I Quelques points d'algorithmique

1) Types

Type "simple" : Identificateur commençant par une majuscule = Description du type

Les principaux types : les entiers naturels (int, long, long long) ou relatifs (précisé unsigned devant), les réels (float, double), les caractères (char)

à noter qu'un mot est un tableau de caractères : char[NB_CARACTERES]

et que les booléens sont à définir de la manière suivante :

typedef enum {FAUX,VRAI} Booleen;

Type énumération : spécifie la liste des valeurs du type (voir booléen)

Type composé

- Construit à partir d'autres types
  -> Notion de composant
  - Type des composants
  - Organisation de ces composants
  - accès à ces composants

Structure : Regroupement d'informations de différents types

- Les composants sont appelés champs
- Opérations :
  - Sur le composé : l'affectation et l'accès aux composants
  - Sur les composants : celles autorisées sur leurs types

Exemple :

```
typedef enum {TSP, TEM} Specialite;
typedef struct { char prenom[LONGUEUR];
                 int age;
                 Specialite spe;} Eleve;
```

eleve.spe dit si eleve est TSP ou TEM

2) 3 principaux schémas

a) schéma conditionel

```
if (CONDITION)
{
   ALORS INSTRUCTION;
}
```

optionnel :

```
else
{
   INSTRUCTION;
}
```

b) schéma itératif "Tant que"

```
while (CONDITION)
{
   INSTRUCTION;
}
```

c) schéma itératif "Pour"

```
for (i=DEBUT;i<FIN;i++)
{
  INSTRUCTION;
}
```

3) Récursivité

Principé : Fonction définie à partir d'elle-même.

On doit aussi avoir une condition d'arrêt. Elle est donc souvent construite avec un si.

Exemple :

```
fonction factorielle( n :Naturel ) : Naturel

    si n ≤ 1 alors                          <----- Condition d'arrêt (n<=1)
       retourner 1
    sinon
       retourner n × factorielle ( n-1 )    <----- Appel à elle-même
    fsi

ffct factorielle
```

II Exemple classique : les tris

1) Tri par sélection

Principe : Dans notre tableau à trier, on a une première partie trié (initialement vide), et une autre trié. A chaque itération, on va chercher le minimum de la partie non triée pour le placer à la suite de la partie triée.

Complexité $O(n^2)$

```
procédure tri_selection(tableau t, entier n)
      pour i de 1 à n - 1
          min ← i
          pour j de i + 1 à n
              si t[j] < t[min], alors min ← j
          fin pour
          si min ≠ i, alors échanger t[i] et t[min]
      fin pour
  fin procédure
```

2) Tri par insertion

Principe : comme pour le prix par sélection, on a une partie triée et une non triée. Cette fois ci, on prend le premier élément de la liste non triée et on vient le placer à la bonne position dans la partie triée

Complexité $O(n^2)$

```
 procédure tri_insertion(tableau T, entier n)
      pour i de 1 à n-1
          x ← T[i]
          j ← i
          tant que j > 0 et T[j - 1] > x
              T[j] ← T[j - 1]
              j ← j - 1
          fin tant que
          T[j] ← x
     fin pour
  fin procédure
```

3) Tri à bulles

A chaque itération, on compare deux éléments consécutifs et on les inverse si l'ordre n'est pas le bon.

Complexité $O(n^2)$

```
procédure tri_bulle(tableau T)
    n = longueur(T)
    pour i de n - 1 à 1
        aucun_échange = vrai
        pour j de 1 à i
            si T[j] > T[j + 1], alors
                échanger T[j] et T[j + 1]
                aucun_échange = faux
            fin si
        fin pour
        si aucun_échange : fin procédure
    fin pour
fin procédure
```

4) Tri rapide

Ici, on choisit un pivot (de la manière qu'on veut : premier, dernier, le milieu). Puis on partitionne le tableau à trier suivant si les éléments sont inférieurs ou supérieurs au pivot. On se rappelle récursivement sur les deux sous-tableaux, le pivot et alors à sa position finale.

Complexité O(nlog(n))

```
partitionner(tableau T, premier, dernier, pivot)
```

```
    échanger T[pivot] et T[dernier]
    j := premier
    pour i de premier à dernier - 1
        si T[i] <= T[dernier] alors
            échanger T[i] et T[j]
            j := j + 1
    échanger T[dernier] et T[j]
    renvoyer j

tri_rapide(tableau t, entier premier, entier dernier)
  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
```

5) Tri fusion

Le principe est de diviser le tableau en deux, puis diviser chaque sous-tableaux en deux par récursivité, et ainsi de suite, jusqu'à ce qu'il n'y est plus qu'un élément. Il est alors bien placé dans son sous-tableau, on fusionne alors les sous-tableaux deux à deux en veillant à conservé l'ordre.

Complexité O(nlog(n))

```
fonction scinder(liste0) :
    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)

  fonction fusionner(liste1, liste2) :
    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 de queue : fusionner(queue(liste1),liste2)
       sinon, renvoyer la liste de tête : tête(liste2) et de queue :
fusionner(liste1,queue(liste2))

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

III Un Programme

1) stdlib.h et stdio.h

stdlib : contient la définition des constantes symboliques EXIT_SUCCESS (=0) et EXIT_FAILURE (= 1)

En cas de déroulement normal, la fonction main renvoie EXIT_SUCCESS, sinon EXIT_FAILURE.

stdio : contient les fonctions de base printf et scanf

2) scanf et printf

scanf : permet la saisie d'une donnée.

Lorsque l'on demande à l'utilisateur de rentrer une valeur pour une variable, on l'utilise de la façon suivante : scanf("%d", &NOM_DE_LA_VARIABLE);

(%d pour int, %ld pour long, %f pour float,%lf pour double,%c pour char, %s chaîne de caractères)

printf : permet d'afficher un texte et/ou des variables (dépend du type).

Exemple : printf("%d", NOM_DE_LA_VARIABLE) ;

3) la fonction main, fonction et procédure

Fonction : renvoie une valeur grâce à un return, de nimporte quel type du moment que c'est celle annoncé avant le nom de la fonction

Procédure : fonction de type void (vide), ne renvoie donc rien par un return, peut en renvoyer par utilisation des parametres en resultat

La fonction main doit être de type int, ne prendre aucun paramètre en entrée, et finir par un "return EXIT_SUCCESS" (ou un return EXIT_FAILURE si il y a un cas où le programme plante)

```
int main () {
blablabla;
return EXIT_SUCCESS;
}
```

4) passage de paramètres

Paramètres données :

Type_resultat nomDeLaFonction (Type_parametres_donnée parametre)

Paramètres résultats

Type_resultat nomDeLaFonction (Type_parametres_resultats *parametre)

Le "*" signifie qu'on donne en réalité l'adresse du paramètre, le * peut se traduire par "objet adressé par", ce qui permet de récupérer le resultat dans une autre fonction sans utiliser return. Ceci peut être pratique lorsqu'on veut renvoyer plusieurs résultats à la fonction main.

Le "&" peut être traduit par "l'adresse de cet objet", il est souvent utilisé lorsqu'on fait appel à une fonction utilisant des paramètres résultats (exemple scanf)

```
int main () {
        fonction(donnée1, donnée2, &donnée_resultat);
}
```

5) modules, compilation et makefile

Lorsqu'on programme en module on utilise des fichiers.h et des fichiers.c

Les fichiers.c contiennent le code des fonctions que vous programmez, l'utilisateur n'y a pas accès.

Les fichiers.h contiennent la declaration des fonctions présentes dans le fichier.c et est accessible à l'utilisateur, ainsi il peut utiliser les fonctions sans en voir le code et comprendre comment elles marchent.

Dans un .h :
        on met le prototype des fonctions, c'est-à- dire leur type, nom et arguments (exemple : int factorielle(int n);)
        les constantes, headers et bibliothèques (exemple : #DEFINE N 10 ou #include"stdio.h")

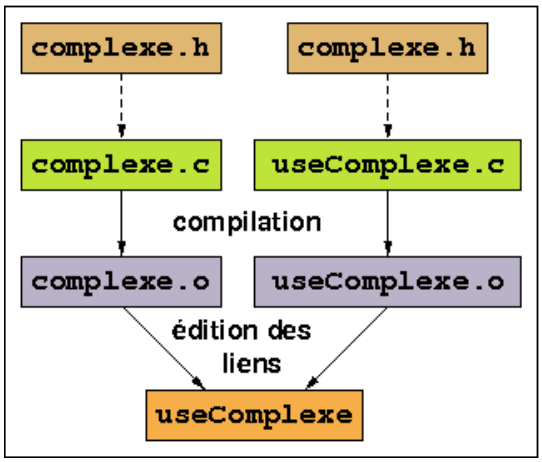
Dans le .c :
        on met surtout la définition des fonctions
        -> moins il y a de choses, mieux c'est !

Pour compiler un .c seul : gcc -Wall FICHIERACOMPILER.c -o FICHIERCOMPLIE
On peut avoir besoin des plusieurs fichiers.c dans un autre fichier.c, on les compile alors pour en faire des fichiers.o : gcc -Wall FICHIERACOMPILER.c
Les fichiers.o sont des morceaux d'éxécutable, ils seront rassemblés lors de l'édition de liens.

Makefile :

Le makefile est un fichier texte contenant les instructions dans l'ordre pour compiler l'intégralité des modules programmés et en faire un éxécutable.

Pour faire un makefile qui compile plusieurs fichiers :
Imaginons la situation suivante : un fichier useComplexe.c dans lequel on déclare une fonction useComplexe qui a besoin d'un autre fichier complexe.o.
On va d'abord dire à la machine de générer useComplexe puis chacuns des .o.




```
useComplexe: useComplexe.o complexe.o
        gcc -Wall useComplexe.o complexe.o -o useComplexe

useComplexe.o: useComplexe.c useComplexe.h
        gcc -Wall useComplexe.c -c useComplexe.o

complexe.o: complexe.c complexe.h
        gcc -Wall complexe.c -c complexe.o
```

Lorqu'on utilise une librairie, ajouter l'option -lNOM-LIBRAIRIE (juste avant le-o pour le makefile)

Un makefile doit être constitué de déclarations sur deux lignes :
module_cible : liste des modules dont il a besoin
<TAB> gcc etc

PS : n'écrivez pas <TAB>, faites simplement une tabulation

ces déclarations se font de la manière suivante : les cibles n'ont besoin que de modules situés en dessous dans le makefile

Exemple :

```
prgm_principal : prgm1.o prgm2.o
	gcc -Wall prgm1.o prgm2.o -o prgm_principal

prgm1.o : prgm2.h prgm1.c
	gcc -Wall prgm1.c

prgm2.o : prgm2.h prgm2.c
	gcc -Wall prgm2.c
```

On utilise la commande make qui lit le fichier makefile et execute les instructions de compilations (et seulement les nécessaires)

Lorsqu'on utilise une librairie, dans le terminal, si il n'arrive pas à trouver la libraire, utiliser la commande export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:. pour exporter la variable du chemin.

IV Les pointeurs

1) Qu'est ce qu'un pointeur

Lorsqu'on déclare une variable(int a par exemple), un espace mémoire est reservé. On a donc deux choses différentes : le contenu de cet espace (a) et son adresse (&a).

Un pointeur est une variable qui contient cette adresse, c'est une sorte de flèche qui pointe vers notre espace mémoire.

On déclare un pointeur de la mainière suivante :

int pointeur; le symbole "*" signifie qu'on utilise un pointeur

pointeur est donc une adresse, et si on veut utiliser le contenu de l'espace vers lequel il pointe, on utilise *pointeur

exemple : la fonction echanger

```
void echanger ( int *pt1, int *pt2) {
	int aux = *pt1;
	*pt1=*pt2;
	*pt2= aux;
}
```

malloc et free : dans l'exemple suivant, on peut avoir besoin d'un espace mémoire inconnu au moment de l'écriture du programme. Pour résoudre se problème, il faut demander au processeur de nous donner de l'espace grâce à la fonction malloc, qui revoie un pointeur *void sur notre espace reservé

exemple : `t=malloc(nbElements*sizeof(int))`

On veut ici une place pour un certain nombre d'entiers, défini par l'utilisateur durant l'éxécution.

On utilise free(t); pour rednre de nouveau cet espace libre après notre utilisation.

2) Interêts : exemple du tableau dynamique

On peut avoir envie d'écrire un programme sur des tableaux de tailles choisis par l'utilisateur, pour cela on utilise les tableaux dynamiques.

```
#include <stdlib.h> // malloc renvoie un pointeur générique (void *)
#include <stdio.h> // pour printf et scanf

int main() {
      int nbElements, i, *t1D; // t1D est un pointeur sur int
      printf("nombre d'éléments : ");
      scanf("%d",&nbElements);
      if((t1D=malloc(nbElements*sizeof(int)))==NULL) { // allocation mémoire
            printf("erreur allocation mémoire"); // test de la valeur de retour
            return EXIT_FAILURE; // NULL si échec
      }
      for (i=0;i<nbElements;i++) { // même utilisation qu'un tableau statique
            printf("t1D[%d] ? ",i);
            scanf("%d ",&t1D[i]); // &t1D[i] ou t1D+i
      }
      free ( t1D ); // désallocation
      return EXIT_SUCCESS;
}
```

Enfin, de manière plus pragmatique, si on travaille avec des objets lourds, il est moins coûteux de donner à une fonction un pointeur vers l'objet que l'objet lui même.

Trucs et astuces :

En c, au lieu d'écrire i=i+1, on écrit i++. De même, i+=5 ~ i=i+5.

types	Caractère, Entier, Réel	`char, int, double`
Définitions d'objets	objet1, objet2 : NomType	`NomType objet1, objet2 ;`
Actions simples	instruction	`expression ;`
Actions composées (N>1)	instruction 1 ...	`{ expression 1 ;` `...`

	instruction N	`expression N ; }`
Affectation	var valeur	`var = valeur ;`

Conditionnelle (1ère forme)	**si** condition **alors**
	  action 1
	sinon
	  action 2
	fsi
		`if ( condition )`
		`  action 1`
		`else`
		`  action 2`

Itération (forme canonique)	**tant que** condition **faire** action **ftq**	`while ( condition )` `  action`

Itération (variante)	**faire**
	  action
	tant que (condition)
		`do`
		`  action`
		`while ( condition ) ;`

*Itération (schéma **pour**)*	**pour** i = val1 : Naturel **à** val2 **pas** h action **fpour** i	`for (i=val1 ; i<=val2 ; i=i+h)` `  action`
Deux opérateurs	**div** **mod**	`/ // opérandes de type entier` `%`
Deux actions simples	saisir (x) afficher (y)	`scanf ("%d", &x ) ;` `printf ("%d ", y ) ;`