n.n = n^3$$

Ce qui donne :
— le meilleur cas : $Min_A(n) = 11(n-9)$
— le pire cas : $Max_A(n) = n^3$
— le cas moyen : $Moy_A(n) = \frac{1}{2}11(n-9) + \frac{1}{2}n^3$
$= \frac{1}{2}n^3 + \frac{11}{2}n - \frac{9}{2}$

Exercice 10 : (calcul des trois types de complexité)

Le nombre d'opérations arithmétiques réalisées par le listing :

BLOC 1 :

$$T1(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 2 = 2\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1 = 2\sum_{i=0}^{n-1} n = 2n^2$$

BLOC 2 :

$$T2(n) = \sum_{i=7}^{n-7}\sum_{j=i-7}^{i+7} 2 = 2\sum_{i=7}^{n-7}\sum_{j=i-7}^{i+7}$$

$$= 2\sum_{i=7}^{n-7}(i+7-(i-7)+1) = 2\sum_{i=7}^{n-7} 15$$

$$= 30\sum_{i=7}^{n-7} = 30(n-7-7+1) = 30(n-13)$$

BLOC 3 :

$$T3(n) = \sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=1}^{j} 3 = 3\sum_{i=1}^{n}\sum_{j=1}^{i} j$$

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

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

$$= \frac{3}{2}(\frac{(2n+1)(n+1)n}{6} + \frac{(n+1)n}{2})$$

$$= \frac{1}{4}(2n^3+6n^2+4n) = \frac{1}{2}n^3 + \frac{3}{2}n^2 + n$$

1. au meilleur des cas : $Min(n) = 30(n-13)$.
2. au pire des cas : $\frac{1}{2}n^3 + \frac{3}{2}n^2 + n$
3. au moyen des cas : $\frac{1}{3}(2n^2 + 30(n-13) + \frac{1}{2}n^3 + \frac{3}{2}n^2 + n) = \frac{1}{6}n^3 + \frac{7}{6}n^2 + \frac{31}{3}n - 130$

Exercice 11 : (calcul des trois types de complexité)

On étudie la complexité des deux cas :
— si $b > 0 : C_P(n,m) = 2.min(m,n)$,
— si $b <= 0 : C_P(n,m) = 2.max(m,n)$.
Par conséquent, on obtient les complexités suivantes :

1. dans le meilleur des cas :
$$\begin{aligned} Min_P(m,n) &= min(2.min(m,n), 2.max(m,n)) \\ &= 2.min(m,n) \in O(min(m,n)) \end{aligned}$$
2. dans le pire des cas :
$$\begin{aligned} Max_P(m,n) &= max(2.min(m,n), 2.max(m,n)) \\ &= 2.max(m,n) \in O(max(m,n)) \end{aligned}$$
3. dans le moyen des cas, avec une distribution uniforme de la probabilité des deux cas :

   Avec une distribution uniforme des deux cas, on a $P_{b>0} = \frac{1}{2}$ et $P_{b<=0} = \frac{1}{2}$.

   Par conséquent, on obtient la complexité en moyenne suivante :

   $Moy_P(n,m) = \frac{1}{2}(2.min(m,n)) + \frac{1}{2}(2.max(m,n))$
   $= min(m,n) + max(m,n) = m + n \Rightarrow Moy_P(n,m) = m + n$

5

STRUCTURES DE DONNÉES COMPLEXES

5.1 EXERCICES

Exercice 1 : (listes contiguës)

Soit L une liste linéaire contiguë de taille n et dans laquelle les éléments sont des entiers. Donnez en C++ :

1. la déclaration correspondante,
2. les fonctions *insert* et *remove*,
3. la fonction *symmetry* qui vérifie si le tableau est symétrique.
4. la fonction *reverse* qui renverse l'ordre des éléments de la liste,
5. la fonction *notrep* qui supprime tous les doublons.

Testez toutes les fonction dans un main.

Exercice 2 : (listes contiguës)

Pour représenter et manipuler en machine, une suite de chaînes de caractères qui peuvent comporter des espaces et dont la taille ne dépassent pas 20, on utilise le code C++ ci-dessous :

```
typedef char tch[50];
struct chaine{
        tch tab[50];
        unsigned int longueur;
};
```

On suppose que les primitives :
— *void insert(chaine &, int, tch)* qui insère un élément dans la liste,
— et *void remove(chaine &, int)* qui supprime un élément de la la liste,
sont prédéfinies.

Donnez en C++ :

1. la fonction *void read(chaine &, int)*, qui saisit les valeurs de la liste (le deuxième paramètre représente le nombre de valeurs à saisir). On doit obtenir une liste dont l'ordre des valeurs est conforme à l'ordre de saisit. Par exemple, si on saisie la suite des valeurs : "Info3", "cours de Info3", "devoir surveillé", on doit obtenir la liste contiguë suivante :

"Info3"	"cours de Info3"	"devoir surveillé"	...

2. la fonction *void remove_string(chaine &, tch)*, qui permet de supprimer les chaînes de caractères identiques à la chaîne de caractères reçue en entrée.

Exercice 3 : (listes simplement chaînées)

On considère le type *flist* représentant des listes simplement chaînées de flottants. Donnez en C++ :

1. la déclaration de la liste,
2. les différentes primitives applicables sur *flist*,
3. deux fonctions, l'une récursive et l'autre itérative, permettant de calculer la somme des éléments de la liste,
4. la fonction itérative permettant de supprimer toutes les occurrences d'un flottant f de la liste,
5. la fonction itérative permettant de supprimer la première occurrence d'un flottant f de la liste.
6. Soient deux listes simplement chaînées de flottants de type *flist*. En supposant qu'elles ne comportent pas de doublons, donnez une fonction itérative *IntersectIter* et une autre récursive *IntersectRec* permettant de déterminer une troisième liste résultat de l'intersection des deux premières.

Exercice 4 : (listes chaînées)

Pour représenter et structurer un texte en C++, on utilise une liste chaînée simple. Sachant qu'une ligne correspond à une chaîne de caractères qui peut contenir des espaces.

Soit la structure *line* qui permet de décrire une cellule de la liste chaînée. Chaque cellule doit définir : la chaîne de caractères définissant une ligne du texte, la taille de la chaîne de caractères et son nombre d'occurrences, comme suit :

```
struct line {
  char str[50];//la chaîne de caractères
  unsigned int taille;//la taille de la chaîne de caractères
  unsigned int nbr;//le nombre d'occurrences
  line*next;//pointeur sur la cellule suivante
};
line* tete; // l'en-tête de la liste
```

1. A partir de la définition donnée ci-dessus, donnez :
   a) la fonction *alloc* pour l'allocation de la mémoire pour une cellule de type *line*,
   b) la fonction *insert1* qui ajoute une cellule à l'entête de la liste,

c) la fonction *chfind* qui recherche une chaîne de caractères dans la liste et qui retourne l'adresse de la cellule,

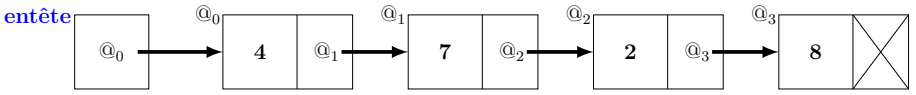
d) la fonction *insert2* qui incrémente le nombre d'occurrences *nbr* si la chaîne existe déjà et dans le cas contraire fait appel à *insert1*,

2. Écrivez un programme global qui lit sur l'entrée standard et qui construit la liste des lignes. Chaque cellule contient une des lignes, sa taille (nbre de caractères) et son nombre d'occurrences. A la fin, le programme affiche chaque ligne avec sa taille et son nombre d'occurrences,
3. Pour faciliter le *tri* des éléments de la liste chaînée, donnez :
   a) la structure *line*, avec un double chaînage,
   b) la fonction *sort* qui permet de trier la liste chaînée, par rapport au nombre d'occurrences, sans recopier ses éléments.

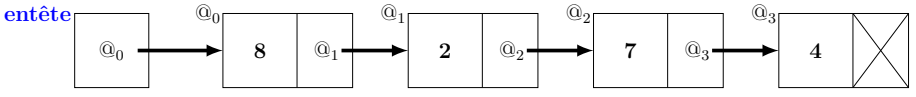
Exercice 5 : (listes simplement chaînées)

Pour représenter et manipuler en machine une suite d'entiers :

1. Donnez en C++ :
   a) la structure d'une liste chaînée simple *entier*, dont chaque maillon doit définir un élément de la suite,
   b) la fonction *int nbre_element(entier*)*, qui retourne le nombre d'éléments contenus dans la liste chaînée,
   c) la fonction *entier * adr_pos(entier*...,int pos)*, qui retourne l'adresse du maillon qui est à la position *pos* (la position varie entre 0 et la taille de la liste -1),
   d) la fonction *void reverse(entier*)*, qui inverse la liste (voir la figure 6).



(a) La liste en entrée



(b) La liste en sortie

FIGURE 6 – Application de la fonction *reverse* sur la liste

2. On suppose que les fonctions :
   — *void read(entier*&, int)* qui permet de saisir le contenu de la liste (le deuxième paramètre correspond au nombre de valeurs à saisir),
   — et *void display(entier*)* qui permet d'afficher le contenu de la liste.
   prédéfinies.
   Donnez le programme principal qui permet de tester les fonctions *read*, *display* et *reverse*.

Exercice 6 : (listes simplement chaînées)

Soit une liste simplement chaînée *list* définie comme suit :

```
struct list
{
    float elt;
    list * suivant;
};
list * tete=NULL;
```

Donnez en C++, les fonctions :

1. *... dernier(...)* qui retourne l'adresse du dernier élément de la liste,
2. *void insert(..., int n)* qui insert *n* flottants, et qui permet d'obtenir une liste chaînée avec les éléments positifs placés dans l'extrême gauche de la liste et les éléments strictement négatifs placés dans l'extrême droite. Ceci en faisant appel à la fonction *dernier* et les primitives d'insertion *insert_tete* et *insert_queue* supposées prédéfinies.
3. *... duplicate(...)* récursive qui vérifie si la liste contient des doublons.

Exercice 7 : (listes simplement chaînées)

Un rectangle est défini par sa couleur, sa hauteur et sa largeur, donnez en C++ :

1. la structure *ListRec* d'une liste chaînée, dans laquelle on pourra ranger les informations de plusieurs rectangles,
2. les fonctions :
   a) *Ajouter*, pour ajouter un élément au niveau de l'entête de la liste,
   b) *Supprimer*, pour supprimer un élément de la liste au niveau de l'entête,
   c) *Afficher*, pour afficher la couleur et la surface de chaque rectangle rangé dans la liste.
3. le programme principal, en faisant appel aux fonctions ci-dessus, qui permet de :
   — saisir et insérer les informations de 100 rectangles,
   — afficher la couleur et la surface de chaque rectangle de la liste,
   — supprimer les 10 derniers éléments insérés.

Exercice 8 : (listes doublement chaînées)

Soit la structure *individu* suivante :

```
struct individu
{
  string nom;
  string prenom;
  int age;
};
```

Donnez en C++ :

1. la déclaration d'une liste doublement chaînée dont les éléments sont de type *individu*,
2. les différentes primitives,
3. la fonction *inert_n* qui insère n éléments à l'entête de la liste,
4. la fonction *supprimer_age* qui permet de supprimer tous les éléments dont $age \geq 60$,
5. la fonction *ordonner* qui permet d'ordonner la liste dans l'ordre croissant par rapport au champ *age*,
6. la fonction *afficher* qui affiche le contenu de la liste,
7. le *main* qui permet de tester les fonctions *inert_n*, *inert_n* et *afficher*.

Exercice 9 : (piles)[Sedgewick, 1991]

En utilisant la structure et les primitives nécessaires, donner le contenu de la pile pour chaque opération de la suite *Q*UES***TI*ON*FAC***IL***E***. Chaque lettre provoque un empilement et chaque astérisque un dépilement.

Exercice 10 : (files)[Sedgewick, 1991]

En utilisant la structure et les primitives nécessaires, donner le contenu de la file pour chaque opération de la suite *Q*UES***TI*ON*FAC***IL***E***. Chaque lettre provoque un enfilement et chaque astérisque un défilement.

Exercice 11 : (piles)

L'analyse syntaxique est une phase de la compilation qui nécessite une pile. Par exemple, pour la reconnaissance des mots bien parenthésés, nous devons écrire un programme qui :
— accepte les mots comme (), ()(()) ou ((()())()) ;
— rejette les mots comme)(, ()(() ou ((()()))).
En utilisant les piles, donnez :

1. la structure de la pile ainsi que les fonctions *empiler* et *depiler*,
2. la fonction *correct* qui retourne « true » si une chaîne de caractères est bien parenthésée et « false » sinon,
3. le programme principal qui permet de saisir la chaîne de caractères et de tester la fonction *correct*.

Exercice 12 : (listes simplement chaînées et les piles)

On définit la structure *element* comme suivant :

```
struct element{
        unsigned int nbre;
        element * autre;
};
```

1. Soit une liste chaînée simple dont le maillon est de type *element* et l'entête :

```
element * tete = NULL;
```

   Donnez :

   a) les différentes primitives nécessaires pour : (1) la création du maillon, (2) l'insertion au niveau de l'entête, (3) la suppression au niveau de l'entête,

   b) la fonction *saisilist* qui permet de saisir et d'insérer 10 entiers positifs dans la liste chaînée (l'insertion doit se faire au niveau de l'en-tête).

2. Soit deux piles dont les éléments sont de type *element* et les sommets respectifs :

```
element *sommet1 = NULL;
element *sommet2 = NULL;
```

   Sachant que *sommet1* pointe sur le sommet de la pile des nombre paires et *sommet2* sur le sommet de la pile des nombres impaires. Donnez les fonctions :

   a) *empiler*,

   b) *void empiler12(element *&sommet1,element *&sommet2, unsigned int x)* qui doit empiler l'entier positif x dans la première pile s'il est paire sinon dans la deuxième pile.

3. Donnez la fonction d'entête *void listpile(element *& tete, element *& sommet1, element *& sommet2)* et qui permet de :
   — saisir les dix entiers positifs en faisant appel à *saisilist*,
   — parcourir la liste chaînée et charger les deux piles avec les valeurs convenues en faisant appel à la fonction *empiler12*.

Exercice 13 : (piles et files)

1. Donnez la fonction itérative *unsigned int nbchiffre(unsgned int)* qui retourne le nombre de chiffres contenus dans un nombre entier positif.
2. Soit une suite de chiffres. Donnez en C++ :

   a) la structure *pile* correspondante,

   b) les primitives *void empiler(pile*&, int)* et *int depiler(pile*&)*,

   c) la fonction *insertp* qui permet d'insérer les chiffres de gauche à droite dans la pile, sachant que l'entrée de la fonction est un nombre entier positif,

d) la structure *file* correspondante,

e) les primitives *void enfiler(file*&,file*&,int)* et *int defiler(file*&, file*&)*,

f) la fonction *insertf* qui permet d'insérer les chiffres de gauche à droite dans la file, sachant que l'entrée de la fonction est un nombre entier positif.

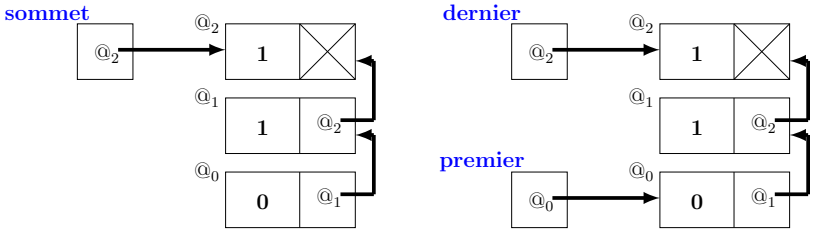
3. Un nombre palindrome est un nombre symétrique écrit dans une certaine base *a* comme ceci : $a_1 a_2 a_3 \cdots \mid \cdots a_3 a_2 a_1$ (exemple : 15851, 1661, etc.). En utilisant les structures *pile* et *file* et les fonctions citées ci-dessus, donnez la fonction *bool verify(...)* qui vérifie si un nombre est palindrome.
4. Testez toutes les fonctions réalisées ci-dessus dans main.

Exercice 14 : (piles et files)

Pour une suite de bits (0 ou 1) à déclarer de type entier, donnez en C++ :

1. les structures *pile* et *file* correspondantes (voir la figure 7),
2. la fonction *void CInsert(int N, pile *& ..., file *&..., file*&...)* qui permet d'empiler et d'enfiler le résultat de la conversion du nombre N en binaire (voir l'exemple 5.1.1). On suppose que les primitives *empiler* et *enfiler* sont déjà prédéfinies.

   Exemple 5.1.1 (conversion décimal-binaire) *Soit la conversion* $6_{(10)} = 110_{(2)}$. *Après application de la fonction CInsert sur le nombre* $N = 6$, *on obtient les structures illustrées dans la figure 7.*



(a) La pile (b) La file

FIGURE 7 – L'état de la pile et de la file après la conversion de $6_{(10)}$

3. les fonctions *int depiler(pile*&sommet)* et *int defiler(file*&premier, file*& dernier)* qui retournent respectivement, la valeur dépilée et la valeur défilée.
4. la fonction *int BinDeci(...)* qui reçoit en entrée une des deux structures et qui fournit le nombre en décimal. On suppose que le nombre de bits est non défini (Ex. $110_{(2)} = 1 * 2^2 + 1 * 2^1 + 0 * 2^0$).
5. le main qui permet de tester les fonctions *CInsert* et *BinDeci*.

Exercice 15 : (arbres)

Soit l'arbre binaire illustré dans la figure 8 ci-dessous :

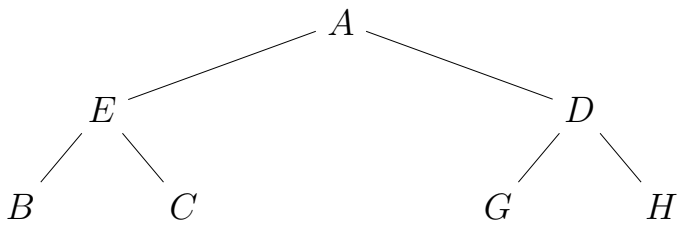


FIGURE 8 – Arbre binaire

Associer à chaque information de la colonne **1** (différents parcours d'un arbre binaire), une et une seule information de la colonne **2** (ordre de parcours des nœuds de l'arbre) :

Colonne 1	**Colonne 2**
(1) Parcours en profondeur préfixe	(a) B-E-C-A-G-D-H
(2) Parcours en profondeur infixe	(b) A-E-D-B-C-G-H
(3) Parcours en profondeur suffixe	(c) B-C-E-G-H-D-A
(4) Parcours en largeur	(d) A-E-B-C-D-G-H

Exercice 16 : (arbres binaires)

Un arbre binaire de recherche est tel que tout nœud a une valeur supérieure à celles des nœuds de son sous-arbre gauche et inférieure à celles des nœuds de son sous-arbre droit.

Exemple 5.1.2 (arbre de recherche) *Pour une liste aléatoire de 14 entiers : 25, 60, 35, 10, 5, 20, 65, 45, 70, 40, 50, 55, 30, 15, on obtient l'arbre de recherche illustré dans la figure 9.*

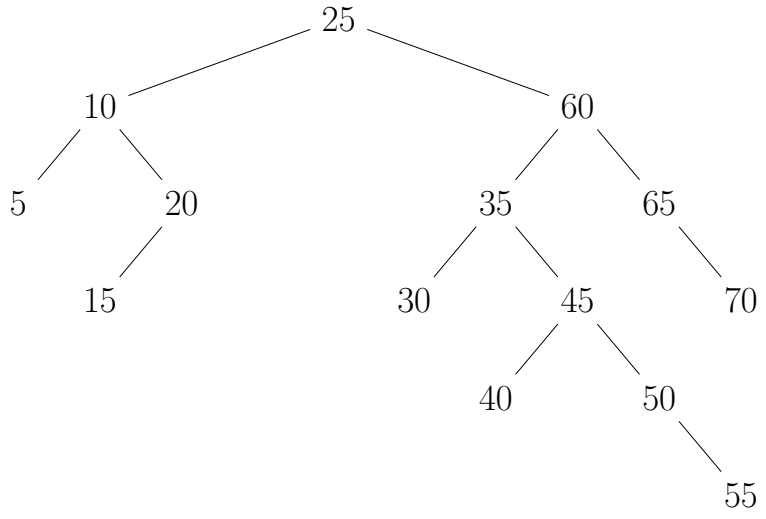


FIGURE 9 – Arbre binaire de recherche

Donnez :

1. la structure de l'arbre, sachant que chaque nœud porte une valeur de type entier,
2. les fonctions :
   — *create* qui permet de créer un nœud,
   — *insert* récursive qui insère un élément (une valeur entière) dans l'arbre en respectant le principe de l'arbre de recherche,
   — *hauteur* qui calcule la hauteur de l'arbre.
3. le programme principal qui permet de tester les fonctions *insert* et *hauteur*.

Exercice 17 : (arbres)

1. Soit l'arbre binaire illustré dans la figure 10 ci-dessous :

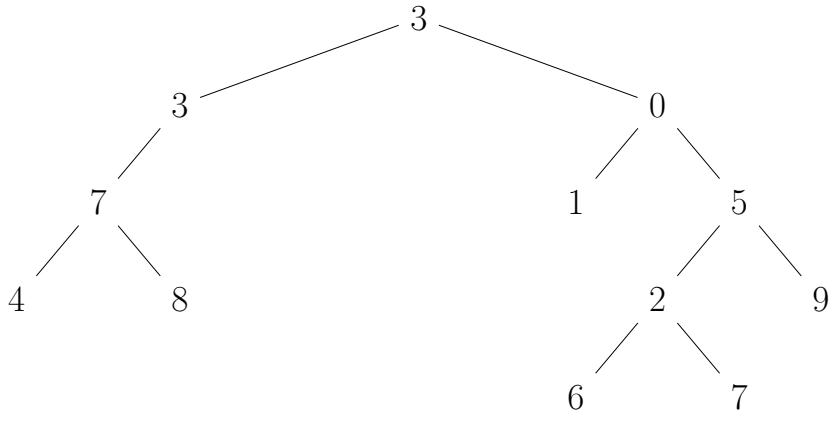


FIGURE 10 – Arbre binaire

   a) donnez la structure générale en C++ de l'arbre illustré dans la figure 10,
   b) citez l'algorithme de parcours en profondeur préfixe,
   c) appliquez cet algorithme sur l'arbre de la figure 10, et donnez l'ordre préfixe des nœuds (faire la trace de l'algorithme).

2. Donnez l'arbre binaire correspondant à l'arbre général de la figure 11 ci-dessous :

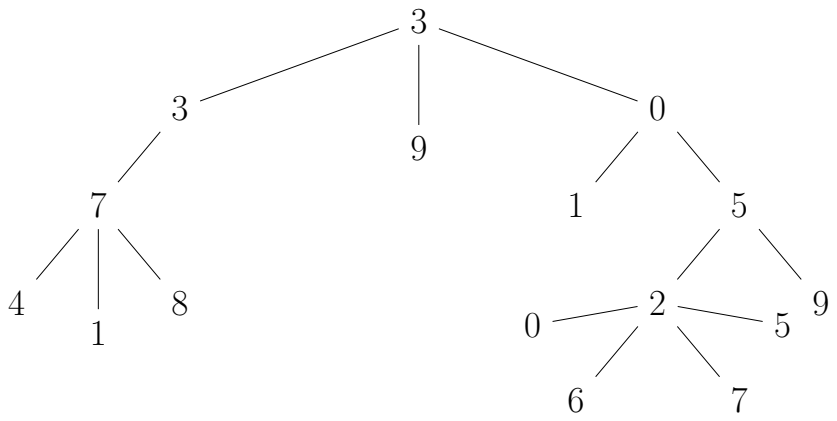


FIGURE 11 – Arbre général

Exercice 18 : (arbres binaires)

Soit deux arbres T_1 et T_2, dont les nœuds correspondent à des caractères. Donnez en C++ :

1. la structure *tree* correspondant avec *racine1* et *racine2* les racines respectives de T_1 et T_2,
2. la fonction *Hauteur* qui permet de calculer la hauteur d'un arbre ayant la structure *tree*,
3. la fonction *bool Idemtree(...)*, qui permet de vérifier si les deux arbres sont identiques. On utilise le parcours en profondeur préfixe.

Exercice 19 : (graphes)

Soit le graphe donné dans la figure 12 ci-dessous :

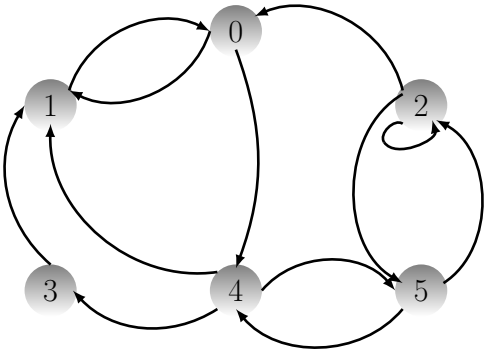


FIGURE 12 – Graphe orienté non valué

1. donnez en C++ les trois représentations possibles pour un graphe donné,
2. en utilisant la représentation avec la matrice d'incidence sommet-sommet, donner les fonctions :

a) *saisiegraph* qui saisit le graphe,

b) *ArcList* qui affiche tous les arcs du graphe,

c) *ArcExterieur* affiche tous les arcs incidents à un sommet donné vers l'extérieur,

d) *Chemin* qui vérifie s'il existe un chemin entre deux sommets x et y donnés.

3. testez les trois fonctions dans un *main* et faire la trace.

Exercice 20 : (fichiers)

1. Associez à chaque information de la colonne **A**, une et une seule information de la colonne **B** :

Colonne A	**Colonne B**
(1) ifstream	(a) écrire un caractère
(2) ofstream	(b) ouvrir le fichier en écriture
(3) eof	(c) test de la fin du fichier
(4) getline	(d) lire un caractère
(5) put	(e) fermer le fichier
(6) get	(f) lire une ligne complète
(7) close	(g) ouvrir le fichier en lecture

2. Donnez un programme en C++ qui permet de saisir et mémoriser dans un fichier nommé *repertoire.txt*, le nom et le numéro de téléphone de 10 personnes,

Exercice 21 : (fichiers)

Complétez le programme ci-dessous :

```
#include<iostream>
using namespace std;
#include <cstring>
#include <stdio.h>
...
main(){
        ofstream flog("fphysique1.txt");
        log << "ES2\n" << "Semestre2\n" << "Info2\n";
        flog.close();
        ... ilog("fphysique1.txt");
        ... olog("fphysique2.txt");
        ... st;
        while(!ilog.eof()){
                getline(...,st); cout << st << endl;
                olog << st << endl;}
                ilog.close();
        ...
        }
```

Exercice 22 : (fichiers)

Donnez en C++ :

1. un programme qui permet de saisir et mémoriser dans un fichier nommé *composant.txt*, le nom et le symbole de 10 composants chimiques.
2. un programme qui permet d'afficher à l'écran le contenu du fichier *composant.txt*.

5.2 SOLUTIONS

Exercice 1 : (listes contiguës)

1. La déclaration correspondante :

```
const int taille=30;
struct L{
        int tab[taille];
        unsigned int longueur;
};
```

2. Les fonctions *insert* et *remove* :

```
void insert(L & lg, int val, int ind)
{
  unsigned int l=lg.longueur;
  if (l<taille-1 && ind<=l)
  {
    for(int i=l-1;i>=ind; i--)
      lg.tab[i+1]=lg.tab[i];
    lg.tab[ind]=val;
    lg.longueur=l+1;
  }
  else cout << "Insertion Impossible !" << endl;
}
void remove(L & lg, int ind)
{
  unsigned int l=lg.longueur;
  if (ind<=l-1)
  {
    for(int i=ind;i<l-1; i++)
      lg.tab[i]=lg.tab[i+1];
    lg.longueur = l-1;
  }
  else cout << "Suppression Impossible !" << endl;
}
```

3. La fonction *symmetry* :

```
bool symmetry(L & lg)
{
   for(int i=0; i<lg.longueur/2; i++)
      if (lg.tab[i]!=lg.tab[lg.longueur-i-1]) return false;
   return true;
}
```

4. La fonction *reverse* :

```
void reverse(L & lg)
{
   int temp;
   for(int i=0; i<lg.longueur/2; i++)
   {
               temp=lg.tab[i];
               lg.tab[i]=lg.tab[lg.longueur-i-1];
               lg.tab[lg.longueur-i-1]=temp;
   }
}
```

5. La fonction *notrep* :

```
void notrep(L & lg)
{
   for(int i=0; i<lg.longueur-1; i++)
      for(int j=i+1; j<lg.longueur;)
         if (lg.tab[i]==lg.tab[j]) remove(lg,j);
         else j++;
}
```

Le *main* :

```
#include <iostream>
using namespace std;
...
main()
{
    int val;
    unsigned int nb;

    L list;
    list.longueur = 0;
    cout << "Le nombre d'éléments à saisir = "; cin >> nb;
    for (int i=0; i<nb; i++){
      cout << "tab[" << i << "]="; cin >> val;
      insert(list,val,0);
    }
   cout << "------- la liste après insertion ------" << endl;
   for (int i=0; i<list.longueur; i++)
      cout << "tab[" << i << "]=" << list.tab[i] << endl;

   cout << "--------- symmetry ---------" << endl;
   if (symmetry(list)) cout <<"la liste est symétrique\n";
   else cout <<"la liste n'est pas symétrique\n";
   cout << "--------- reverse ---------" << endl;
   reverse(list);
   for (int i=0; i<list.longueur; i++)
      cout << "tab[" << i << "]=" << list.tab[i] << endl;
   cout << "--------- notrep -------" << endl;
   notrep(list);
   for (int i=0; i<l.longueur; i++)
      cout << "tab[" << i << "]=" << list.tab[i] << endl;
}
```

Exercice 2 : (listes contiguës)

```
#include<stdio.h>
#include<cstring>
#include<iostream>
using namespace std;
```

1. La fonction *void read(chaine & g, int)* :

```
void read(chaine & g, int nbr)
{
      tch v;
      for(int i=0;i<nbr;i++)
      {
         cout << "Chaine = ";
         gets(v);
         insert(g,i,v);
      }
}
```

2. La fonction *void remove_string(chaine &, tch)* :

```
void remove_string(chaine & g, tch val)
{
      for(int i=0;i<g.longueur; i++)
           if (strcmp(val,g.tab[i])==0) remove(g,i);
}
```

Exercice 3 : (listes simplement chaînées)

1. La déclaration de la liste :

```
struct flist{
        float elt;
        flist *suivant;
};
flist *tete=NULL;
```

2. Les différentes primitives :

Création d'un maillon

```
flist * creer_maillon(float val)
{
  flist * maillon = new flist;
  if (maillon) {
    maillon->elt = val;
    maillon->suivant = NULL;
  }
  return maillon;
}
```

Primitives d'insertion

```
void ajout_tete(flist * & tete, float val)
{
  flist *maillon=creer_maillon(val);
  if (maillon != NULL)
  {
    maillon->elt = val;
    maillon->suivant = tete;
    tete = maillon;
  }
}
void ajout_milieu(flist * pred, float val)
{
  flist *maillon=creer_maillon(val);
  if (maillon != NULL)
  {
    maillon->elt = val;
    maillon->suivant = pred->suivant;
    pred->suivant = maillon;
  }
}
void ajout_queue(flist * q, float val)
{
  flist *maillon=creer_maillon(val);
  if (maillon != NULL)
  {
    maillon->elt = val;
    maillon->suivant = NULL;
    q->suivant = maillon;
  }
}
```

Primitives de suppression

```
void supprimer_tete(flist *& tete)
{
  if (tete != NULL)
  {
    flist * p = tete;
    tete = p->suivant;
    delete p;
  }
}
void supprimer_milieu(flist * pred)
{
  if (pred->suivant != NULL)
  {
    flist * p = pred->suivant;
    pred->suivant = p->suivant;
    delete p;
  }
}
void supprimer_queue(flist * pred)
{
  if (pred->suivant != NULL)
  {
    flist * p = pred->suivant;
    pred->suivant = NULL;
    delete p;
    }
}
```

3. Les deux fonctions *SommeIter* et *SommeRec* :

```
float SommeIter(flist*tete)
{
        flist *p=tete;
        float s(0);
        while (p)
        {
                s+=p->elt;
                p=p->suivant;
        }
        return s;
}
```

```
float SommeRec(flist*t)
{
        if (!t) return 0;
        else return t->elt+SommeRec(t->suivant);
}
```

4. La fonction *SuppF* :

```
void SuppF(flist *& tete, float f)
{
        if (tete)
                while(tete->elt==f)
                        supprimer_tete(tete);
        if (tete){
                flist *p=tete;
                while (p->suivant->suivant)
                {
                  if (p->suivant->elt==f) supprimer_milieu(p);
                  p=p->suivant;
                }
                if (p->suivant->elt==f) supprimer_queue(p);
        }
}
```

5. La fonction *SuppM* :

```
void SuppM(flist *& tete, float n)
{
        if (tete)
        {
                if (tete->elt==n) supprimer_tete(tete);
                return;
        }
        if (tete){
                flist *p=tete;
                while (p->suivant->suivant)
                {
                        if (p->suivant->elt==n) {
                                supprimer_milieu(p);
                                return;
                        }
                        p=p->suivant;
                }
                if (p->suivant->elt==n) supprimer_queue(p);
        }
}
```

6. Les fonctions *IntersectIter* et *IntersectRec* :

Déclaration des trois listes

```
flist*tete1=NULL;
flist*tete2=NULL;
flist*tete3=NULL;
```

```
void IntersectIter(flist *tete1, flist *tete2,flist *&tete3)
{
        flist *p1=tete1;
        while (p1)
        {
                flist *p2=tete2;
                while (p2)
                {
                  if (p1->elt==p2->elt) ajout_tete(tete3,p1->elt);
                  p2=p2->suivant;
                }
                p1=p1->suivant;
        }
}
```

```
void IntersectRec(flist *p1, flist *p2,flist *&tete3)
{
        if (!p1) return;
        if (p1->elt==p2->elt) ajout_tete(tete3,p1->elt);
        if (!p2->suivant) IntersectRec(p1->suivant,tete2,tete3);
        else IntersectRec(p1,p2->suivant,tete3);
}
```

Exercice 4 : (listes simplement chaînées)

1. À partir de la définition :
   a) La fonction *alloc* :

```
line * alloc(char * c)
{
   line * l=new line;
   if (l)
   {
     strcpy(l->str, c);
     l->taille = strlen(c);
     l->nbr = 1;
     l->next;
   }
   return l;
}
```

   b) La fonction *insert1* :

```
void insert1(line *&tete, char *c)
{
    line * l= alloc(c);
    if(l!=NULL)
    {
       l->next = tete;
       tete = l;
    }
}
```

c) La fonction *chfind* :

```
line * chfind(line * tete, char *c)
{
    line * l = tete;
    while(l!=NULL)
    {
      if (strcmp(l->str, c)==0) return l;
      l=l->next;
    }
    return NULL;
}
```

d) La fonction *insert2* :

```
void insert2(line *& tete,char *c)
{
  line *l=chfind(tete,c);
  if (l!=NULL) l->nbr++;
  else insert1(tete,c);
}
```

2. Le *main* :

```
#include <iostream>
using namespace std;
#include <stdio.h>
#include <cstring>
...
main()
{
    char c[20];
    int nbchaine = 7;
    for(int i=0; i<nbchaine; i++)
     {
      cout << "chaine = "; gets(c);
      insert2(tete,c);
     }
    cout << "++++ Affichage du contenu de la liste ++++" << endl;
    line * l = tete;
    while(l!=NULL)
    {
      cout << "chaine = " << l->str << endl;
      cout << "occur = " << l->nbr << endl;
      cout << "taille = " << l->taille << endl;
      cout << "-----------------------------" << endl;
      l=l->next;
    }
}
```

3. Pour faciliter le *tri* :
   a) La structure *line* :

```
struct dline {
        char str[50];
        unsigned int taille;
        unsigned int nbr;
        dline *next;
        dline *before;
};
dline* tete;
```

   b) La fonction *sort* :
   À faire avec le chargé des travaux dirigés.

Exercice 5 : (listes simplement chaînées)

1. En C++ :
   a) La structure :

```
struct entier {
        int en;
        entier *next;
};
entier* tete=NULL;
```

b) La fonction *int nbre_element(entier*)* :

```
int nbre_element(entier*tete)
{
   entier *p=tete; int nbr(0);
   while(p!=NULL)
   {
      nbr++;
      p=p->next;
   }
   return nbr;
}
```

c) La fonction *entier * adr_pos(entier*...,int pos)* :

```
entier * adr_pos(entier *tete,int pos)
{
   entier *p=tete;
   for(int i=0; p!=NULL;i++)
   {
       if (i==pos) return p;
       p=p->next;
   }
   return p;
}
```

d) La fonction *void reverse(entier*)* :

```
void reverse(entier*tete)
{
    entier* pp, *p=tete;
    int taille=nbre_element(tete);
    int z;
    for(int i=0; i<taille/2; i++)
    {
       pp=adr_pos(tete,taille-1-i);
       z=p->en;
       p->en=pp->en;
       pp->en=z;
       p=p->next;
    }
}
```

2. Le *main* :

```
main()
{
    int nb;
    cout << "nb = "; cin>>nb;
    read(tete,nb);
    reverse(tete);
    display(tete);
}
```

Exercice 6 : (listes simplement chaînées)

1. La fonction ... *dernier(...)* :

```
list * dernier(list *tete)
{
        if (tete)
        {
                list *p=tete;
                while(p->suivant)
                        p=p->suivant;
                return p;
        }
        return NULL;
}
```

2. La fonction *void insert(..., int n)* :

```
void insert(list *&tete, int n)
{
        float v;
        for(int i=0; i<n; i++)
        {
                cout << "val " << i << " = ";
                cin >> v;
                if (v>=0) insert_tete(tete,v);
                else
                        if (tete) {
                                list * queue = dernier(tete);
                                insert_queue(queue,v);
                        }
                        else insert_tete(tete,v);
        }
}
```

3. La fonction récursive ... *duplicate(...)* :

```
bool duplicate(list *t)
{
        if (!t) return false;
        else
        {
                list *p=t->suivant;
                while(p)
                {
                        if (p->elt==t->elt) return true;
                        p=p->suivant;
                }
                return duplicate(t->suivant);
        }
}
```

Exercice 7 : (listes simplement chaînées)

1. La structure *ListRec* :

```
struct  ListRec{
                string couleur;
                float hauteur;
                float largeur;
                ListRec *next;
};
struct ListRec* tete;
```

2. Les fonctions :

   a) *Ajouter* :

```
ListRec * Ajouter(ListRec *& tete,string c,float h, float l)
{
    ListRec *lr = new ListRec;
    if (!lr) cout << "Plus de mémoire" << endl;
    else{
                lr->couleur = c;
            lr->hauteur = h;
                lr->largeur = l;
                lr->next = tete;
                tete = lr;
    }
}
```

   b) *Supprimer* :

```
ListRec * Supprimer(ListRec *&tete)
{
        ListRec *lr = tete;
        tete = lr->next;
        delete lr;
}
```

c) *Afficher* :

```
void Afficher(ListRec * tete)
{
        ListRec *lr = tete;
        while(lr!=NULL)
        {
                cout << "Couleur = " << lr->couleur;
                cout << "Surface = " << lr->hauteur * lr->largeur;
        }
}
```

3. Le *main* :

```
#include <iostream>
#include <string.h>
using namespace std;
...
main( )
{
        tete = NULL;
        string c;
        float h,l;
        for(int i=0; i<100; i++)
        {
                cout << "Couleur = " ; cin >> c;
                cout << "Hauteur = "; cin >> h;
                cout << "Largeur = "; cin >> l;
                Ajouter(tete,c,h,l);
        }
        Afficher(tete);
        for(int i=0; i<10; i++)
        {
                Supprimer(tete);
        }
}
```

Exercice 8 : (listes doublement chaînées)

1. La déclaration de la liste doublement chaînée :

```
struct dlist{
        individu prs;
        dlist * precedent;
        dlist * suivant;
};
dlist * tete=NULL;
```

2. Les différentes primitives :

Création d'un maillon

```
dlist * ceer_maillon(individu idv)
{
   dlist * l=new dlist;
   if (l)
   {
     l->prs = idv;
     l->precedent = NULL;
     l->suivant = NULL;
   }
   return l;
}
```

Primitives d'insertion

```
void ajout_tete(dlist * & tete, individu idv)
{
        dlist * l=ceer_maillon(idv);
        if (l != NULL)
        {
                l->suivant = tete;
                l->precedent = NULL;
                if(tete) tete->precedent = l;
                tete = l;
        }
}
void ajout_milieu(dlist * pred, individu idv)
{
        dlist * l=ceer_maillon(idv);
        if (l != NULL)
        {
                l->suivant = pred->suivant;
                l->precedent = pred;
                pred->suivant->precedent = l;
                pred->suivant = l;
        }
}
void ajout_queue(dlist * pred, individu idv)
{
        dlist * l=ceer_maillon(idv);
        if (l != NULL)
        {
                l->precedent = pred;
                l->suivant = NULL;
                pred->suivant = l;
        }
}
```

Primitives de suppression

```
void supprimer_tete(dlist * & tete)
{
        if (tete != NULL)
        {
                dlist * p = tete;
                tete = p->suivant;
                p->precedent = NULL;
                delete p;
        }
}
void supprimer_milieu(dlist * p)
{
        p->precedent->suivant = p->suivant;
        p->suivant->precedent = p->precedent;
        delete p;
}
void supprimer_queue(dlist * p)
{
        p->precedent->suivant = NULL;
        delete p;
}
```

3. La fonction *insert_n* :

```
void insert_n(dlist *& tete,int n)
{
        individu val;
    for(int i(0);i<n;i++)
    {
        cout << "----INDIVIDU "<< i<<"----" << endl;
                cout << "NOM = ";
                cin >> val.nom;
        cout << "PRENOM = ";
                cin >> val.prenom;
                cout << "AGE = ";
                cin >> val.age;
                ajout_tete(tete,val);
     }
}
```

4. La fonction *supprimer_age* :

```
void supprimer_age(dlist * & tete)
{
        if (tete)
                while(tete->prs.age>=60)
                        supprimer_tete(tete);
        dlist * p=tete;
        if (p)
        {
                while(p->suivant)
                {
                        if (p->prs.age>=60) supprimer_milieu(p);
                        p=p->suivant;
                }
                if (p->prs.age>=60) supprimer_queue(p);
        }
}
```

5. La fonction *ordonner* :

```
void ordonner(dlist *& tete)
{
        dlist * p=tete;
        individu inter;
        if (p)
        {
                while(p->suivant)
                {
                        if (p->prs.age > p->suivant->prs.age)
                        {
                                inter=p->prs;
                                p->prs=p->suivant->prs;
                                p->suivant->prs=inter;
                        }
                        p=p->suivant;
                }
        }
}
```

6. La fonction *afficher* :

```
void afficher(dlist *& tete)
{
        dlist * p=tete;
        while(p)
        {
                cout << "NOM = "<< p->prs.nom << endl;
                cout << "PRENOM = "<< p->prs.prenom << endl;
                cout << "AGE = " << p->prs.age <<endl;
                p=p->suivant;
        }

}
```

7. Le *main* :

```
main()
{
    int m;
    cout << "Nombre d'éléments à insérer :";
    cin >> m;
    insert_n(tete,m);
    cout << "----APRES INSERTION----" << endl;
    afficher(tete);
    supprimer_age(tete);
    cout << "----APRES SUPPRESSION----" << endl;
    afficher(tete);
    cout << "----LA LISTE ORDONEE----" << endl;
    ordonner(tete);
    afficher(tete);
}
```

Exercice 9 : (piles)

La structure et les primitives

```
struct cellule
{
  char cc;
  cellule *next;
};
cellule* sommet = NULL;

cellule* cellnew(char c)
{
  cellule* cel = new cellule;
  if (cel == 0) {
  cout << "Plus de mémoire" << endl;
  exit(1);
 };
  cel->cc = c;
  return cel;
}
void empiler(cellule * &sommet, char c)
{
  cellule* cel = cellnew(c);
  cel->next = sommet;
  sommet = cel;
}
void depiler(cellule * &sommet)
{
  if (sommet)
  {
   cellule * cel = sommet;
   sommet = sommet->next;
   delete cel;
  }
  else cout << "pile vide !" << endl;
}
```

Le main

```
#include <iostream>
#include <stdlib.h>
using namespace std;
main()
{
        char ch[100];
        char *pch=ch;
        cout << "Donnez une une chaîne de caractères : ";
        cin >> ch;
        while (*pch!='\0')
        {
                if (*pch!='*') empiler(sommet,*pch);
                else depiler(sommet);
                pch++;
        }
        if (sommet) cout << "La pile n'est pas vide\n";
        else cout << "La pile est vide\n";
}
```

Exercice 10 : (files)

La structure et les primitives

```
struct file{
        char cc;
        file *lien;
};
file * dernier=NULL;
file * premier=NULL;

file* cellnew(char c)
{
        file* f = new file;
        if (f == 0) {
                cout << "Plus de mémoire" << endl;
                exit(1);
        };
        f->lien=NULL;
        f->cc = c;
        return f;
}
void enfiler(file *&premier,file *&dernier, char c)
{
        file *f=cellnew(c);
        if (dernier==NULL) {
                premier=f;
        }
        else dernier->lien = f;
        dernier = f;
}
void defiler(file *& premier,file *& dernier)
{
        if (premier == NULL) return;
        else
        {
                if (premier==dernier) dernier=NULL;
                file *f = premier;
                premier = f->lien;
                delete f;
        }
}
```

Le main

```
#include <iostream>
#include <stdlib.h>
using namespace std;
main()
{
        char ch[100];
        char *pch=ch;
        cout << "Donnez une une chaîne de caractères : ";
        cin >> ch;
        while (*pch!='\0')
        {
                if (*pch!='*') enfiler(premier,dernier,*pch);
                else defiler(premier,dernier);
                pch++;
        }
        if (premier) cout << "La file n'est pas vide\n";
        else cout << "La file est vide\n";
}
```

Exercice 11 : (piles)

1. La structure et les fonctions *empiler* et *depiler* :

```
#include <iostream>
#include <stdlib.h>
using namespace std;
struct cellule
{
        char cc;
        cellule *next;
};
cellule* sommet = NULL;
cellule* cellnew(char c)
{
        cellule* cel = new cellule;
        if (cel == 0) {
                cout << "Plus de mémoire" << endl;
                exit(1);
        };
        cel->cc = c;
        return cel;
}
void empiler(cellule * &l, char c)
{
        cellule* cel = cellnew(c);
        cel->next = l;
        l = cel;
}
void depiler(cellule * &l)
{
        if (l)
        {
                cellule * cel = l;
                cout << "caractère dépilé : " << l->cc << endl;
                l = l->next;
                delete cel;
        }
        else cout << "pile vide !" << endl;
}
```

2. La fonction *correct* :

```
bool correct(cellule * &s, char *t)
{
        for (int i=0; t[i]!='\0'; i++)
        {
                if (t[i] == ')')
                if (!s) return false;
                else depiler(s);
            else
                if (t[i] == '(') empiler(s, t[i]);
        }
        if (s) return false;
        else return true;
}
```

3. Le *main* :

```
#include <iostream>
#include <stdlib.h>
using namespace std;
...
main()
{
        char t[30];
        cout << "Donnez une chaîne de caractères = ";
        cin >> t;
        bool cr = correct(sommet, t);
        if (cr) cout << "la chaîne de caractères est
                correcte" << endl;
        else
                cout << "la chaîne de caractères est incorrecte"
                << endl;
}
```

Exercice 12 : (listes simplement chaînées et piles)

1. La liste chaînée :
   a) Les primitives nécessaires :

```
element* create(unsigned int x)
{
        element * e = new element;
        e->nbre = x;
        e->autre = NULL
        return e;
}
void insert(element *& tete, unsigned int x)
{
        element * e= create(x);
        e->autre = tete;
        tete = e;
}
void remove(element *& tete)
{
        element * e= tete;
        tete = e->autre;
        delete e;
}
```

b) La fonction *saisilist* :

```
void saisilist(element *& tete)
{
        unsigned int d; element *e = tete;
        for(int i=0; i<10; i++) {
                cout << "entier ="; cin >> d;
                insert(tete, d);
        }
}
```

2. Les deux fonctions d'empilement :

a) La fonction *empiler* :

```
void empiler(element *& sommet, unsigned int x)
{
        element * e= create(x);
        e->autre = sommet;
        sommet = e;
}
```

b) La fonction *empiler12* :

```
void empiler12(element *& sommet1, element *& sommet2,
                                    unsigned int x)
{
        if (x%2==0) empiler(sommet1,x);
        else empiler(sommet2,x);
}
```

3. La fonction *listpile* :

```
void listpile(elemnt *& tete, element *& sommet1,
                          element *& sommet2)
{
        saisilist(tete);
        element *p=tete;
        while(p!=NULL){
                empiler(sommet1, sommet2, p->nbre);
                p = p->autre;
        }
}
```

Exercice 13 : (piles et files)

1. La fonction *unsigned int nbchiffre(unsigned int)* :

```
int nbchiffre(unsigned int nb)
{
        static int s(0);
        if (nb<10) return s+1;
        else return s+nbchiffre(nb/10);
}
```

2. a) La structure *pile* :

```
struct pile{
        int chiffre;
        pile *lien;
};
pile *sommet=NULL;
```

   b) Les primitives *void empiler(pile*&, int)* et *int depiler(pile*&)* :

```
pile * empiler(pile *&sommet, int x)
{
        pile *p=new pile;
        p->chiffre = x;
        p->lien = sommet;
        sommet = p;
}

int depiler(pile *&sommet)
{
        if (sommet){
                pile * p=sommet;
                int t=p->chiffre;
                sommet = sommet->lien;
                delete p;
                return t;
        } else return -1;
}
```

c) La fonction *insertp* :

```
void insertp(pile *& sommet, int nbre)
{
        int x=nbre, y;
        while (x>=10)
        {
                y=x%10;
                x=x/10;
                empiler(sommet,y);
        }
        empiler(sommet,x);
}
```

d) La structure *file* :

```
struct file{
        int chiffre;
        file *lien;
};
file * dernier=NULL;
file * premier=NULL;
```

e) les primitives *void enfiler(file*&,file*&,int)* et *int defiler(file*&, file*&)* :

```
void enfiler(file *& premier,file *& dernier, int x)
{
        file * f=new file;
        f->chiffre = x;
        f->lien=NULL;
        if (dernier) {
                dernier->lien=f;
                dernier=f;
        }
        else {
                dernier = f;
                premier = f;
        }
}

int defiler(file *&premier,file *&dernier)
{
        int t(-1);
        if (premier)
        {
                file * f=premier;
                t=f->chiffre;
                premier=premier->lien;
                if (premier==NULL) dernier=NULL;
                delete f;
        }
        return t;
}
```

f) La fonction *insertf* :

```
void insertf(file *&premier,file *&dernier, int nbre)
{
        int x=nbre, y;
        while (x>=10)
        {
                y=x%10;
                x=x/10;
                enfiler(premier, dernier, y);
        }
        enfiler(premier, dernier, x);
}
```

3. La fonction *bool verify(...)* :

```
bool verify(pile*&sommet, file *&premier, file *&dernier, int nbre)
{
        int res = nbchiffre(nbre)/2;
        int np, nf;
        for(int i=0; i<res; i++)
        {
                np = depiler(sommet);
                nf = defiler(premier, dernier);
                if (np!=nf) return false;
        }
        return true;
}
```

4. Le *main* :

```
#include<iostream>
using namespace std;
...
main()
{
        int z;
        cout << "Nombre = "; cin>> z;
        insertp(sommet, z);
        insertf(premier, dernier, z);
        if (verify(sommet, premier, dernier, z))
          cout << z << " est palyndrome" << endl;
        else
          cout << z << " n'est pas un nombre palindrome" << endl;
}
```

Exercice 14 : (piles et files)

1. Les structures *pile* et *file* :

```
struct pile
{
        int chiffre;
        pile * lien;
};
pile *sommet=NULL;
```

```
struct file
{
        int chiffre;
        file * lien;
};
file *premier=NULL;
file *dernier=NULL;
```

2. La foncion *CInsert* :

```
void CInsert(int N, pile *& sommet, file*& premier, file* & dernier)
{
        int nb=N, x;
        while(nb!=0)
        {
                x=nb%2;
                empiler(sommet,x);
                enfiler(premier,dernier,x);
                nb=nb/2;
        }
}
```

3. Les fonctions *defiler* et *depiler* :

```
int depiler(pile*&sommet)
{
        int v=-1;
        if (sommet!=NULL)
        {
                pile*temp=sommet;
                v=temp->chiffre;
                sommet=sommet->lien;
                delete temp;
        }
        return v;
}
```

```
int defiler(file*& premier, file*& dernier)
{
        int v=-1;
        if (premier!=NULL)
        {
                file*temp=premier;
                v=temp->chiffre;
                premier=premier->lien;
                delete temp;
                if (premier==NULL) dernier=NULL;
        }
        return v;
}
```

4. La fonction *BinDeci* :

```
int BinDeci(file*&premier, file*&dernier)
{
        int d=0, p=1, def;
        while(premier!=NULL)
        {
                def=defiler(premier, dernier);
                d=d+(def*p);
                p=p*2;
        }
        return d;
}
```

5. Le *main* :

```
main()
{
   int z;
   cout << "Nombre = "; cin>> z;
   CInsert(z,sommet,premier, dernier);
   cout << "le nombre = " << BinDeci(premier, dernier) << endl;
}
```

Exercice 15 : (arbres)

(1)–>(d)

(2)–>(a)

(3)–>(c)

(4)–>(b)

Exercice 16 : (arbres)

1. La structure de l'arbre :

```
struct noeud{
        int x;
        noeud * g;
        noeud * d;
};
noeud *r=NULL;
```

2. Les fonctions :
   — La fonction *create* :

```
noeud * create(int y)
{
        noeud *no=new noeud;
        no->g=NULL;
        no->d=NULL;
        no->x=y;
        return no;
};
```

   — La fonction *insert* :

```
void insert(noeud *& r,int y)
{
        if (r==NULL) r=create(y);
        else
        {
                if (r->x<y) insert(r->g,y);
                else insert(r->d,y);
        }
}
```

   — La fonction *hauteur* :

```
int hauteur(noeud *r)
{
        noeud *d=r;
        if (r==NULL) return 0;
        else return (1+max(hauteur(r->g), hauteur(r->d)));
}
```

3. Le *main* :

```
#include <math.h>
#include <iostream>
using namespace std;
struct noeud{
        int x;
        noeud * g;
        noeud * d;
};
noeud *r=NULL;
noeud * create(int);
void insert(noeud *&,int);
int hauteur(noeud *);
main()
{
        int a,nbval;
        cout << "Donnez le nombre de valeurs :"
        cin >> nbval;
        for(int i=0; i<nbval;i++)
        {
                cout << "noeud " << i << " = ";
                cin >> a;
                insert(r,a);
        }
        cout << "hauteur=" << hauteur(r)<< endl;
}
//les corps des fonctions
...
```

Exercice 17 : (arbres)

1. La structure générale :

```
struct Noeud{
        TypeElt elt;
        Noeud * gauche;
        Noeud * droit;
}
Noeud * T; // la racine de l'arbre
```

2. L'algorithme de parcours en profondeur préfixe :

```
void parcours_profondeur_prefixe(Noeud*T)
{
        if (! est_vide(T))
        {
                traiter(T); // Traiter le noeud T
                parcours_profondeur_prefixe(gauche(T));
                parcours_profondeur_prefixe(droit(T));
        }
}
```

3. Application de l'algorithme sur l'arbre :
   la fonction *parcours_profondeur_prefixe* traite les nœuds de l'arbre dans l'ordre : racine, gauche, droite. Ce qui donne :

   3-3-7-4-8-0-1-5-2-6-7-9

Exercice 18 : (arbres binaires)

1. la sturcture *tree* :

```
struct tree
{
   char val;
   tree* d;
   tree* g;
}
tree *racine1=NULL;
tree *racine2=NULL;
```

2. La fonction *Hauteur* :

```
int Hauteur(tree *r)
{
        tree *d=r;
        if (r==NULL) return 0;
        else return (1+max(Hauteur(r->g), Hauteur(r->d)));
}
```

3. La fonction *Idemtree* :

```
bool b=true;
bool IdemTree(tree*r1,tree*r2)
{
        if (r1!=NULL && r2!=NULL && b){
                if (r1->x != r2->x) return b=false;
                else
                {
                        b=IdemTree(r1->g, r2->g);
                        b=IdemTree(r1->d, r2->d);
                }
        }
        else
                if (r1!=NULL || r2!=NULL) return false;
                else return b;
}
```

Exercice 19 : (graphes)

1. Les trois représentations possibles : (voir le cours)
2. Les fonctions :
   La représentation avec la matrice d'incidence sommet-sommet :

```
bool **graphe=NULL;
bool *sommet;

bool ** alloc(int n)
{
  bool ** d=new bool*[n];
  for(int i=0;i<n;i++)
        d[i]=new bool[n];
  return d;
}
```

   a) La fonction *saisiegraph* :

```
void saisiegraph(bool ** g, int n)
{
  for(int i=0;i<n;i++)
    for(int j=0;j<n;j++)
    {
      cout << i << "-->" << j << " = ";
      cin >> g[i][j];
    }
}
```

   b) La fonction *ArcList* :

```
void arclist(bool ** g, int n)
{
    for(int i=0;i<n;i++)
      for(int j=0;j<n;j++)
         if (g[i][j]==1)  cout << i << "-->" << j << endl;
}
```

c) La fonction *ArcExterieur* :

```
void arcexterieur(bool **g, int n, int s)
{
   for(int j=0;j<n;j++)
        if (g[s][j]==1)  cout << s << "-->" << j << endl;
}
```

d) La fonction *Chemin* :

```
bool chemin(bool **g, int n, int s1, int s2)
{
  if (g[s1][s2]==1) return true;
  else
  { bool b=false;
    for(int i=0;i<n && !b;i++)
    {
      if (g[s1][i]==1 && sommet[i]==0)
      {
        sommet[i]=1; b=true;
        b=chemin(g,n,i,s2);

      }
    }
   return b;
  }
```

3. Le *main* et sa trace :

```
#include <iostream>
using namespace std;
...
main()
{
  int s1,s2,nn=6;
  graphe = alloc(nn);
  cout << "********** SAISIE ***********" << endl;
  saisiegraph(graphe,nn);
  cout << "***************AFFICHAGE**************"<< endl;
  arclist(graphe, nn);
  cout << "********** EXTERIEUR ***********" << endl;
  int ss;
  cout << "donnez un sommet : "; cin >> ss;
  arcexterieur(graphe, nn, ss);
  sommet = new bool[nn];
  for(int i=0;i<nn;i++) sommet[i]=0;
  cout << "*************CHEMIN**************" << endl;
  cout << "donner deux sommets ! "<< endl;
  cout << "s1="; cin>> s1;
  cout << "s2="; cin>> s2;
  if (chemin(graphe,nn,s1,s2))
        cout << "Il existe un chemin entre " << s1 << " et " <<
        s2 << endl;
  else cout << "Il n'existe pas un chemin entre " << s1 <<
        " et " << s2 << endl;
}
```

À faire la trace avec le chargé des travaux dirigés.

Exercice 20 : (fichiers)

1. ifstream $\longrightarrow$ (g) ouvrir le fichier en lecture
2. ofstream $\longrightarrow$ (b) ouvrir le fichier en écriture
3. eof $\longrightarrow$ (c) test de la fin du fichier
4. getline $\longrightarrow$ (f) lire une ligne complète
5. put $\longrightarrow$ (a) écrire un caractère
6. get $\longrightarrow$ (d) lire un caractère
7. close $\longrightarrow$ (e) fermer le fichier

Le programme :

```
# include <fstream.h>
main ( )
{
  string nom, tel;
  ofstream fichrep ("repertoire.txt");
  for(int i=0; i<10; i++)
  {
    cout << "\npersonne "<< i+1 << ":\n"
    cout << "nom? ";  cin >> nom;  fichrep <<nom << " ";
    cout << "\ntelephone?"; cin >> tel; fichrep << tel << "\n";
  }
  fichrep.close();
}
```

Exercice 21 : (fichiers)

Le programme complété :

```
#include<iostream>
using namespace std;
#include <cstring>
#include <stdio.h>
#include<fstream>
main(){
        ofstream flog("fphysique1.txt");
        flog << "ES2\n" << "Semestre2\n" << "Info2\n";
        flog.close();
        ifstream ilog("fphysique1.txt");
        ofstream olog("fphysique2.txt");
        string st;
        while(!ilog.eof()){
                getline(ilog,st);
                cout << st << endl; olog << st << endl;
        }
        ilog.close();
        olog.close();
}
```

Exercice 22 : (fichiers)

1. La saisie :

```
#include<fstream>
#include<iostream>
using namespace std;
main ( )
{
  string nom, symbole;
  // ouverture du fichier en écriture
  ofstream fich ("composant.txt");
  for(int i=0; i<10; i++)
  {
    cout << "composant "<< i+1 << ":\n";
    cout << "nom = ";  cin >> nom;  fich <<nom << " ";
    cout << "symbole = "; cin >> symbole; fich << symbole << "\n";
  }
  fich.close();
}
```

2. L'affichage :

```
#include <fstream>
#include<iostream>
using namespace std;
main()
{
  string nom, symbole;
  // ouverture du fichier en écriture
  ifstream log ("composant.txt");
  while(!log.eof())
  {
    log >> nom >> symbole;
    cout << nom << " : " <<  symbole << endl;
  }
  log.close();
}
```

6

EXERCICES SUPPLÉMENTAIRES

1

Exercice 1 : (programmation structurée)

1. Écrivez une fonction *polynome* renvoyant un polynôme (défini sous forme d'une structure). Cette fonction aura en paramètre un entier contenant le degré du polynôme et devra effectuer l'allocation dynamique du tableau contenant les coefficients.
2. Testez la fonction *polynome* dans un programme principal.
3. Écrivez une fonction *solution* prenant en paramètres un polynôme P et une valeur x et renvoyant la valeur du polynôme $P(x)$.
4. On souhaite disposer d'une fonction *polynome2* permettant de saisir au clavier les données d'un polynôme. La fonction à écrire doit renvoyer un polynôme et ne nécessite aucun paramètre. Elle est chargée de :
   — saisir au clavier le degré du polynôme,
   — créer le polynôme en utilisant la fonction *polynome*,
   — saisir les coefficients au clavier et les stocker dans un tableau.
5. testez les fonctions *polynome2* et *solution* dans un programme principal.

Exercice 2 : (fonctions)

Soit le code C++ ci-dessous :

1. Ces exercices sont conçus à être traités en travaux pratiques

```
#include <iostream>
using namespace std;
const int N=5;
typedef int Tv[N];
void fonction(Tv a)
{
        int nb,tmp;
        for(int i=0;i<N;i++)
        {
          cout<<"Veuillez taper l'entier numero "<<i<<" : ";
          cin>>a[i];
        }
        do{
          nb=0;
          for(int i=0;i<N-1;i++)
                if(a[i]>a[i+1]){
                        tmp=a[i];a[i]=a[i+1];a[i+1]=tmp;
                        nb++;
            }
        }while(nb!=0);
}
```

1. Testez la fonction *fonction* dans un programme principal.
2. Que fait cette fonction ?
3. Donnez la fonction *swap* qui fait la permutation de deux entiers, en utilisant le passage par référence.
4. Donnez la fonction *decroissant* qui a en paramètre le tableau de type *Tv* et qui renvoie en sortie le tableau trié dans l'ordre décroissant. Ceci en faisant appel aux fonctions *fonction* et *swap*.
5. Testez la fonction *decroissant* dans un programme principal.

Exercice 3 : (programmation structurée)

Soit le type, à définir, *tmat* d'une matrice carrée $n \times n$ d'entiers. Donnez les fonctions :

1. *alloc* qui reçoit en entrée la taille de la matrice et qui fait l'allocation dynamique de cette dernière,
2. *saisie* récursive qui fait la lecture des éléments d'une matrice de type *tmat*,
3. *prod* récursive qui reçoit deux matrices carrées de même taille et qui calcule leur produit,
4. *som* récursive qui reçoit deux matrices carrées de même taille et qui calcule leur somme,
5. *pal* récursive qui vérifie si un nombre donné en entrée est palindrome,
6. *nbPal* récursive qui reçoit en entrée la matrice d'entiers et retourne le nombre d'entiers palindromes,

7. *arithmetic* qui calcule le produit et la somme de deux matrices dont la taille est reçue en entrée. Cette fonction doit retourner les deux résultats en sorties.

Donnez le *main* qui permet de tester les fonctions ci-dessus. Sachant que ce programme affichera le menu suivant :

```
1 - ALLOCATION-SAISIE de la 1ère matrice carrée.
2 - ALLOCATION-SAISIE de la 2ème matrice carrée.
3 - CALCUL du PRODUIT de deux matrices.
4 - CALCUL de la SOMME de deux matrices.
5 - CALCUL de la SOMME et du PRODUIT de deux matrices.
6 - VÉRIFICATION d'un entier s'il est palindrome.
7 - CALCUL du nombre d'entiers palindromes.
8 - ARRÊT du programme.
```

Ces choix correspondent respectivement aux traitements :
— Allocation et saisie de la $1^{\text{ère}}$ matrice.
— Allocation et saisie de la $2^{\text{ème}}$ matrice.
— Calcul et affichage du produit des deux matrices.
— Calcul et affichage de la somme des deux matrices.
— Calcul et affichage de la somme et du produit des deux matrices.
— Saisie et vérification d'un entier s'il est palindrome.
— Calcul du nombre d'entiers palindromes contenus dans la $1^{\text{ère}}$ matrice.
— Sortir du programme.

NB :
- les choix 3, 4 et 5 restent inactifs tant que 1 et 2 n'ont pas été exécutés,
- le choix 7 reste inactif tant que 1 n'a pas été exécuté.

Exercice 4 : (tableau à deux dimensions)

Donnez en C++ :

1. une fonction *triangle* qui construit le triangle de *PASCAL* de degré N et le mémorise dans une matrice carrée de dimension $(N+1) \times (N+1)$.

   Exemple 6.0.1 (triangle de Pascal de degré 6) *Le triangle de Pascal de degré 6 se donne comme suit :*

```
1
1  1
1  2   1
1  3   3   1
1  4   6   4   1
1  5  10 10  5  1
1  6  15 20 15 6  1
```

2. la fonction *int SommePuiss(int a, int b, int n, int ** mat)* qui permet de calculer $(a+b)^n$, avec *mat* la matrice $(n+1) \times (n+1)$ qui contient les valeurs du triangle de *PASCAL* de degré n.

Exemple 6.0.2 ($(a+b)^n$)

$(a+b)^2 = 1a^2 + 2ab + 1b^2$

$(a+b)^3 = 1a^3 + 3a^2b + 3ab^2 + 1b^3$

$(a+b)^4 = 1a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + 1b^4$

3. testez les deux fonctions dans le *main*.

NB : la matrice est un tableau dynamique à deux dimensions.

Exercice 5 : (tests sur la récursivité)

1. Compléter le programme ci-dessous :

```
#include <iostream>
using namespace std;
int fact(int n)
{
        int p(1);
        for(int i=1; i<=n; i++)
                p=p*i;
        return p;
}
int comb_iter1(int n, int p)
{       ...  }
int comb_rec(int n, int p)
{       ...  }
main()
{       ...  }
```

Sachant que :

— *comb_iter* calcule le coefficient binomial défini par :

$$C_n^k = C_n^k = \frac{n!}{k!(n-k)!}, \quad 0 \leq k \leq n$$

On utilise la version itérative *fact* qui calcule le factoriel.

— *comb_rec* est une fonction récursive correspondant à la version itérative *comb_iter*.

— le programme principal doit tester la fonction *comb_iter1* suivie de *comb_rec* avec n et k variables.

2. Testez les deux fonctions pour $(n,k) = (12,5)$, puis pour $(n,k) = (16,13)$. Que peut-on remarquer ? Pourquoi ?

3. Ajoutez à votre code source les fonctions *min* et *comb_iter2*, suivantes :

```
int min(int a, int b)
{
  return (a<b)? a: b;
}
int comb_iter2(int n, int p)
{
    int **C=new int*[n+1];
    for (int i=0; i<=n;i++)
       C[i] = new int[p+1];
    for (int i = 0; i <= n; i++)
       for (int j = 0; j <= min(i, p); j++){
                 if (j == 0 || j == i) C[i][j] = 1;
                 else C[i][j] = C[i-1][j-1] + C[i-1][j];
       }
    return C[n][p];
}
```

La fonction *comb_iter2* est une autre version itérative de *comb_rec*. Ajoutez les instructions nécessaires pour calculer les temps d'exécution correspondant à *comb_iter2* et *comb_rec*, pour $(n,k) = (16,13)$, puis pour $(n,k) = (40,20)$. Que peut-on remarquer ? Pourquoi ?

Exercice 6 : (fichiers)

1. Écrire dans le fichier *im.ppm* les lignes suivantes :

```
P3
# Image I
3 3 255
0 0 0
255 255 255
100 100 100
255 0 0
0 255 0
0 0 255
255 255 0
255 0 255
0 255 255
```

2. Ouvrez le fichier avec un visionneur d'images.
3. Commentez le résultat de chacune des actions suivantes :
   a) inverser entre la dernière et l'avant dernière ligne,
   b) remplacer $P3$ par $P2$,
   c) etc.

7

PROBLÈMES

7.1 ÉNONCÉS

7.1.1 *Problème 1 : (tableaux et programmation structurée)*

1. Donnez la fonction itérative *int vonvert(int, int)* qui convertit un nombre décimal N dans une base b inférieure à 10.
2. Donnez la fonction récursive *bool exist(int nb, int c)* qui vérifie si le chiffre c apparaît au moins une fois dans le nombre *nb*.
3. Soient un tableau T de taille n et un entier b. En utilisant les fonction *convert* et *exist*, donnez la fonction *void f(int *T, int n, int b, ...)* qui fournit :
   — un tableau dynamique C d'entiers, dont chaque valeur $C[i]$ correspond à la valeur décimale $T[i]$ convertie dans la base b. L'allocation de la mémoire nécessaire pour C, doit se faire à l'intérieur de la fonction.
   — le nombre de valeurs dans C, ayant au moins un chiffre égal à $b-1$.
4. Donnez le *main* qui permet de tester la fonction *f*.

7.1.2 *Problème 2 : (récursivité, structures complexes et programmation structurée)*

Partie 1 :

La Multiplication Russe de deux nombres X et Y est basée sur une succession de divisions de X sur 2 et de multiplications de Y fois 2. La valeur du produit $X \times Y$ est donné par la somme des multiples correspondant aux divisions impaires.

Exemple 7.1.1 (multiplication russe) *En suivant le principe de la multiplication russe et le tableau ci-dessous, le produit* 13×15 *est égal à* $195 = 15 + 60 + 120$.

13/2=6	*6/2=3*	*3/2=1*	*1/0=0*	*Les divisions avec des restes non nuls : 6, 1 et 0*
15	$15 \times 2 = 30$	$30 \times 2 =$***60***	$60 \times 2 =$***120***	*Les multiplicandes à sommer : 15, 60 et 120*

Donnez en C++, la fonction récursive *int RussProd(int, int)* qui permet de calculer le produit de deux entiers naturels en utilisant le principe de la multiplication russe décrit ci-dessus.

Partie 2 :

Soient les structures suivantes :

```
struct pile{
  int val;
  pile *lien;
};
pile *sommet=NULL;
```

et

```
struct file{
  int val;
  file *lien;
};
file * dernier=NULL;
file * premier=NULL;
```

En supposant que les primitives *void empiler(...)*, *int depiler(...)*, *void enfiler(...)* et *int defiler(...)* soient prédéfinies, donnez en C++ les fonctions :

1. *void pfinsert(..., int)* qui permet d'empiler les quotients et d'enfiler les restes de la succession de divisions par deux réalisées sur un entier naturel donné,
2. *int fprod(...)* qui permet de calculer le produit de deux nombre naturels, en utilisant le principe de la multiplication russe citée ci-dessus et la structure file. On suppose que la file contient les restes de la succession des divisions du premier nombre par 2,
3. *int maxpair(...)* qui permet de calculer le plus grand nombre pair inférieur à un nombre naturel donné en entrée, en utilisant la pile qui contient les quotients de la succession des divisions réalisées sur ce nombre.

Testez toutes les fonctions dans un main.

7.1.3 *Problème 3 : (récursivité, structures complexes et programmation structurée)*

La décomposition en produit de facteurs premiers, aussi connue comme la factorisation entière en nombres premiers, consiste à chercher à écrire un entier supérieur ou égal à 2 sous forme d'un produit de nombres premiers. Par exemple, si le nombre donné est 45, la factorisation en nombres premiers est : $3^2 \times 5$, soit $3 \times 3 \times 5$.

Partie 1 :

Un nombre premier est un entier naturel qui admet exactement deux diviseurs distincts entiers et positifs, le nombre lui même et 1. Donnez la fonction récursive *bool premier(int)* qui permet de vérifier si un nombre est premier.

Partie 2 :

On suppose que les primitives *void empiler(...)*, *int depiler(...)*, *void enfiler(...)* et *int defiler(...)* sont prédéfinies.

1. en utilisant la fonction *premier*, donnez la fonction *void fprime(...)* qui permet de calculer et d'enfiler tous les nombres premiers inférieur ou égal à un nombre donné,
2. donnez la fonction *void pprod(...)* qui permet de décomposer un nombre naturel n en produit de facteurs premiers. Elle doit réaliser deux empilements pour chaque facteur premier ie. elle doit empiler le facteur premier puis la puissance de ce facteur. On suppose que la file contient au préalable les valeurs résultat de la fonction *fprime* appliquée sur n,
3. en utilisant le contenu de la pile, donnez la fonction *int compute(...)* qui permet de calculer le nombre naturel correspondant.

7.1.4 *Problème 4 : (programmation structurée et complexité algorithmique)*

Soit un tableau de flottants T de taille N, donnez :

1. la fonction $void \quad move(float*, int, int)$, qui déplace un flottant de T de la position e_1 vers la position e_2, ceci en appliquant une suite de décalages vers la droite (voir l'exemple ci-dessous).

   Exemple 7.1.2 (application de la fonction *move*) *Considérons le tableau T suivant :*

0.5	-1.1	0	6.3	-5.8	10

   Pour déplacer la valeur -5.8 *de sa position* $e_1 = 4$ *vers la position* $e_2 = 1$, *le tableau T résultat, après application de la fonction move, se donne comme suit :*

0.5	-5.8	-1.1	0	6.3	10

2. la fonction $void \quad sort(float*, int)$ qui fait appel à la fonction *move*, pour trier le tableau T dans l'ordre croissant.
3. le *main* qui permet de :
   a) définir le tableau T, avec N variable,
   b) saisir les valeurs du tableau T,
   c) tester la fonction *sort*.

   On suppose que T est un tableau dynamique.
4. le calcul de la complexité asymptotique $C_{sort}(N)$ correspondant à la fonction *sort*. On suppose que $C_{move}(N) = N$ (la complexité asymptotique de la fonction *move*).

7.1.5 *Problème 5 : (structures complexes)*

Soient les structures suivantes :

```
struct pile{
  unsigned  int val;
  pile *lien;
};
pile *sommet=NULL;
```

et

```
struct file{
  unsigned  int val;
  file *lien;
};
file * dernier=NULL;
file * premier=NULL;
```

Partie 1 :

En supposant que les primitives *void empiler(...)*, *int depiler(...)*, *void enfiler(...)* et *int defiler(...)* soient prédéfinies :

1. Donnez la fonction *insertpf* qui permet d'insérer les chiffres de droite à gauche dans la pile et dans la file.
2. Un nombre palindrome est un nombre symétrique écrit dans une certaine base *a* comme ceci : $a_1a_2a_3... \mid ...a_3a_2a_1$ (exemple : 15851, 1661, etc.).

   Partie 2 :

   En utilisant les structures *pile* et *file* précédentes, donnez la fonction *bool verify(...)* qui vérifie si un nombre est palindrome. On suppose que les chiffres du nombre sont déjà chargés de droite à gauche dans la pile et dans la file.
3. Un arbre binaire de recherche est tel que tout nœud a une valeur supérieure à celles des nœuds de son sous arbre gauche et inférieure à celles des nœuds de son sous arbre droit. Donnez en C++ :
   a) la structure de l'arbre, sachant que chaque nœud porte une valeur de type entier naturel,
   b) la fonction *insert* récursive qui insère un élément dans l'arbre en respectant le principe de l'arbre de recherche. On suppose que les éléments à insérer sont déjà chargés dans une pile de type *pile* défini ci-dessus.

7.2 SOLUTIONS

Problème 1 : (tableaux et programmation structurée)

1. La fonction itérative *int vonvert(int, int)* :

```
#include <math.h>
int convert(int n, int b)
{
        int x=0, p(0);
        do
        {
                x+=(n%b)*((int)pow(10,p));
                n=n/b;
                p++;
        }while(n>=b);
        return x+n*((int) pow(10,p));
}
```

2. La fonction récursive *bool exist(int nb, int c)* :

```
#include <math.h>
bool exist(int nb, int c)
{
        if (nb>10)
                if (nb%10 == c) return true;
                else return exist(nb/10,c);
        else
                if (nb==c) return true;
                else return false;
}
```

3. La fonction *void f(int *T, int n, int b, ...)* :

```
void f(int *T, int n, int b, int *& V, int & nb)
{
        V=new int(n);
        nb=0;
        for (int i=0; i<n; i++)
        {
           V[i]=convert(T[i],b);
           if (exist(V[i], b-1)) nb++;
        }
}
```

4. Le *main* :

```
#include<iostream>
using namespace std;
main()
{
        int tab[5]={10,55,12,17,111};
        int s(5), b(4);
        int r, *res;
        f(tab,s,b,res,r);
        for (int i=0; i<s; i++)
           cout << "res[" << i << "]=" << res[i] << endl;
        cout << "r=" << r << endl;
}
```

7.2.1 *Problème 2 : (récursivité, structures complexes et programmation structurée)*

Partie 1 :

La fonction récursive *int RussProd(int, int)* :

```
int RussProd(int x, int y)
{
        int p(0), m(y);
        while(x>0)
        {
                if (x%2!=0) p+=m;
                x=x/2;
                m=m*2;
        }
        return p;
}
```

Partie 2 :

1. La fonction *void pfnsert(..., int)* :

```
void  pfinsert(pile *&sommet, file *& premier,
                            file *& dernier, int n)
{
    int d,r ;
    while(n !=0)
    {
         r=n%2; enfiler(premier, dernier,r);
         n=n/2; empiler(sommet,n);
    }
}
```

2. La fonction *void fprod(...)* :

```
int fprod(file *&premier, file *& dernier, int n)
{
    int d, m=n, p=0;
    while(premier!=NULL)
    {
                d=defiler(premier,dernier);
                if (d!=0) p+=m;
                m=m*2;
    }
    return p;
}
```

3. La fonction *int maxpair(...)* :

```
int maxpair(pile *&sommet)
{
    int d;
    while(sommet!=NULL)
        d=depiler(sommet);
    return(d * 2) ;
}
```

7.2.2 *Problème 3 : (récursivité, structures complexes et programmation structurée)*

Partie 1 :

La fonction récursive *premier* :

```
bool nbpremier(int n)
{
        for(int i=n-1; i>=2; i--)
                if (n%i==0) return 0;
        return 1;
}
```

Partie 2 :

1. La fonction *void fprime* :

```
void fprime(file *&premier, file *& dernier, int n)
{
        for(int i=2; i<=n; i++)
                if (premier(i))  enfiler(premier,dernier,i);
}
```

2. La fonction *void pprod* :

```
void pprod(pile *&sommet, file *&premier, file *& dernier, int n)
{
        int x,i; bool b;
        while(premier!=NULL)
        {
                x=defiler(premier, dernier);
                i=0; b=true;
                while(b)
                {
                        if (n%x!=0) b=false;
                        else{ i++; n=n/x;}
                }
                if (i!=0)
                {
                        empiler(sommet, x);
                        empiler(sommet, i);
                }
        }
}
```

3. La fonction *int compute(...)* :

```
int compute(pile *&sommet)
{
        int x=1, p,k;
        while(sommet!=NULL)
        {
                k=depiler(sommet);
                p=depiler(sommet);
                x*=puiss(p,k);
        }
        return x;
}
```

7.2.3 *Problème 4 : (programmation structurée et complexité algorithmique)*

1. La fonction *move* :

```
void move(float *tab, int e1, int e2)
{
        float ee=tab[e2];
        for(int i(e2); i>=e1; i--)
                tab[i]=tab[i-1];
        tab[e1]=ee;
}
```

2. La fonction *sort* :

```
void sort(float *tab, int N)
{
  for(int i(0); i<N; i++)                    -->N itérations
  {
        for(int j(i+1); j<N; j++)            -->N-(i+1) itérations
      if (tab[i]>tab[j]) move(tab,N,i,j);-->N ops fondamentales
  }
}
```

3. Le *main* :

```
#include <iostream>
using namespace std;

main()
{
        int N;
        cout << "La taille du tableau T = ";
        cin >> N;
        float *T=new float[N];
        for(int i(0); i<N; i++)
        {
                cout << "T[" << i << "]=";
                cin >> T[i];
        }
        sort(T,N);
        for(int i(0); i<N; i++)
                cout << "T[" << i << "]=" << T[i] << endl;
}
```

4. Le calcul de la complexité :

$$C_{sort}(N) = \sum_{i=0}^{N-1}\sum_{j=i+1}^{N-1}(N) = N\sum_{i=1}^{N}\sum_{j=i+1}^{N-1}1 = N\sum_{i=1}^{N}(N-(i+1))$$

$$= N(N\sum_{i=0}^{N-1}1 - \sum_{i=0}^{N-1}(i+1)) = N(N.N - \sum_{i=1}^{N}i) = N^3 - \frac{N(N+1)}{2}$$

$$= N^3 - \frac{1}{2}N^2 - \frac{1}{2}N \in O(N^3)$$

7.2.4 *Problème 5 : (structures complexes)*

Partie 1 :

1. La fonction *insertpf*

```
void insertpf(unsigned int n, pile *&sommet, file *& premier,
                            file *& dernier)
{
        int x(n), r;
        do
        {
                r=x%10;
                empiler(sommet, r);
                enfiler(premier,dernier, r);
                x=x/10;
        }while(x!=0);
}
```

2. La fonction *verify* :

```
bool verify(pile *&sommet, file *& premier,file *& dernier)
{
        while(sommet)
                if (depiler(sommet) != defiler(premier,dernier))
                        return false;
        if (premier) return true;
        else return true;
}
```

Partie 2 :

a) La structure de l'arbre :

```
struct arbre{
        unsigned int x;
        arbre * gauche;
        arbre * droit;
};
arbre *racine=NULL;
```

b) La fonction *insert* :

```
void insert(noeud *& racine, pile *& sommet)
{
        int y=depiler(sommet);
        if(y>=0)
        {
          if (racine==NULL) {
            arbre * N= new arbre;
            N->x=y; N->gauche=NULL; N->droit=NULL;r=N;
          }
          else {
            empiler(sommet, y);
            if (racine->x<y) insert(racine->gauche,sommet);
            else insert(racine->droit,sommet);
          }
        } else return;
}
```

BIBLIOGRAPHIE

[Baynat et al., 2007] Baynat, B., Chrétienne, P., Hanen, C., Kedad-Sidhoum, S., Munier-Kordon, A. et Picouleau, C. (2007). Exercices et problèmes d'algorithmique. Dunod.

[Bensaoud-Senhadji, 2005] Bensaoud-Senhadji, T. (2005). Concepts et outils de base de la programmation. Science et Technologie. ANEP.

[Delannoy, 1995] Delannoy, C. (1995). Programmer en Langage C++. Eyrolles, 2nd édition.

[Delannoy, 1999] Delannoy, C. (1999). Exercices en langage C++. Eyrolles, 2nd édition.

[Sedgewick, 1991] Sedgewick, R. (1991). Algorithmes en langage C. I.I.A. Informatique intelligence artificielle. Dunod.