22/11/2014

FICHE TECHNIQUE

INFORMATIQUE 3

« Comme la Hongrie, le monde informatique a une langue qui lui est propre. Mais il y a une différence. Si vous restez assez longtemps avec des Hongrois, vous finirez bien par comprendre de quoi ils parlent. » Dave Barry

Cette fiche a pour objectif de recenser les points nécessaires pour aborder le contrôle continu d'algorithmique et programmation en C. Elle vous présente également une méthodologie de travail qui vous permettra à coup sûr d'améliorer votre compréhension de l'informatique. Enfin au travers de ce document, nous vous proposons un ensemble d'exercices et corrigés susceptibles de vous guider vers l'expression de votre savoir.

ATIBITA Jonas Adrien, 4ème année génie informatique

Objectifs

- Maitriser les éléments de base d'algorithmique et de la programmation dans le langage C
- Savoir écrire un programme en pseudocode et son équivalent dans le langage C
- Maitriser la programmation modulaire (au moyen de sous-programmes) en C
- Maitriser le passage d'arguments aux sous-programmes (par valeur et par adresse)
- Savoir manipuler les pointeurs
- Comprendre la récursivité et pouvoir réaliser un programme récursif
- Savoir manipuler les structures de données aussi bien en algorithmique que dans le langage C

Rappels sur les notions importantes

La structure d'un algorithme :

```
Algorithme : Nom_algorithme
/**Objectifs**/
/**Données**/
/**Résultats**/
/**Partie déclarative**/
Constante :
Type :
Variable :
/**Les fonctions et les procédures**/
…….
/**Partie exécution**/
Debut
Instructions ;
Fin
```

La structure générale d'un programme C

```
/**Description du programme**/

/**Les directives préprocesseur (chacune sur une seule ligne)

Ex: #include <stdio.h>, #include <stdlib.h>, #include <string.h>, #include
<math.h>,#define MAX 20*/

/*La déclaration des variables externes et globales*/

/*La définition des fonctions secondaires. Elles peuvent être placée avant
ou après la fonction main() * /

/*La fonction main*/

Int main(){

      /**Déclaration des variables internes

      Ex : int tab[20] ; */

      Instructions ;
```

Les Types prédéfinis en C

- **Char** pour les **caractères** (ne peut en contenir qu'un seul à la fois)
- **Float**, **double**, **long double** pours les nombres **réels** ou **flottants**
- **Short, Int, long, long long** pour les **nombres entiers relatifs signés ou non signés**
- **Unsigned** permet de signifié qu'un type ne peut contenir que des nombres positifs

NB : En C il existe un mécanisme de conversion entre les caractères et les nombres. Il n'est donc pa étonnant qu'une instruction comme **A+2 ;** soit validée par le compilateur.

Les opérateurs

1. L'affectation simple

En algorithmique on utilise une flèche ←. En C, on utilise =.

Les opérateurs arithmétiques :

Opérateurs	Notation en pseudo code	Notation en C
Addition	+	+
Soustraction	-	-
Division	/	/
Modulo (reste de la division entière)	%	%

2. Les opérateurs relationnels

Opérateurs	Notation en pseudo code	Notation en C
Strictement supérieur	>	>
Supérieur ou égal	>=	>=
Strictement inférieur	<	<
Inférieur ou égal	<=	<=
égal	=	==
Différent	≠	!=

3. Les opérateurs logiques booléens

Opérateurs	Notation en pseudo code	Notation en C
ET logique	ET	&&
OU logique	OU	||
Non logique	NON	!

4. Les opérateurs logiques bit à bit

Les six opérateurs suivants sont uniquement disponibles en C. Ils permettent de manipuler des entiers au niveau du bit. Ils s'appliquent aux entiers de toute longueur (short, int ou long), signés ou non.

Opérateurs	Notation en C
ET	&
OU	|
OU exclusif	~
OU exclusif	^
Décalge à gauche	<<
Décalage à droite	>>

5. Les opérateurs d'affectation composée

Les opérateurs d'affectation composée sont : += ; -= ; *= ; /= ; %= ; &= ; ^= ; |= ; <<= ; >>=

Pour tout opérateur d'affectation composée op, l'expression : ***expression_1 op= expression_2*** est équivalente à : ***expression_1 = expression_1 op expression_2***

Les opérateurs d'incrémentation et de décrémentation

Les opérateurs d'incrémentation ++ et de décrémentation -- s'utilisent aussi bien en **suffixe (i++)** qu'en **préfixe (++i).** Dans les deux cas la variable i sera incrémentée, toutefois dans la notation suffixe la valeur retournée sera l'ancienne valeur de i alors que dans la notation préfixe se sera la nouvelle. Par exemple,

```
int a = 3, b, c;
b = ++a; /* a et b valent 4 */
c = b++; /* c vaut 4 et b vaut 5 */
```

6. L'opérateur virgule

Une expression peut être constituée d'une suite d'expressions séparées par des virgules:

Expression_1, expression_2, … , expression_n

Cette expression est alors évaluée de gauche à droite. Sa valeur sera la valeur de l'expression de droite. Par exemple,

```
main (){
      int a, b;
      b = ((a = 3), (a + 2));
       printf ("\n b = %d \n",b);
// imprime b = 5.
}
```

La virgule séparant les arguments d'une fonction ou les déclarations de variables n'est pas considérée comme l'opérateur virgule. En particulier l'évaluation dans l'ordre de gauche à droite n'est pas garantie. Par exemple l'instruction composée :

```
main (){
       int a=1;
       printf("\%d \%d",++a,a);
}
```

(compilée avec gcc) produira la sortie **2 1** sur un PC Intel/Linux et la sortie **2 2** sur un DEC Alpha/OSF1.

7. L'opérateur conditionnel ternaire

L'opérateur conditionnel **?** est un opérateur ternaire. Sa syntaxe est la suivante : ***condition ? expression_1 : expression_2***

Cette expression est égale à **expression_1** si condition est satisfaite, et à **expression_2** sinon. Par exemple, l'expression ***x >= 0 ? x : -x*** correspond **à la valeur absolue d'un nombre**. De même l'instruction ***m = ((a > b) ? a : b);*** affecte à **m le maximum de a et de b**.

8. L'opérateur de conversion de type

L'opérateur de conversion de type, appelé **cast**, permet de modifier explicitement le type d'un objet. On écrit ***(type) objet***

Par exemple,

```
main(){
     int i = 3, j = 2;
     printf("%f \n",(float)i/j); // retourne la valeur 1.5 au lieu de 1.
}
```

9. L'opérateur adresse

L'opérateur d'adresse **&** applique à une variable retourne l'adresse-mémoire de cette variable.

La syntaxe est **&objet**

Règles de priorité des opérateurs

Le tableau suivant classe les opérateurs par ordres de priorité décroissants. Les opérateurs placés sur une même ligne ont même priorité. Si dans une expression figurent plusieurs opérateurs de même priorité, l'ordre d'évaluation est défini par la flèche de la seconde colonne du tableau. On préfèrera toutefois mettre des parenthèses en cas de doute...

Opérateurs	Diffusion de la priorité
() ; [] ; ->	→
! ; ~ ; ++ ; -- ; -(unaire) ; (type) ; *(indirection) ; &(adresse) ; sizeof	←
* ; / ; % ;	→
+ ; -(binaire)	→
<< ; >>	→
< ; <= ; > ; >=	→
== ; !=	→
&(et bit-à-bit)	→
^	→
|	→
&&	→
||	→
? :	←
= ; += ; -= ; *= ; /= ; %= ; &= ; ^= ; |= ; <<= ; >>=	←
,	→

Les structures de contrôle

1. Sélection simple :

En algorithmique	Dans le langage de programmation C

Si Expression alors
  Instructions ;
Finsi
	If (expression) {
	  Instructions ;
	}

Si expression alors
  Instruction(s)_1 ;
Sinon
  Instruction(s)_2 ;
finSi
	If (expression){
	  Instruction(s)_1 ;
	Else{
	  Instruction(s)_2 ;
	}

2. Sélection multiple

En algorithmique	Dans le langage de programmation C

Cas où (variable_entiere)
  Constante1 : instruction(s)_1 ;
  Constante2 : instruction(s)_2 ;
  ….
  ConstanteN : instruction(s)_N ;
Sinon
Instruction(s)
FinCas
	Switch(variable){
	  Case Constante1 : instruction(s)_1 ;break;
	  Case Constante2 : instruction(s)_2 ;break;
	  ….
	  Case ConstanteN : instruction(s)_N;break;
	Default :
	  Instruction(s)
	}

NB : Si une contantei (i=1..N) à la même valeur que la variable ou variable_entiere, la série d'instruction correspondante est exécutée jusqu'à la rencontre d'un break. Sinon **instruction(s)** est exécutée.

Structure iterative:

En algorithmique	Dans le langage de programmation C

Pour compteur← initial (pas) final faire
Instruction ;
FinPour
	For(compteur=initial ;compteur(op)final ;compteur=compteur+pas){
	Instruction ;
	}

TantQue expression faire{
Instruction(s) ;
FinTantQue
	While (expression){
	Instruction(s) ;
	}

Repeter Instruction(s) ; Jusqu'à (expression)	Do{ Instruction(s) ; }while !(expression) ;

NB :

- Il ne faut surtout pas modifier le compteur à l'intérieur de la boucle **Pour**
- **(op)** étant un opérateur de comparaison.
- La boucle **Pour** est utilisée lorsque le nombre d'itération est connu avant d'entrer dans la boucle
- Les boucles **TantQue** et **Répeter** sont utilisées pour un nombre d'itérations inconnu. Avec ceci de particulier que **Répéter** traduit le fait que la boucle s'exécute au moins une fois.

Les instructions de branchement inconditionnel

1. L'instruction break

L'instruction break peut, plus généralement, être employée `à l'intérieur de n'importe quelle boucle. Elle permet d'interrompre le déroulement de la boucle, et passe à la première instruction qui suit la boucle. En cas de boucles imbriquées, **break** fait sortir de la boucle la plus interne. Par exemple, le programme suivant :

```
main(){
   int i;
   for (i = 0; i < 5; i++){
        printf("i = %d\n",i);
        if (i == 3)
             break;
        }
        Printf ("valeur de i a la sortie de la
boucle = %d\n",i);
```

imprime à l'écran

```
i = 0
i = 1
i = 2
i = 3
Valeur de i à la sortie de la boucle = 3
```

2. Branchement non conditionnel : continue

L'instruction continue permet de passer directement au tour de boucle suivant, sans exécuter les autres instructions de la boucle. Ainsi le programme

```
main(){
    int i;
    for (i = 0; i < 5; i++){
        if (i == 3)
            continue;
        printf ("i = %d\n",i);
    }
     printf("valeur de i a la sortie de la boucle
= %d\n",i);
}
```

Imprime

```
i = 0
i = 1
i = 2
i = 4
Valeur de i à la sortie de la boucle = 5
```

3. Branchement non conditionnel goto

L'instruction **goto** permet d'effectuer un saut jusqu'à l'instruction étiquette correspondant. Elle est à proscrire de tout programme C digne de ce nom.

Les fonctions d'entrées-sorties classiques

1. En pseudocode

Les instructions **lire ()** et **écrire ()** permettent respectivement d'entrer les données aux claviers et afficher les informations à l'écran.

Avant de lire chaque variable, il faut prendre l'habitude d'écrire les libellés à l'écran pour prévenir l'utilisateur de ce qu'il doit entrer.

2. Dans le langage de programmation C

les fonctions **getchar()** et **scanf()** permettent de faire des entrées à partir du clavier tandis que les fonctions **putchar()** et **printf()** réalisent les affichages à l'écran.

Les pointeurs

1. Déclaration :

En pseudocode	Dans le langage C
¬typeVariable nomPointeur	typeVariablePointée *nomPointeur

2. Manipulation des pointeurs

Opérateur	En pseudo code	En langage C	But
Adresse de	@	&	Obtenir l'adresse d'une variable
Contenu de	©	*	Accéder au contenu d'un pointeur

Récursivité

Un sous-programme récursif est une procédure ou une fonction qui fait appel à elle-même au cours de son déroulement.

Structure d'un sous-programme récursif :

```
Procédure recursive(paramètres)
/*Objectif:….*/
/*Paramètres d'entrée:…*/
/*Paramètre de sortie :….*/
Début
     Si condition_d'arret alors
       Instruction du point d'arret ;
      Sinon
       Début
            …..
            Appeler recursive(paramètres changés) ; //appel
récursif
            ….
       Fin
      Finsi
Fin
```

Les structures de données

1. Les tableaux

Déclaration en algorithmique	Déclaration en C
Type Nom_du_type_tableau=Tableau[nombre_d'éléments] de type_des_éléments Variable Nom_du_type_tableau :nom_tableau ;	Type_des_éléments nom_du_tableau[nombre_eléments];

2. Les tableaux multidimensionnels

Ce sont des tableaux dont le type des éléments est un autre tableau.

Déclaration en algorithmique	Déclaration en C
Type Nom_du_type_tableau=Tableau[nombre_d'éléments1] [nombre_d'éléments2]… [nombre_d'élémentsN] de type_des_éléments Variable Nom_du_type_tableau :nom_tableau ;	Type_des_éléments nom_du_tableau[nombre_eléments1] [nombre_d'éléments2]… [nombre_d'élémentsN];

Les pointeurs

Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiés de manière univoque par un numéro qu'on appelle **adresse**. Pour retrouver une variable, il suffit donc de connaître l'adresse de l'octet où elle est stockée (ou, s'il s'agit d'une variable qui recouvre plusieurs octets contigus (ex : tableau), l'adresse du premier de ces octets). Pour des raisons évidentes de lisibilité, on désigne souvent les variables par des identificateurs, et non par leur adresse.

C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire. Toutefois, il est parfois très pratique de manipuler directement une variable par son adresse.

Déclaration en algorithmique	Déclaration en C
¬ Type_pointeur :nom_variable;	type *nom_du_pointeur;

Tableaux et pointeurs

Une variable de type tableau correspond à un **pointeur constant** qui est dirigé vers le premier élément du tableau

Gestion des tableaux de taille variable

Pour gérer les tableaux de taille variable, on fait appel aux pointeurs et on fait des allocations dynamiques de mémoire.

Tableaux multidimensionnel

Manipulation en algorithmique	Manipulation en C
Type Nom_du_type_tableau=Tableau[nombre_d'éléments1] [nombre_d'éléments2]… [nombre_d'élémentsN] de type_des_éléments Variable Nom_du_type_tableau :nom_tableau ;	Type_des_éléments nom_du_tableau[nombre_eléments1] [nombre_d'éléments2]… [nombre_d'élémentsN];

Le passage d'arguments

Il existe deux modes de transmissions des arguments dans un programme. **Le passage des arguments par valeur** et **le passage des arguments par adresse (ou référence)**. Dans le passage des arguments par valeur, les arguments transmis au sous-programme sont des copies des variables stockées en mémoire. Dans le passage des arguments par adresse(ou référence), ce sont des adresses mémoire des variables qui sont transmises.

Quelques techniques usuelles pour résoudre les problèmes spécifiques

- Bien comprendre les problèmes posés
- Modéliser le problème dans le langage informatique. Vous devez ici cerner les outils informatiques qui vous permettront de résoudre le problème posé et établir la liste des types de données à manipuler.
- Sélectionner si nécessaire le type de programme à écrire (fonction ou procédure) et envisager pour chacune d'elle le mode de transmission des paramètres (par valeur ou par référence).
- Être intuitif. L'intuition étant une qualité qui s'acquiert en traitant des exercices et des problèmes.

Astuces et méthodologie de rédaction

- Toujours travailler à partie d'un brouillon. Ceci afin d'avoir une copie d'examen propre et sans ratures.
- Veuillez à bien indenter ses codes
- Toujours vérifier ses algorithmes en s'assurant qu'ils réalisent bien ce qui est demandé
- Effectuer un travail efficace en évitant de sprinter sur les exercices. L'objectif étant de « couvrir une part significative de l'épreuve de manière importante » (comme aime à le dire le Dr NDONG NGUEMA).

Quelques exercices (avec éventuellement corrigés)

« Un problème créé ne peut être résolu en réfléchissant de la même manière qu'il a été créé. » A. Einstein

TP du chapitre 1/ Exercice 7: Vérification de l'ordre alphabétique

Programme C :

En C, il existe une correspondance entre les caractères et les entiers.

```
/**Réalise la vérification de la classification par ordre alphabétique
de 3 caractères*/
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char caractere1,caractere2,caractere3,fin_de_ligne;
    printf("EXERCICE7:Vérification de l'ordre alphabétique");
    printf("\nEntrer le nombre 1: ");
    caractere1=getchar();
    scanf("%c",&fin_de_ligne);
    printf("\nEntrer le nombre 2: ");
    caractere2=getchar();
    scanf("%c",&fin_de_ligne);
    printf("\nEntrer le nombre 3: ");
    caractere3=getchar();
    scanf("%c",&fin_de_ligne);
    if(caractere1<=caractere2){
        if(caractere2<=caractere3){
            printf("\nDans l'ordre aphabétique");
        }else{
            printf("\nPas dans l'ordre alphabétique");
        }
    }else{
        printf("\nPas dans l'ordre alphabétique");
    }
}
```

TP du chapitre 1/ Exercice 8:

Programme C

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int nombre;
    printf("\nEXERCICE8:Unités Décimales");
    printf("#\nNombre de chiffre?");
    scanf("%d",&nombre);
    switch(nombre){
        case 2: printf("\n#deca-"); break;
        case 7: printf("\n#meca-"); break;
        case 10: printf("\n#giga-"); break;
        case 13: printf("\n#tera-"); break;
        case 16: printf("\n#peta-"); break;
        case 19: printf("n\#exa-"); break;
        case 22: printf("\n#zetta-"); break;
        case 25: printf("\n#yotta-"); break;
        default: printf("\n#Je ne sais pas");break;
    }
}
```

TP du chapitre 1/ Exercice 11: Jeu du juste numéro

Programme C

```
#include <stdio.h>
#include <stdlib.h>
#define MIN 10
#define MAX 20

int main()
{
    int alea();
    int nombre_cherche=alea(),nombre_entre;
     printf("EXERCICE 11:  Jeu du juste numéro\n");
    do{
        printf("\nEnter un nombre compris entre %d et %d : ",MIN,MAX);
        scanf("%d",&nombre_entre);
        if(nombre_cherche==nombre_entre){
            printf("BRAVO! Vous avez trouvé le nombre %d\n",nombre_cherche);
        }else{
            if(nombre_entre<nombre_cherche){
                printf("Plus grand !\n");
            }else{
                printf("Plus petit !\n");
            }
        }
    }while(nombre_cherche!=nombre_entre);
    return 0;
}
int alea(){
     srand(time(NULL));
     return (MIN+(100*rand())%(MAX-MIN));
}
```

TP du chapitre 1/ Exercice 15: minimum de plusieurs nombres

Programme C

```
/**Minimum de plusieurs nombres**/
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int nombre,compteur_niveau=0,niveau,maximum;
    printf("\nEXERCICE15:Minimum de plusieurs nombres ");
    printf("\nEntrer votre série de nombre: \n");
    scanf("%d",&nombre);
    compteur_niveau++;
    maximum=nombre;
    niveau=compteur_niveau;
    while(nombre!=0){
        if(nombre>maximum){
            maximum=nombre;
            niveau=compteur_niveau;
        }
        scanf("%d",&nombre);
        compteur_niveau++;
    }
    printf("\n#Maximum= %d",maximum);
    printf("\n#Niveau= %d",niveau);
}
```

TP du chapitre 1/ Exercice 16 :

Écrire un algorithme permettant de lire un montant (multiple de 5) et simule la remise de la monnaie. NB : Minimise le nombre de pièce d'une coupure de FCFA

Programme C

```
/**Réalise les remboursements sur la base d'un
montant multiple de 5**/
#include <stdio.h>
#include <stdlib.h>
int main(){
    int
coupures[9]={10000,5000,2000,1000,500,100,50,25,5};
    int remboursement, i, montant=0;
     do{
      printf("Le montant?: ");
       scanf("%d",&montant);
    }while(montant%5!=0);
    If(montant!=0){
        printf("#");
        for(i=0;i<9;i++){
            remboursement=montant/coupures[i];
            montant=montant%coupures[i];
            if(remboursement!=0){
                printf("%d
\t\"%dF\"\n",remboursement,coupures[i]);
            }
        }
    }
    return 0;
}
```

Algorithme

```
Algorithme remboursement{
    Type
       tRemboursement=Tableau[9] d'entiers
Variable
       tRemboursement : coupures ;
       entier remboursent,i ;
int remboursement, i, montant=0;
    Répéter {
      Écrire ("Le montant?: ");
       Lire (montant);
    }jusqu'à (montant%5!=0);
    Si (montant!=0){
        Écrire ("#");
        Pour (i←0(1)9){
            remboursement=montant/coupures[i];
            montant=montant%coupures[i];
            Si(remboursement!=0){
                Ecrire(remboursement +"
«  "+coupures[i]+" F »");
            }
        }
    }
}
```

TP du chapitre 2/ partie 3 / Exercice 2: suite de Fibonacci

```
/**Réalise les remboursements sur la base d'un montant multiple de 5**/
#include <stdio.h>
#include <stdlib.h>
int main(){
    int coupures[9]={10000,5000,2000,1000,500,100,50,25,5};
    int remboursement, i, montant=0;
     do{
      printf("Le montant?: ");
       scanf("%d",&montant);
    }while(montant%5!=0);
    If(montant!=0){
        printf("#");
        for(i=0;i<9;i++){
            remboursement=montant/coupures[i];
            montant=montant%coupures[i];
            if(remboursement!=0){
                printf("%d \t\"%dF\"\n",remboursement,coupures[i]);
            }
        }
    }
    return 0;
}
```

TP du chapitre 2/ partie 3 / Exercice 8: Récursivité

1. La condition d'arrêt X: i=MIN(ou i=MAX)
2. L'argument changé Y : Y=(i-1,result) ou Y=(i+1,result) si l'on a choisi la condition d'arrêt (i=MAX).
3. L'instruction d'appel : Rec(MAX,init) ou Rec(MIN,init) si l'on a choisi la condition d'arrêt (i=MAX).
4. Programme en C

```
/**Somme récursive des n premiers entiers**/
#include <stdio.h>
#include <stdlib.h>
int main(){
    int somme(int ,int);
    int n;
    printf("TP du chapitre 2/ partie 3 / Exercice 8");
    printf("\n#Le nombre?: ");
    scanf("%d",&n);
    printf("#La somme des %d premiers entiers:
%d",somme(n,0));
}
int somme(int i,int result){
    int a=result;
    if(i==0){
        return(result);
    }else{
        result+=i;
        result=somme(i-1,result);
    }
}
```

TP du chapitre 2/ partie 3 / Exercice 3:

1. Écrire en C les fonctions itératives et récursives de la factorielle.
2. Appeler l'une des fonctions pour afficher la combinaison de P dans N, P et N sont 2 entiers.

TP du chapitre 2/ partie 3 / Exercice 6: Calcul de N^p en versions itératives

Objectif : Conversion d'un nombre N de la base 10 à la base B.

TP du chapitre 2/ partie 3 / Exercice 7:

```
/**Resolution d'une equation du second
dégré**/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(){
    double calculDiscriminant(double
,double ,double );
    void solutionne(double ,double
,double );
    double* saisir();
    double a,b,c,D;
    double* val;
    printf("EXERCICE7: Résolution dans R
de l'equation aX^2+bX+c=0");
    val=saisir();
    /*Pour des besoins de compréhension
on ne manipulera pas val, mais plutôt a,
b et c* /
    a=val[0];
    b=val[1];
    c=val[2];

    if(a==0){
        if(b==0){
            if(c==0){
                printf("\n#Infinité de
solutions\n");
            }else{
                printf("\n#Aucune
solution\n");
            }
        }else{
            printf("\n#Unique solution:
%lf\n", -c/b);
        }
    }else{
        D=calculDiscriminant(a,b,c);
        solutionne(a,b,D);
    }
}
```

```
/**Calcule le discriminant de
l'équation aX^2+bX+c=0**/
double calculDiscriminant(double
a,double b,double c){
    return(b*b-4*a*c);
}
/**Recueille les variables fournit
par l'utilisateur**/
double* saisir(){
    double val[3];
    printf("\nEntrer la valeur de a:
");
    scanf("%lf",&val[0]);
    printf("\nEntrer la valeur de b:
");
    scanf("%lf",&val[1]);
    printf("\nEntrer la valeur de c:
");
    scanf("%lf",&val[2]);
    return val;
}
/**Calcule et affiche les solutions
d'une équation aX^2+bX+c sur la base
de a,b et D**/
void solutionne(double a,double
b,double D){
    double x,y;
    if(D==0){
        printf("\n#Unique solution:
%lf\n", -b/(2*a));
    }else{
        if(D<0){
            printf("\n#Pas de
solution réelles\n");
        }else{
            x=(-b-sqrt(D))/(2*a);
            y=(-b+sqrt(D))/(2*a);
            printf("\n#Deux solutions
: %lf\t%lf\n",x,y);
        }
    }
}
```

TP du chapitre 3/ partie 1 / Exercice 5:

Voir cours soutien

TP du chapitre 3/ partie 1 / Exercice 7:

Voir cours soutien

TP du chapitre 3/ partie 1 / Exercice 10:

Voir cours soutien

TP du chapitre 3/ partie 1 / Exercice 13:

Voir cours soutien

TP du chapitre 3/ partie 1 / Exercice 17: Fonctions de tri

1. Tri à bulle

Le tri à bulles ou tri par propagation est un algorithme de tri qui consiste à faire remonter progressivement les plus grands éléments d'un tableau, comme les bulles d'air remontent à la surface d'un liquide.

Le tri à bulles est souvent enseigné en tant qu'exemple algorithmique. Cependant, sa complexité est de l'ordre de n^2 en moyenne (où n est la taille du tableau), ce qui le classe parmi les mauvais algorithmes de tri. Il n'est donc quasiment pas utilisé en pratique.

Description de l'algorithme

L'algorithme parcourt le tableau, et compare les couples d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. Après chaque parcours complet du tableau, l'algorithme recommence l'opération. Lorsqu'aucun échange n'a lieu pendant un parcours, cela signifie que le tableau est trié. On arrête alors l'algorithme.

Voici la description en pseudocode du tri à bulle, pour trier un tableau T de n éléments numérotés de 0 à n-1 :

```
procédure tri_bulle(tableau T, entier n)
 Variable : boolén :aucun_échange;
                Entier :i, j, min;
début
    répéter
         aucun_échange = vrai
         pour j ← 0 (1) n - 2
             si (T[j] > T[j + 1]) alors
                 échange (T, j, j + 1)
                 aucun_échange = faux
    tant que (aucun_échange = faux)
fin
```

```
Procédure échange(tableau t, entier i, entier j)
    Variable :   réel :Temp;
   début
      Temp=T[i]
     T[i]=T[j]
     T[j]=Temp
   fin
```

2. Tri par sélection

Le tri par sélection (ou tri par extraction) est un algorithme de tri par comparaison. Il est particulièrement simple, mais inefficace sur de grandes entrées, car il s'exécute en temps quadratique en le nombre d'éléments à trier.

Description de l'algorithme

Sur un tableau de n éléments (numérotés de 1 à n), le principe du tri par sélection est le suivant :

- rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 1 ;
- rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 2 ;
- continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

En pseudocode, l'algorithme s'écrit ainsi :

```
procédure tri_selection(tableau t, entier n){
    Variable : réel :temp ;
               Entier :i, j, min;
   début
        pour i ← 1 (1) n − 1 faire
          min ← i
          pour j ← i + 1 (1) n - 1
              si t[j] < t[min], alors min ← j
          FinPour
        si min ≠ i, alors // on échange t[i] et t[min]
           échange(t,i,min);
        Finsi

   Fin
```

```
Procédure échange(tableau t, entier i, entier j)
    Variable : réel :Temp;
   début
      Temp=T[i]
     T[i]=T[j]
     T[j]=Temp
   fin
```

3. Tri par insertion

Le tri par insertion est un algorithme de tri classique dont le principe est très simple. C'est le tri que la plupart des personnes utilisent naturellement pour trier des cartes : prendre les cartes mélangées une à une sur la table, et former une main en insérant chaque carte à sa place.

En général, le tri par insertion est beaucoup plus lent que d'autres algorithmes comme le tri rapide et le tri fusion pour traiter de grandes séquences, car sa complexité asymptotique est quadratique.

Le tri par insertion est cependant considéré comme le tri le plus efficace sur des entrées de petite taille. Il est aussi très rapide lorsque les données sont déjà presque triées.

Description de l'algorithme

Dans l'algorithme, on parcourt le tableau à trier du début à la fin. Au moment où on considère le i-ème élément, les éléments qui le précèdent sont déjà triés. Pour faire l'analogie avec l'exemple du jeu de cartes,

lorsqu'on est à la i-ème étape du parcours, le i-ème élément est la carte saisie, les éléments précédents sont la main triée et les éléments suivants correspondent aux cartes encore mélangées sur la table.

L'objectif d'une étape est d'insérer le i-ème élément à sa place parmi ceux qui précèdent. Il faut pour cela trouver où l'élément doit être inséré en le comparant aux autres, puis décaler les éléments afin de pouvoir effectuer l'insertion. En pratique, ces deux actions sont fréquemment effectuées en une passe, qui consiste à faire « remonter » l'élément au fur et à mesure jusqu'à rencontrer un élément plus petit.

Voici une description en pseudocode de l'algorithme présenté. Les éléments du tableau T sont numérotés de 0 à n-1.

```
procédure tri_insertion(tableau T, entier n)
    Variable : réel :x ;
               Entier :i, j;
   début
        pour i ← 1 à n − 1 faire
           x ← T[i]
           j ← i
           Tantque (j > 0 et T[j - 1] > x) faire
               T[j] ← T[j - 1]
               j ← j - 1;
           FinTantQue

           T[j] ← x
        FinPour
  Fin
```

4. Tri rapide

Le tri rapide (en anglais quicksort) est un algorithme de tri fondé sur la méthode de conception diviser pour régner.

La complexité moyenne du tri rapide pour n éléments est proportionnelle à n log n, ce qui est optimal pour un tri par comparaison, mais la complexité dans le pire des cas est quadratique. Malgré ce désavantage théorique, c'est en pratique un des tris les plus rapides, et donc un des plus utilisés. Le pire cas est en effet peu probable lorsque l'algorithme est correctement implémenté. Le tri rapide ne peut cependant pas tirer avantage du fait que l'entrée est déjà presque triée. Dans ce cas particulier, il est par exemple plus avantageux d'utiliser le tri par insertion.

Description de l'algorithme :

La méthode consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui lui sont inférieurs soient à sa gauche et que tous ceux qui lui sont supérieurs soient à sa droite. Cette opération s'appelle le partitionnement. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

```
Procédure tri_rapide(tableau t, entier premier, entier dernier)
    Variable : Entier :pivot;
   début
      si (premier < dernier) alors
        pivot ← choix_pivot(t,premier,dernier)
        pivot ← partitionner(t,premier, dernier,pivot)
        tri_rapide(t,premier,pivot-1)
        tri_rapide(t,pivot+1,dernier)
      fin si
    fin
```

Choisir un pivot aléatoirement dans l'intervalle [premier, dernier] garantit l'efficacité de l'algorithme. Je vous laisse donc la latitude d'écrire la procédure **choix_pivot.** Vous pourrez également éviter d'écrire cette procédure en choisissant directement le premier, le dernier ou l'élément du milieu de tableau comme pivot.

```
Fonction partitionner(tableau T, premier, dernier, pivot) :entier
      Variable : réel : Temp ;
                 Entier :i, j;
début
     //on échange T[pivot] et T[dernier]
         Échange(T,pivot,dernier)
     j ← premier
     pour i ← premier (1) dernier – 1 faire
         si T[i] < T[dernier] alors
             échange (T, i, j)
             j ← j + 1
     échange( T, dernier, j)
     renvoyer j
fin
```

```
Procédure échange(tableau t, entier i, entier j)
    Variable : réel :Temp;
   début
      Temp=T[i]
     T[i]=T[j]
     T[j]=Temp
   fin
```

5. Tri fusion

Le tri fusion est un algorithme de tri dont la complexité temporelle pour une entrée de taille n est de l'ordre de n log n (ce qui est asymptotiquement optimal). Le tri fusion se décrit naturellement sur des listes (Voir CHAP 4), mais fonctionne aussi sur des tableaux.

Ce tri est basé sur la technique algorithmique diviser pour régner (DPR). L'opération principale de l'algorithme est la fusion, qui consiste à réunir deux listes triées en une seule. L'efficacité de l'algorithme vient du fait que deux listes triées peuvent être fusionnées en temps linéaire.

La version la plus simple du tri fusion sur les tableaux a une efficacité comparable au tri rapide, mais elle n'opère pas en place : une zone temporaire de données supplémentaire de taille égale à celle de l'entrée est nécessaire.

Description de l'algorithme :

L'algorithme peut être décrit récursivement :

1. On découpe en deux parties à peu près égales les données à trier
2. On trie les données de chaque partie
3. On fusionne les deux parties

La récursivité s'arrête car on finit par arriver à des listes composées d'un seul élément et le tri est alors trivial.

En pseudocode, l'algorithme pourrait s'écrire ainsi :

```
fonction triFusion(liste0) :
début
   si longueur(liste0) <= 1, renvoyer liste0
     sinon
        soit (liste1, liste2) = scinder(liste0)
        renvoyer fusionner(triFusion(liste1), triFusion(liste2))
Fin
```

```
fonction scinder(liste0)
debut
   si longueur(liste0) <= 1, renvoyer le couple
(liste0,liste_vide)
   sinon,
     soient e1 et e2 les deux premiers éléments de liste0, et
reste le reste de liste0
     soit (liste1, liste2) = scinder(reste)
     renvoyer le couple de listes (liste de tête : e1 et de  queue
: liste1,      liste de tête : e2 et de queue : liste2)
fin
```

```
fonction fusionner(liste1, liste2)
debut
     si la liste1 est vide, renvoyer liste 2
     sinon
        si la liste2 est vide, renvoyer liste 1
        sinon
         si tête(liste 1) <= tête(liste2), renvoyer la liste de
tête : tête(liste1) et la liste de queue :
fusionner(queue(liste1),liste2)
          sinon
             renvoyer la liste de tête : tête(liste2) et la
liste de queue : fusionner(liste1,queue(liste2))
fin
```

TP du chapitre 3/ partie 1 / Exercice 18: Guide : algorithme général de la recherche dichotomique

```
Algorithme recherche_dichotomique
//déclarations
 début, fin, val, mil : Entiers
 t : Tableau [0..100] d'entiers classé
 trouvé : Booléen

début
//initialisation
 début ← 0
 fin ← 100
 trouvé ← faux
 Saisir val

//Boucle de recherche
  Répéter
   mil ← partie entière( début + ((fin-début) / 2) )
   Si t[mil] = val alors
     trouvé ← vrai
   Sinon
     Si val > t[mil] Alors
       début ← mil+1

     Sinon
       fin ← mil-1

     FinSi
    FinSi
   /**La condition début inférieur ou égal à fin permet d'éviter de faire
   une boucle infinie si 'val' n'existe pas dans le tableau.**/
   Tant que trouvé = faux ET début ≤ fin
        //Affichage du résultat
        Si trouvé Alors
                Ecrire ("La valeur ", val , " est au rang ", mil)
        Sinon
                Ecrire ("La valeur ", val , " n'est pas dans le tableau")
        FinSi
fin
```

TP du chapitre 3/ partie 1 / Exercice 19:

Voir cours de soutien

ENFIN !!!!!

Qu'il me soit maintenant permis en mon nom propre et en celui de toute la grande équipe qui a travaillé à la réalisation de ce chef d'œuvre de te transmettre mes ***sincères encouragements*** et ***mes souhaits de réussite.***

Toute Fois Je Les Remercie Pour Les Efforts Fourni

BONNE CHANCE A TOI