JI3-2011

3èmes Journées Informatiques des CPGE

Agadir, du 12 au 14 mai 2011

Algorithmique Programmation

Driss El Ghanami, Ecole Mohammedia des Ingénieurs

SERIE D'EXERCICES EN INFORMATIQUE

Proposée par Pr. D. El Ghanami

Ecole Mohammadia d'Ingénieurs

Mai 2011

Enoncés

Exercice 1.1 :

Les types manipulés en algorithmique sont : Entier, Réel, Caractère et Booléen (les types qui peuvent être représentés en binaire). Donner le type et la valeur des expressions suivantes :

1. *2 + 3 * 4*
2. *2.0 + 3 * 4*
3. *vrai et (faux ou vrai)*
4. *(2 < 3) et (4 > 5)*

Exercice 1.2 :

Parmi les instructions suivantes, lesquelles sont correctement écrites (justifier votre réponse en indiquant le type possible de chaque variable).

1. *z ← (x > 2) et (y < 5)*
2. *z ← (x > 2) et y*
3. *z ← (x > 2 et y) < 5*
4. *z ← (x + y) > 2 et y*
5. *si (y ≠ 0) alors z ← x/y sinon z ← faux ; finsi ;*

Exercice 1.3 :

Donner la table de vérité des expressions booléennes suivantes :

- *(a et non b) ou c*
- *(a et non b) ou (non a et b)*

Exercice 1.4 :

Soient *x, y, z, t* des entiers. Donner les expressions booléennes correspondant aux situations suivantes:

1. *Les valeurs de x et de y sont toutes les deux supérieures à 3*
2. *Les variables x, y et z sont identiques*
3. *Les valeurs de x, y et z sont identiques mais différentes de celle de t*
4. *Les valeurs de x sont strictement comprises entre les valeurs de y et t*
5. *Parmi les valeurs de x, y et z deux valeurs au moins sont identiques*
6. *Parmi les valeurs de x, y et z deux valeurs et seulement deux sont identiques*
7. *Parmi les valeurs de x, y et z deux valeurs au plus sont identiques*

Indication : Utiliser les parenthèses si l'expression logique comporte des Ou et des Et

Exercice 1.5 :

Quelles seront les valeurs des variables *a*, *b* et *c* après exécution des instructions suivantes :

a ← 1 ;
b : ← 5 ;
c ← a – b ;
a ← 2 ;
c ← a + b ;

Exercice 1.6 :

On considère trois variables entières *x, y, z*. Donner des expressions booléennes pour déterminer si :

1. *x* est pair.
2. *x* est impair.
3. *x* et *y* ont la même parité.
4. L'une au moins des trois est paire.
5. Deux d'entre-elles au moins ont la même parité.
6. Exactement deux sur les trois sont paires.

Indication : Utiliser la fonction mod qui permet d'avoir le reste de la division de deux entiers.

Exercice 1.7 :
Ecrire un algorithme qui permet d'échanger les valeurs de deux variables entières *a* et *b*.
Indication : Lors de l'affectation, la valeur précédente d'une variable est remplacée par la nouvelle.

Exercice 1.8 :
Ecrire un algorithme qui affiche à l'écran "Bonjour"

Exercice 1.9 :
Écrire un algorithme qui à partir de 3 notes d'un étudiant et 3 coefficients calcule et affiche la moyenne.
Indication : Définir d'abord les données d'entrées et de sorties du problème, leur type et par la suite le traitement à faire.

Exercice 1.10 :
Écrire un algorithme qui à partir d'une somme d'argent donnée, donne le nombre minimal de : billets de 50Dh, 20Dh, les pièces de 2Dh et de 1Dh qui la compose.
Indication : On suppose que le montant est la différence entre le prix à payer par un client dans un magasin et le montant qu'il donne au caissier.
Pour avoir un minimum de billets et de pièces à rendre, il faut maximiser le nombre de billets de grandes valeurs et minimiser celui de pièces de petites valeurs.

Exercice 1.11 :
Écrire un algorithme qui permet d'effectuer une permutation circulaire des valeurs entières de trois variables *x, y, z* (la valeur de *y* dans *x*, la valeur de *z* dans *y* et la valeur de *x* dans *z*).
Indication : Sauvegarder la valeur d'une variable et commencer à partir d'elle.

Exercice 2.1 :
Écrire un algorithme qui détermine si une année est bissextile ou non. Les années bissextiles sont multiples de *4*, mais pas de *100*, sauf pour les millénaires qui le sont.

Exercice 2.2 :
Écrire un algorithme qui prend en entrée trois entiers et qui les trie par ordre croissant.

Exercice 2.3 :
Ecrire un programme qui lit deux valeurs entières quelconques (*A* et *B*) au clavier et qui affiche le signe de la somme de *A* et *B* sans faire l'opération de l'addition.
Exemple : $A = -8, B = 3$ le signe de la somme est négatif.

Exercice 2.4 :
Écrire un algorithme qui pour un temps donné (représenté sous la forme : heure, minute, seconde) affiche le temps (sous la même représentation) après avoir ajouté une seconde.
On suppose que $0 \leq h < 24, 0 \leq m < 60, 0 \leq s < 60$.

Exercice 2.5 :
Écrire un algorithme qui détermine le numéro d'un jour dans l'année en fonction du jour, du mois, et de l'année.

Exercice 2.6 :
Programmer une petite calculatrice qui demande à l'utilisateur une opération à effectuer sous forme de caractère (par exemple '*', '+', '-', '/'), demande ensuite *2* nombres et effectue le calcul demandé et affiche le résultat.

Exercice 2.7 :

Dans une entreprise, le calcul des jours de congés payés s'effectue de la manière suivante : si une personne est entrée dans l'entreprise depuis moins d'un an, elle a droit à deux jours de congés par mois de présence, sinon à *28* jours au moins. Si c'est un cadre et s'il est âgé d'au moins *35* ans et si son ancienneté est supérieure à *3* ans, il lui est accordé *2* jours supplémentaires. S'il est âgé d'au moins *45* ans et si son ancienneté est supérieure à *5* ans, il lui est accordé *4* jours supplémentaires, en plus des *2* accordés pour plus de *35* ans. Écrire un algorithme qui calcule le nombre de jours de congés à partir de l'âge, l'ancienneté et l'appartenance au collège cadre d'un employé.

Exercice 2.8 :

Résolution d'une équation du second degré dans C : $a.x^2 + b.x + c = 0$

Exercice 3.1 :

1- Écrire un algorithme qui calcule la somme des *1/i* pour *i* allant de *1* à *n* pour une valeur de *n* rentrée au clavier. Faire cette somme en partant de *1* à *n* et la recalculer en partant de *n* à 1. Comparer ces deux sommes et conclure pour différentes valeurs de *n*.

2- Chercher la valeur de *n* à partir de laquelle les deux sommes deviennent différentes.

Remarque : pour utiliser un entier en tant que réel, il suffit de le multiplier par *1.0*

Exercice 3.2 :

Écrire un algorithme qui effectue la multiplication de deux entiers positifs (notés *x* et *y*) donnés en utilisant uniquement l'addition entière.

Exercice 3.3 :

Afficher la table de conversion entre les degrés Fahrenheit et Celsius de *250* à *−20* degré F par pallier de *10* degrés. On passe de *x* degré F au degré C en calculant (*5/9 x − 160/9*).

Exercice 3.4 :

Écrire un algorithme qui dessine, à l'aide du signe '+', les figures de hauteur n suivantes :

N.B : on suppose l'existence de la procédure retourLigne() qui permet un retour à la ligne.

1-

```
+
+       +
+       +       +
```

2-

```
                +
        +       +       +
+       +       +       +       +
```

3-

```
+       +       +
+               +
+       +       +
```

4-

```
                +
        +               +
+                               +
        +               +
                +
```

Exercice 3.5 :

Une école primaire veut disposer d'un logiciel pour enseigner aux enfants quelques rudiments d'arithmétique, en particulier les tables de multiplication. Le fonctionnement d'un tel système doit

permettre à tout élève de vérifier ses connaissances au cours d'une session (suite de questions du système, réponses de l'élève). Chaque session se déroulera ainsi :

- Le système propose deux nombres, entre 0 et 10, tirés au hasard par la fonction hasard(10).
- L'élève en donne le produit.
- En cas de réponse, un message s'affiche et une nouvelle question est posée.

La fin de la session survient quand l'élève a fourni 20 bonnes réponses ou 10 fausses avec droit à trois essais successifs maximum pour un essai donné.
Ecrire un algorithme qui simule le jeu.
N.B. La fonction rand() permet de générer une valeur aléatoire.

Exercice 3.6 :
Écrire un algorithme qui vérifie si N un entier positif est un carré parfait.
Exemple : 4, 9, 16 sont des carrés parfaits.

Exercice 3.7 :
Ecrire un algorithme qui permet de calculer et d'afficher le nombre d'occurrences d'un chiffre ($0 \leq chiffre < 10$) dans un nombre positif.
Exemples : L'occurrence du chiffre *7* dans le nombre *778* est *2*.
L'occurrence du chiffre *8* dans le nombre *20681* est *1*.
L'occurrence du chiffre *5* dans le nombre *2771* est *0*.

Exercice 3.8 :
Trouver tous les entiers entre 1 et n vérifiant $x^2 + y^2 = z^2$
Exemples de résultats : *(8, 6, 10), (8, 15, 17)*
Indication : Tester la formule pour un triplet (x, y, z) puis inclure les actions dans un ensemble de répétitions imbriquées afin d'obtenir tous les triplets possibles.

Exercice 3.9 :
Lorsque *x* est proche de *0*, *arcsin (x)* peut être approximé à l'aide de la formule suivante :

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

Écrire un algorithme qui calcule une approximation de *arcsin(x)*.

Exercice 3.10 :
Ecrire un algorithme qui recherche le premier nombre entier naturel dont le carré se termine par *n* fois le même chiffre.
Exemple : pour $n = 2$, le résultat est *10* car *100* se termine par *2* fois le même chiffre

Exercice 3.11 :
Ecrire un algorithme qui simule le problème de Syracuse définit par :
$u_0 = m > 1$
$u_{n+1} = u_n/2$ si u_n est pair
$u_{n+1} = 3u_n+1$ sinon
Exemple : *6 → 3 → 10 → 5 → 16 → 8 → 4 → 2 → 1*

Rappel sur les tableaux
Déclaration de tableau :
int T[3] ; déclare un tableau *T* avec réservation de *3* cases mémoires de la taille d'un entier.
int T[3] = {12, 14, 6} ; déclare *T* un tableau d'entiers initialisés de dimension *3*.
Déclaration de matrice :
int T[2][3] ; déclare *T* comme un tableau de dimension *2* de tableaux de *3* entiers.
int T[2][3] = {{1, 5, 1}, {2, 2, 0}} ; déclare *T* une matrice d'entiers initialisés de dimension *2*x*3*.

Un tableau ou une matrice des paramètres d'une fonction sont passés par référence.
L'indice d'un tableau commence de *0* en langage C et de *1* en pseudo code (en algo).
Procédure utilisées :
On donne les procédures suivantes qui permettent d'initialiser un tableau ou une matrice.
remplirTab(T) remplit un tableau *T* d'entiers par *N* valeurs.
remplirMat(T) remplit une matrice *T* (*N*x*M*) d'entiers par des valeurs.

Exercice 4.1 :
Ecrire un algorithme qui affiche un tableau à l'envers.

Exercice 4.2 :
Soit un tableau *t* d'entiers. Écrire un algorithme qui change de place les éléments de ce tableau de telle façon que le nouveau tableau *t* soit une sorte de "miroir" de l'ancien.
Exemple : *1 2 4 6* → *6 4 2 1*

Exercice 4.3 :
Ecrire un algorithme qui vérifie si une chaîne est un carré ou pas.
Définition : Une chaîne de caractères est un carré si elle se compose de 2 chaînes identiques.
Exemple : "chercher" et "bonbon" sont des carrés.

Exercice 4.4 :
Un programme qui permet de :
- Lire à partir du clavier un nombre entier *N* ($0 < N \leq 10$)
- Lire à partir du clavier N entiers et les mettre dans un tableau t
- Décaler chaque élément du tableau t vers le haut (le $2^{ème}$ élément devient le 1^{er}, le $3^{ème}$ devient le $2^{ème}$, …, le 1^{er} devient dernier)

Exemple : *t = {-3, 0, 5, -8, 1, 4}*
Décalage vers le haut *t = {0, 5, -8, 1, 4, -3}*

Exercice 4.5 : Recherche dans un tableau
1- Ecrire un algorithme qui recherche l'indice d'une valeur dans un tableau quelconque
Principe : Rechercher l'indice d'une valeur dans un tableau quelconque, parcourir le tableau jusqu'à trouver cette valeur et s'arrêter (recherche séquentielle).
2- Ecrire un algorithme qui recherche l'indice d'une valeur dans un tableau trié.
Principe : Le tableau est trié c'est-à-dire que les éléments sont rangés dans l'ordre (en général croissant). L'algorithme précédent s'exécute parfaitement, mais comment l'améliorer ?
Pour une valeur qui existe dans le tableau, aucune différence : on s'arrête dès qu'on l'a trouvée ; mais, pour une valeur qui ne se trouve pas dans le tableau, on peut s'arrêter dès que l'on arrive sur une valeur trop grande, on forcera donc la boucle à s'arrêter.

Exercice 4.6 :
Écrire un algorithme qui à partir d'un tableau d'entiers *t* d'au moins un entier, fournit le nombre de sous-séquences croissantes de ce tableau, ainsi que les indices de début et de fin de la plus grande sous-séquence.
Par exemple, soit t un tableau de *15* éléments :

1, 2, 5, 3, 12, 25, 13, 8, 4, 7, 24, 28, 32, 11, 14

Les séquences strictement croissantes sont :

< 1, 2, 5 >;< 3, 12, 25 >;< 13 >;< 8 >;< 4, 7, 24, 28, 32 >;< 11, 14 >

Le nombre de sous-séquence est : 6 et la plus grande sous-séquence est :

< 4, 7, 24, 28, 32 >

Exemple 4.7 :

Ecrire un algorithme qui permet de remplir et d'afficher une matrice par des valeurs saisies.

Exercice 4.8 : Moyenne de notes par élève et par matière

On considère une matrice de notes dont les *M* lignes correspondent à *m* élèves et dont les *n* colonnes correspondent à *N* matières.

De plus on dispose d'un vecteur dont les *N* valeurs correspondent à des coefficients.

On demande de calculer la moyenne par élève et, distinctement, de calculer la moyenne par matière

Exercice 4.9 : Produit matriciel

Écrire un algorithme qui réalise le produit *C* de deux matrices *A* et *B*.

Méthode : En multipliant une matrice *A(N,M)* avec une matrice *B(M,P)* on obtient une matrice *C(N,P)*.

La multiplication de deux matrices se fait en multipliant les composantes des deux matrices lignes par colonnes :

$$C_{i,j} = \sum_{k=1}^{M} A_{i,k} * B_{k,j}$$

Exercice 4.10 :

Ecrire un algorithme qui affiche un triangle de Pascal d'ordre n.

1					$(a+b)^0 = 1$
1	*1*				$(a+b)^1 = 1*a + 1*b$
1	*2*	*1*			$(a+b)^2 = 1*a^2 + 2*a*b + 1*b^2$
1	*3*	*3*	*1*		$(a+b)^3 = 1*a^3 + 3*a^2*b + 3*a*b^2 + 1*b^3$
1	*4*	*6*	*4*	*1*	$(a+b)^4 = 1*a^4 + 4*a^3*b + 6*a^2*b^2 + 4*a*b^3 + 1*b^4$

Indication : il ne faut pas utiliser la Formule du triangle de Pascal suivante :

$$C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$$

Exercice 5.1 :

Ecrire une fonction qui à partir d'un réel retourne la valeur absolue de ce réel.

Exercice 5.2 :

Ecrire une fonction qui à partir d'un nombre entier strictement positif retourne *VRAI* si ce nombre est pair et *FAUX* sinon.

Exercice 5.3 :

Ecrire une fonction qui retourne le nombre de chiffres dans un nombre.

Exemple: Pour le nombre entier *21668*, la fonction retourne *5* (le nombre de chiffres).

Exercice 5.4 :

Ecrire un algorithme qui permet de calculer la distance nécessaire à une voiture pour s'arrêter, connaissant la vitesse de la voiture en km/h et l'état de la route : sèche ou mouillée.

La formule de calcul de la distance est :

Distance d'arrêt = chemin de réaction + distance de freinage.

Sachant que la réaction du conducteur est :

Le chemin de réaction = 3mètres pour 10km/h.

Distance de freinage :

Distance sur route mouillée = vitesse^2/100 *(ex. à 20km/h, on a 4m).*

*Distance sur route sèche = 3/4 * Distance sur route mouillée.*

L'algorithme permet aussi de retournez le problème et calcule la vitesse de la voiture d'après la longueur des traces de freinage (les traces de freinage ne représente pas la réaction du conducteur).

Exercice 5.5 :

Écrire une fonction qui à partir d'un entier strictement positif donné, retourne le résultat booléen *VRAI* ou *FAUX* selon que le nombre est premier ou non.
Afficher la liste des nombres premiers inférieurs à *n*
Exemple : *n = 50* donne *1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47.*

Exercice 5.6 :
Écrire un algorithme qui affiche la suite de tous les nombres parfaits inférieurs ou égaux à un nombre, entier positif donné noté *n*. Un nombre est dit parfait s'il est égal à la somme de ses diviseurs stricts.
Exemple : *28 = 1 + 2 + 4 + 7 + 14*
Voici la liste des nombres parfaits inférieurs à *10000* : *6, 28, 496, 8128.*
Indication : Pour un nombre donné, chercher les diviseurs et en faire la somme ; vérifier que le nombre est parfait, puis inclure ces actions dans une répétition permettant de tester tous les nombres.

Exercice 5.7 :
Programmer une procédure qui échange les valeurs de deux entiers.

Exercice 5.8 :
Donner un algorithme qui permet d'afficher les nombres jumeaux compris entre *1* et *n*.
On dit que *2* entiers positifs *p* et *q* sont jumeaux, s'ils sont tous les *2* premiers et si $q=p+2$ (ou $p=q+2$).
Par exemple, *5* et *7* sont jumeaux.

Exercice 5.9 :
Deux nombres entiers n et m sont qualifiés d'amis, si la somme des diviseurs de *n* est égale à *m* et la somme des diviseurs de *m* est égale à *n* (on ne compte pas comme diviseur le nombre lui même et *1*).
Exemple : les nombres *48* et *75* sont deux nombres amis puisque :
Les diviseurs de *48* sont : *2 + 3 + 4 + 6 + 8 + 12 + 16 + 24 = 75*
Les diviseurs de *75* sont : *3 + 5 + 15 + 25 = 48.*
Ecrire un algorithme qui permet de déterminer si deux entiers *n* et *m* sont amis ou non.

Exercice 5.10 :
Lorsque *x* est proche de *0*, *sin (x)* peut être approximé à l'aide de la formule suivante :

$$\sum_{i=0}^{n} \frac{(-1)^i * x^{2i+1}}{(2i+1)!}$$

Écrire une fonction qui retourne une approximation de *sin(x)*.
On s'arrête lorsque l'ajout est inférieur à une précision donnée (par exemple 10^{-6}) ou que l'on a atteint un certain nombre d'itérations *n*.

Exercice 5.11 : Nombre d'Armstrong
Ecrire un algorithme qui vérifie si un Entier positif est un nombre d'Armstrong.
Un nombre d'Armstrong *n* est un entier naturel pour lequel il existe un entier positif ou nul *p* tel que la somme de chacun des chiffres de *n* mis à la puissance *p* est égale à *n*.
Ecrire un algorithme qui vérifie si un Entier positif est un nombre d'Armstrong.
Exemple : $153 = 1^3 + 5^3 + 3^3$
$548834 = 5^6 + 4^6 + 8^6 + 8^6 + 3^6 + 4^6$

Exercice 5.12 :
Programmer une fonction qui insère un élément à sa place dans un tableau qu'on supposera déjà trié.
Principe : Ajouter la valeur de façon que le tableau reste trié, mais sans exécuter de tri.
Décaler les éléments en partant de la fin jusqu'à trouver une place correcte pour placer la nouvelle valeur ; il faut supposer que la place est suffisante c'est-à-dire que le nombre effectif d'éléments est strictement plus petit que la taille du tableau en mémoire.

Exercice 5.13 :
Écrire une fonction qui compte le nombre d'occurrences d'un caractère dans une chaîne de caractères.

Exercice 5.14 :
Ecrire une procédure qui permet de créer une copie d'une chaine de caractères.

Exercice 5.15 :
Ecrire une fonction qui retourne le minimum et sa position et le maximum et sa position dans un tableau d'entiers.
Indication : une fonction ne retourne qu'une seule valeur. Il faut passer les variables par référence.

Exercice 5.16 :
Ecrire une procédure qui permet de concaténer deux mots dans le premier.
Exemple : concaténer "bon" et "jour" donne "bonjour".

Exercice 5.17 :
Ecrire une fonction qui permet de comparer deux mots et qui retourne :
–1 si le premier mot est plus petit
0 si les deux mots sont égaux
1 sinon
Exemple : "*avant*" *et* "*plus*" → *-1*
"*pareil*" *et* "*pareil*" → *0*
"*avant*" *et* "*apres*" → *1*

Exercice 5.18 :
Ecrire un algorithme qui permet de supprimer les espaces supplémentaires (plus d'un espace) dans une chaîne de caractère.

Exercice 5.19 :
Ecrire un algorithme qui permet de purger un tableau (supprimer les éléments qui se répètent) d'entiers positifs de taille N sans utiliser un autre tableau.
Exemple : *t = {1, 5, 5, 10, 9, 1, 1, 30}*
devient *t = {1, 5, 10, 9, 30}*

Exercice 5.20 :
Ecrire une procédure qui permet de fusionner deux tableaux triés passés en paramètre dans un troisième tableau.
Mettre bout à bout deux tableaux triés de façon que le tableau résultant soit trié, et ce, sans exécuter de tris (coûteux).
Indication : Parcourir les 2 tableaux (avec 2 indices), et recopier dans le 3ème tableau la plus petite des 2 valeurs du 1er et du 2nd tableau ; puis avancer dans le tableau d'où on a recopié la valeur. A la fin de ce traitement, l'un des tableaux est copié, il suffit de copier les éléments restant de l'autre.

Exercice 5.21 :
Ecrire une procédure qui échange deux éléments d'un tableau d'entiers.

Exercice 5.22 : Tri de tableau
Écrire une procédure qui réalise le tri d'un tableau d'entiers par la méthode du :
1- tri par sélection.
Principe :
On recherche le plus petit élément *T[m]* de la suite *T[0] ... T[N–1]* et on l'échange avec *T[0]*,
On recherche le plus petit élément *T[m]* de la suite *T[1] ... T[N–1]* et on l'échange avec *T[1]*,

etc.
2- tri par insertion.
Principe :
On suppose les *i–1* premiers éléments triés et on insère le *i*-ème à sa place parmi les *i* premiers.
3- tri à bulles.
Principe :
On parcourt la suite *t[0] ... t[N–1]* en échangeant les éléments consécutifs *t[j–1] > t[j]*
On parcourt la suite *t[0] ... t[N–2]* en échangeant les éléments consécutifs *t[j–1] > t[j]*, etc.

Exercice 5.23 : Anagramme
Deux mots sont des anagrammes si l'un est une permutation des lettres de l'autre. Par exemple les mots suivants sont des anagrammes :
– aimer et maire
– chien et niche
– . . .
Par définition, on considère que deux mots vides sont des anagrammes.
Ecrire une fonction qui vérifie si deux mots sont des anagrammes.

Exercice 5.24 :
Soit *t* un tableau d'entiers non trié. Écrire une fonction qui retourne l'élément qui apparaît le plus souvent dans le tableau *t*, ainsi que son nombre d'occurrences. Si plusieurs éléments différents répondent au problème, votre algorithme doit en fournir un, quel qu'il soit. Vous ne devez utiliser aucun autre tableau que celui sur lequel vous travaillez.

Exercice 5.25 :
Ecrire une fonction qui calcule le produit scalaire de deux vecteurs.
Utiliser cette fonction pour écrire une procédure qui calcule le produit de deux matrices *A* et *B*

Exercice 5.26 :
Soit une matrice *T* de dimensions *N* x *N* x *2* d'entiers représentant l'existence de liaisons ferroviaires directs entre *N* villes et le prix de la liaison s'il existe.
Une ville est indiquée par un numéro (c'est un indice dans la matrice).
T[i,j,0] = 1 (ou *0*) indique l'existence (ou pas) d'une liaison directe entre les villes *i* et *j*
T[i,j,1] = 100 indique le prix de la liaison entre les villes *i* et *j* s'il existe (*T[i,j,1] = 1*).
Sur la diagonale est présente la même ville (*i = j*) et donc pas de liaison (cases vides).
1- Ecrire une fonction qui pour deux villes *k* et *l*, permet de savoir si les deux villes sont joignables directement en donnant le prix du trajet.
2- En cas d'absence de liaison directe, écrire une fonction qui cherche une ville qui permet de joindre indirectement ces deux villes (une seule escale) en donnant le prix du trajet.
3- Ecrire un algorithme qui simule une recherche de billet de train.

Exercice 5.27 :
1- Ecrire une fonction qui retourne la valeur d'un polynôme *P* de degré *n* en un point *x* donné.
Les coefficients (a_n, a_{n-1}, …., a_0) de *P* sont contenus dans le tableau *t* (*t[0]*=a_0, …, *t[n]*=a_n)
$p(x) = a_n*x^n + a_{n-1}*x^{n-1} + ... + a_1*x^1 + a_0$
2- Schéma de Horner
$p(x) = (((a_n*x + a_{n-1})*x + ...)*x + a_1)*x + a_0$

Exercice 5.28 :
En considérant un tableau de trois couleurs 'A', 'B', 'C' (ex. *Bleu, Blanc, Rouge)* réparties aléatoirement de dimension MAX, écrire une procédure qui le tri de telle façon que toutes les couleurs 'A' apparaissent au début du tableau, suivies des 'B' puis des 'C'.

Exercice 5.29 : Transposition de matrice
Écrire un algorithme qui réalise la transposition d'une matrice (en utilisant une seule matrice).

Exercice 5.30 :
Donnez une fonction qui retourne l'indice de la première occurrence de sous chaîne dans une chaîne. Si sous chaîne n'est pas présente dans la chaîne, alors la fonction retourne *-1*.

Exercice 6.1 :
Calculer le *n-ème* nombre de Fibonacci F_n qui est défini de la manière suivante :

$$\begin{cases} F_0 = F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \text{ pour } n > 1 \end{cases}$$

Exercice 6.2 :
Programmer l'exponentiation binaire de manière récursive. Cette fonction consiste à calculer x^n en appelant $(x*x)^{n/2}$ si *n* est pair, et $x*x^{n-1}$ si *n* est impair et retourne *1* si *n = 0*.

Exercice 6.3 :
Ecrire une fonction récursive qui affiche à l'écran des entiers positifs lus au clavier dans l'ordre inverse de leur saisi.
Indication : Afficher dans l'ordre inverse nécessite une mémorisation des valeurs saisies avant l'affichage. Le fait que la dernière valeur saisie est la première à afficher, l'avant dernière valeur saisie et la deuxième à afficher et ce jusqu'à la première valeur saisie qui sera afficher en dernier pousse à utiliser une pile (la pile de la récursivité).

Exercice 6.4 : Ackerman
Ecrire une fonction qui permet de calculer et retourner la valeur de la fonction d'Ackermann pour deux entiers positifs m et n donnés comme paramètres
La fonction d'Ackermann est définie comme suit :
Ackermann(0, n) = n + 1 *si* $n \geq 1$
Ackmann(m, 0) = Ackmann(m–1, 1) *si* $m \geq 1$
Ackmann(m, n) = Ackmann(m–1, Ackmann(m, n–1)) *si* $n \geq 1$ *et* $m \geq 1$.

Exercice 6.5 : Suites récurrentes de Mycielski
On considère les deux suites récurrentes de Mycielski définies par :

$$\begin{cases} m_1 = 2 \\ m_k = 2 * m_{k-1} + 1 \end{cases}$$

$$\begin{cases} c_1 = 1 \\ c_k = 3 * c_{k-1} + m_{k-1} \end{cases}$$

Ecrire un algorithme qui affiche le n-ème terme de la suite c_n.

Exercice 6.6 : PGCD
Programmer le *PGCD* de deux entiers en utilisant l'algorithme d'Euclide par récursivité.
Principe
si x > y alors *PGCD (x, y) = PGCD (y, x)*
si $1 \leq x \leq y$ *alors* *PGCD (x, y) = PGCD (x, y modulo x)*
si x = 0 alors *PGCD (x, y) = y*

Exercice 6.7 : PPCM

Le *PPCM* (plus petit commun multiplicateur) de deux entiers *a* et *b* est le plus petit entier *m* tel que *m* est divisible par *a* et *b*.
Exemple *PPCM (4, 3) = 12*.
Proposer un algorithme qui calcule le *PPCM* de deux entiers strictement positifs.
Indication : le PPCM est un multiple d'un des nombres qui doit être multiple de l'autre, ou en d'autres termes divisible par l'autre ; et si les multiples sont pris par ordre croissant, le premier trouvé sera le bon.

Exercice 6.8 :
Ecrire une procédure qui affiche un entier *x* (exprimé en base 10) en base *b* (*1<b<10*).
Exemple : $(42)_{10} = (101010)_2 = (52)_8$
Indication : La base b est une représentation des nombres qui utilise des chiffres de 0 à b–1 alors que la base 10 utilise de 0 à 9 (ex. base binaire *b=2* est représentée par 0 et 1). Pour transformer un nombre écrit en base 10 en un nombre écrit en base b, il suffit d'effectuer des divisions successives par b en prenant à chaque fois comme diviseur le quotient de la division précédente (sauf pour la première où le diviseur est le nombre donné) et ceci jusqu'à ce que ce diviseur soit égal à 0. Les restes successifs de ces divisions sont alors la représentation inverse en base b du nombre de départ.
Un affichage des restes au fur à mesure des divisions (solution 1) donnera un affichage à l'envers (du bit de poids faible vers le bit de poids fort) alors que pour afficher correctement le résultat, il faut mémoriser les restes jusqu'à la fin et par la suite les affichés à l'envers (c.à.d le dernier reste calculé est le premier affiché, etc.). Ainsi, la solution passe par l'utilisation d'une pile.

Exercice 6.9 :
Ecrire une fonction récursive qui affiche les éléments d'un tableau d'entiers du premier au dernier.
Indication : L'appel de la fonction récursive remplacera l'itération.

Exercice 6.10 : Nombres méchants
Les nombres méchants sont des nombres dont les facteurs premiers sont uniquement 2, 3 ou 5. Les 11 premiers nombres méchants sont : 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15.
Par convention, 1 est suppose un nombre méchant.
Ecrire un programme qui permet de trouver le centième nombre méchant.

Exercice 6.11 :
Ecrire une procédure qui transforme un entier en chaîne de caractères et une procédure qui réalise l'opération inverse
Exemple : *875 → "875"*

Exercice 6.12 :
Les égyptiens de l'antiquité savaient additionner deux entiers strictement positifs, soustraire 1 à un entier strictement positif, multiplier par *1* et *2* tout entier strictement positif et diviser par *2* un entier strictement positif pair.
Exemple de multiplication de *15* par *13* en n'utilisant que ces opérations :

$$\begin{aligned} 15 \times 13 &= 15 + 15 \times 12 \\ &= 15 + 30 \times 6 \\ &= 15 + 60 \times 3 \\ &= 75 + 60 \times 2 \\ &= 75 + 120 \times 1 \\ &= 75 + 120 \\ &= 195 \end{aligned}$$

Ecrire l'algorithme qui permet la multiplication de *2* entiers positifs suivant cette méthode (Donner une version itérative et une version récursive).

Exercice 6.13 :

Ecrire un algorithme qui permet de calculer le produit de *2* naturels en utilisant la Multiplication russe dont le principe est le suivant :

On divise par 2 autant de fois que possible un des deux nombres et en multiplie le $2^{ème}$ nombre par 2. Le produit est égal à la somme des multiples correspondant aux divisions impaires.

Exemple : 37 x 15

37	18	9	4	2	1
15	30	60	120	240	480

37 x 15 = 15 + 60 + 480 = 555

Exercice 6.14 :

On forme une suite de nombres de la façon suivante :

- le point de départ est un entier naturel donné, non nul, multiple de *3* ;
- chaque nombre a pour successeur la somme des cubes de ses chiffres ;
- théorème : toute suite de ce type devient constante, égale à *153* ;
- exemple : partons de *33*

$3^3 + 3^3 = 54$

$5^3 + 4^3 = 189$

$1^3 + 8^3 + 9^3 = 1242$

$1^3 + 2^3 + 4^3 + 2^3 = 81$

$8^3 + 1^3 = 513$

$5^3 + 1^3 + 3^3 = 153$

Ecrire un algorithme pour démontrer ce théorème.

Exercice 6.15 : Palindrome

Écrire un algorithme qui permet de savoir si une chaîne de caractères est un palindrome (un palindrome est un mot qui se lit aussi bien de la gauche vers la droite que de la droite vers la gauche comme par exemple les mots "été" ou "radar").

(Donner une version itérative et une version récursive).

Exercice 6.16 : Recherche dichotomique

Ecrire une fonction qui retourne la position d'un entier dans un tableau trié en utilisant la recherche dichotomique.

Principe :

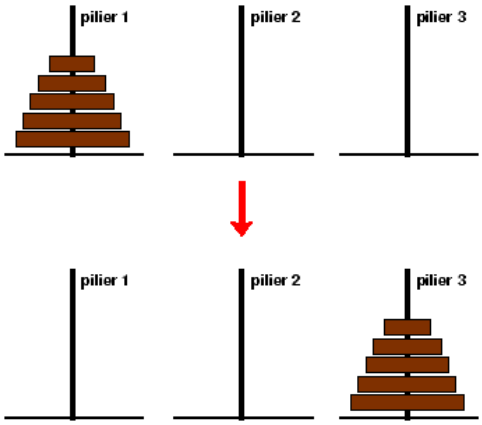
La recherche séquentielle fait apparaître un temps d'exécution en O(n) (proportionnel au nombre effectif de valeurs) ; comment améliorer l'algorithme ?

Le nombre d'éléments étant important, on décide de n'étudier que la moitié, on coupe en deux, on vérifie que la valeur recherchée est exactement au milieu, auquel cas on s'arrête car on a trouvé ; sinon on sait de quel côté (à gauche ou à droite) doit se trouver la valeur si elle existe ; et on recommence pour la partie considérée ; on définit la partie considérée par deux indices désignant la borne inférieure et la borne supérieure.

Exercice 6.17 : Tours de Hanoï

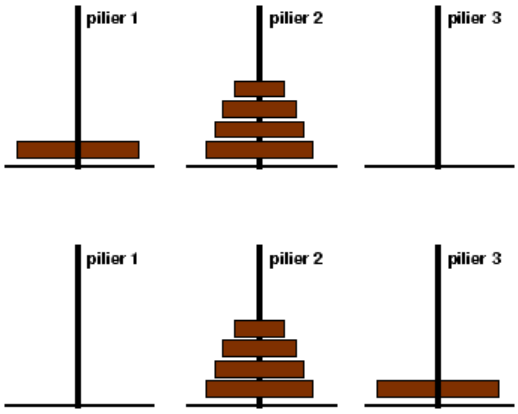
Les tours de Hanoï est un jeu solitaire dont l'objectif est de déplacer les disques qui se trouvent sur une tour (par exemple ici la première tour, celle la plus à gauche) vers une autre tour (par exemple la dernière, celle la plus à droite) en suivant les règles suivantes :

- on ne peut déplacer que le disque se trouvant au sommet d'une tour ;
- on ne peut déplacer qu'un seul disque à la fois ;
- un disque ne peut pas être posé sur un disque plus petit.



Méthode : Résoudre le problème des tours de Hanoï, c'est-à-dire déplacer *n* disques d'une tour source 'S' vers une tour destination 'D' en utilisant une tour intermédiaire 'I', revient à :

- déplacer *n – 1* disques de la tour source vers la tour intermédiaire;
- déplacer *1* disque de la tour source vers la tour destination;
- déplacer *n – 1* disques de la tour intermédiaire vers la tour destination.



Exercice 6.18 : Tri rapide (Quick sort)

Le principe est de placer une valeur du tableau à trier à sa bonne place (appelée pivot) puis recommencer la démarche sur les deux moitiés droite et gauche du pivot jusqu'à la fin du tri.

Exercice 6.19 : Tri par fusion

Écrire des procédures qui réalisent le tri d'un tableau de *n* entiers par la méthode du tri par fusion.

Principe :

Etant donné un tableau de taille *MAX*, on le partage en deux tableaux de taille (approximative) *MAX/2* que l'on trie récursivement et que l'on fusionne ensuite. Bien entendu les appels récursifs se terminent lorsque le tableau est de taille 1 auquel cas il est tout trié.

La procédure de tri par fusion d'un tableau est simple.

Méthode : Il nous faut utiliser une procédure de fusion de 2 sous-tableaux consécutifs d'un même tableau. Le 1er sous tableau commence de l'indice min à m et le 2ème de l'indice m + 1 à max

Exercice 6.20 :

Écrire un algorithme qui calcule le *zéro* d'une fonction *f(x)* sur l'intervalle [*a*; *b*], avec une précision *epsilon*. La fonction f et les réels *a*, *b* et *epsilon* sont donnés. Soit *f(x)* une fonction continue sur l'intervalle [*a*; *b*], où elle ne s'annule qu'une seule et unique fois. Pour trouver ce *zéro*, on procède par dichotomie, c'est-à-dire que l'on divise l'intervalle de recherche par *deux* à chaque étape. Soit *m* le milieu de [*a*; *b*]. Si *f(m)* et *f(a)* sont de même signe, le *zéro* recherché est dans l'intervalle [*m*; *b*], sinon il est dans l'intervalle [*a*; *m*].

Exercice 6.21 :

Calculer le maximum d'un tableau de manière récursive.

Exercice 6.22 : Fractal : courbes de Koch

La courbe de Koch est un fractal dont le détail varie en fonction d'un niveau donné *n*.

La figure ci-dessous à droite présente la courbe de Koch de niveau *2*. En fait, dessiner la courbe de Koch de niveau *n* entre deux points $A(x_a, y_a)$ et $E(x_e, y_e)$ revient à :

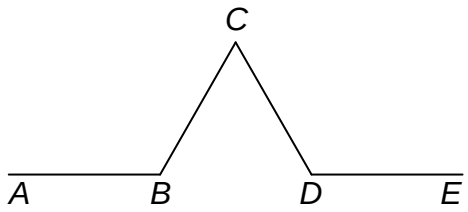
– dessiner le segment de droite [*A*, *E*] si *n = 0*

– dessiner *4* segments de Koch de niveau *n = 1* entre les points *A*, *B*, *C*, *D* et *E* ([*A*, *B*], [*B*, *C*], [*C*, *D*] et [*D*, *E*]) telles que :

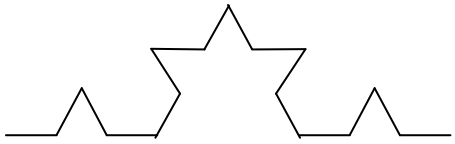
– *AB = AE/3*

– *ED = EA/3*

– Les points *B*, *C* et *D* forment un triangle équilatéral.



n = 1



n = 2

Exercice 6.23

On considère la fonction récursive f sur l'ensemble des entiers positifs donnée par :

f(1) = 0

f(2) = 1

f(n) = f(n-f(n-1)) + f(n-1 - f(n-2));

1. calculer les valeurs de *f(n)* pour la valeur *n = 3*.
2. Ecrire une version itérative du calcul de *f(n)* qui pour obtenir ce résultat.

SERIE DE DEVOIRS SURVEILLES EN INFORMATIQUE

Proposée par Pr. D. El Ghanami

Ecole Mohammadia d'Ingénieurs

Mai 2011

Achat de marchandise dans un magasin

- ✓ On attachera une importance à la concision, à la clarté, et à la précision de la rédaction en langage C.
- ✓ La fonction prédéfinie autorisée est *printf*.
- ✓ La liste ne doit pas être transformée en tableau pour traiter les questions de la partie 2.
- ✓ *a* et *b* deux entiers positifs, *a/b* donne le quotient de la division et *a%b* donne son reste.

***

On veut aider un client dans l'achat de produits dans un magasin. Ce magasin possède un catalogue de ***N*** produits différents qu'il vend représenté par un tableau ***T*** de produits.
T est déclaré dans la fonction ***main()***. ***N*** est une constante globale.

Chaque produit a un numéro ***no*** et un prix de vente ***prix*** en dirhams (des entiers positifs). Chaque produit a un prix unique dans le catalogue et disponible en nombre d'exemplaires suffisant dans le magasin.

Le type produit proposé est :

```
struct   produit {
           int  no
           int  prix ;
} ;
```

avec la déclaration de synonyme : *typedef struct produit prod ;*

Exemple de catalogue *T* de *10* produits différents de numéros : *1* à *10* et des prix : *720dh, …. , 480dh*

1	*2*	*3*	*4*	*5*	…	*10*
720	*204*	*600*	*120*	*500*	…	*480*

Un client se présente dans le magasin avec un montant d'argent ***m*** en dirhams et un camion de capacité de stockage ***c*** pour acheter et transporter des produits.
m et ***c*** des entiers positifs déclarés dans la fonction *main()*.

Chaque produit est mis dans un carton pour le transporter. Tous les cartons ont le même volume ***v*** (entier constant). Un carton ne peut contenir qu'un seul exemplaire de produit.

Pour rentabiliser son déplacement chez le magasin, le client décide de dépenser le maximum d'argent sans oublier la capacité du camion.

Exemple : l'achat d'un produit de *20dh* est mieux que deux produits de *10dh*. L'achat de *3* produits *n°3* et de *4* produits *n°5* coûtera *3800dh* et occupera un espace camion de *7v*.

Le client simule des calculs pour décider quels produits acheter et en combien d'exemplaires.
On veut l'aider dans ses calculs en lui proposant des fonctions développées selon sa logique.

Partie 1

Sachant que le volume d'un carton est ***v*** et le prix du produit le moins cher est ***pm*** et sans programmer, que pouvez-vous dire :

1. Du montant d'argent ***m*** qui restera à la fin d'achat si la capacité du camion ***c*** est illimitée ?
2. De l'espace *c* qui restera dans le camion à la fin d'achat si ***m*** est très grand ?

Au début, le client cherche à acheter le maximum de produits avec son argent.

3. Ecrire la fonction ***prodSelect(T)*** qui retourne *px* le produit qui peut être acheté le plus.
4. Ecrire la fonction ***nbProd(m, c, px)*** qui donne le nombre d'exemplaires achetés de ce produit.

Cependant, ce calcul remplit le camion mais le client lui restera un montant d'argent important non dépensé à cause de la capacité de stockage *c* qui est limitée.

De la même manière, il calcule le minimum de produits qu'il peut acheter.

Ainsi, il constate qu'il lui reste encore de l'espace dans le camion et moins d'argent que précédemment.
Par conséquent, il décide d'adopter ce calcul en continuant l'achat, à chaque fois, du produit le plus cher possible par l'argent qui reste à chaque fois.

Pour faciliter la sélection des produits, il les classe par ordre décroissant de prix et cela selon l'approche suivante : il cherche le produit le plus cher et l'insert à la 1ère place, puis reprend cette démarche pour les autres produits jusqu'à obtenir un catalogue trié selon les prix.

Exemple de prix de produits :

1	*2*	*3*	*4*	*5*	*6*	*7*
120	*204*	*600*	*123*	*500*	*456*	*480*

Quelques étapes de tri :

3	*1*	*2*	*4*	*5*	*6*	*7*
600	*120*	*204*	*123*	*500*	*456*	*480*

3	*5*	*1*	*2*	*4*	*6*	*7*
600	*500*	*120*	*204*	*123*	*456*	*480*

3	*5*	*7*	*1*	*2*	*4*	*6*
600	*500*	*480*	*120*	*204*	*123*	*456*

3	*5*	*7*	*6*	*1*	*2*	*4*
600	*500*	*480*	*456*	*120*	*204*	*123*

3	*5*	*7*	*6*	*2*	*1*	*4*
600	*500*	*480*	*456*	*204*	*120*	*123*

5. Ecrire la fonction ***classCat(T)*** qui classe les produits du catalogue selon le principe ci-dessus.

Après tri, le client commence l'achat et le stockage de produits dans le camion. Lors de cet achat, le client sent qu'il répète, après chaque sélection de produit, la même opération comme s'il vient d'arriver dans le magasin avec un nouveau montant d'argent ***m*** inférieur et une nouvelle capacité du camion *c* inférieure.

6. Ecrire la fonction récursive ***restArgent(T, m, c, …)*** qui donne l'argent qui reste à la fin d'achat.

N.B. Les … en paramètre de la fonction signifient qu'il peut y avoir d'autres paramètres à définir.

7. Cette fonction peut-il être écrite, par simple transformation, en itérative ? (expliquer).
8. Ecrire la fonction itérative ***espLibre(T, m, c)*** qui donne l'espace restant dans le camion à la fin d'achat.

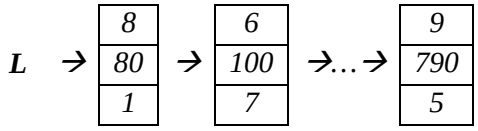
Partie 2

Lors d'achat de produits, le client enregistre les informations relatives aux différents produits achetés dans une liste chaînée ***L*** (le produit ***pr*** et le nombre d'exemplaires achetés de chaque produit ***ne***). Un produit sélectionné ne doit figurer qu'une seule fois dans la liste.

Le type liste proposé est :

```
typedef struct tliste{
        prod pr ;
        int ne ;
        struct tliste *s ;
} liste ;
```

Exemple de liste :



On suppose que cette liste ***L*** est déclaré dans la fonction *main()* par l'instruction :

```
liste *L = NULL ;
```

L'opération de sélection des produits du catalogue suit le principe cité en partie1 et l'enregistrement dans la liste *L* consiste à empiler les informations des produits sélectionnés dans la liste selon le principe le dernier produit acheté est le premier de la liste.

9. Ecrire la fonction ***listeProd(L, T, m, c)*** qui donne la liste des produits achetés et leur nombre.
10. Ecrire la fonction ***totalDep(L)*** qui donne le montant global dépensé lors de l'achat de produits.
11. Ecrire la fonction récursive ***affiche(L)*** qui affiche les prix des produits du plus au moins cher.

***

Des points dans le plan

- ✓ On attachera une importance à la concision, à la clarté, et à la précision de la rédaction en langage C.
- ✓ Il est conseillé de lire la totalité de l'énoncé avant de commencer.
- ✓ La fonction autorisée est ***sqrt(x)*** qui retourne la racine carrée du réel *x*.
- ✓ Une fonction développée dans une question peut être utilisée même si elle est incomplète.

***

Dans ce problème, on considère deux tableaux de réels *Tx* et *Ty* de taille *N* représentant les positions de *N* points différents dans le plan. *Tx* contient les abscisses *x* et *Ty* les ordonnées *y* des *N* points.
Tx et *Ty* sont des variables globales et *N* est une constante globale.

x et *y* sont des réels. Ainsi, le plan est divisé en deux demi plan : demi plan sup ($y \geq 0$) et demi plan inf ($y < 0$).

Exemple de *10* points :

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Tx	*-0,7*	*-0,1*	*-0,5*	*0,5*	*1*	*1*	*-0,7*	*-1*	*0*	*0*
Ty	*1,2*	*-1*	*-1,3*	*1,3*	*-1*	*1*	*0,6*	*1*	*-1,4*	*0*

Le 1er point *(i = 0)* de coordonnée *x = -0,7* et *y = 1,2* se trouve dans le demi plan sup.
Le 2ème point de coordonnée *x = -0,1* et *y = -1* se trouve dans le demi plan inf.

I- On cherche à calculer la distance entre deux points quelconques du plan.

1. Ecrire la fonction *float car(float x)* qui donne le carré du réel *x*.
2. Ecrire la fonction *float dist1(int i, int j)* qui donne *dij* la distance entre les points d'indices *i* et *j*.
3. Ecrire la fonction *float dist2()* qui donne *dmax* la distance entre les *2* points les plus éloignés l'un de l'autre.
4. Ecrire la fonction *float dist3(int i)* qui donne *di* la distance entre le point *i* et l'origine *(0,0)*.

On veut calculer *rmax* le rayon du cercle, autour de l'origine, formé par les points du plan les plus loin de l'origine et le nombre *K* de points appartenant à ce cercle. *K* et *rmax* sont déclarés globaux.

5. Ecrire la fonction *void cercleMax()* qui calcule *rmax* et *K* (donner une solution en un seul parcours de *Tx* et *Ty*).

II- On suppose que les coordonnées des *K* points formant ce cercle sont stockés dans une matrice de réels *M* de taille *2*x*K* déclarée globale. La 1ère ligne contient les abscisses *x* et la 2ème ligne contient les ordonnées *y*.

De l'exemple précédent, *M* est :

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*
x	*-0,7*	*-0,5*	*0,5*	*1*	*1*	*-1*	*0*
y	*1,2*	*-1,3*	*1,3*	*-1*	*1*	*1*	*-1,4*

On veut trier les points de la matrice *M* dans le sens de rotation des aiguilles de la montre en commençant par le demi plan sup.
Pour cela, on propose d'abord le rangement des éléments de *M* de sorte à avoir 2 zones dans *M*, les points du demi plan sup au début de *M* suivi des points du demi plan inf.

Le rangement ne doit pas se baser sur le tri des points, qui nécessite plus d'opérations inutile, mais sur la permutation, deux à deux, de points qui nécessitent de changer de zone (voir exemple ci-dessous).
Les étapes de rangement de *M* :

i	*0*	***1***	*2*	*3*	*4*	***5***	*6*
x	*-0,7*	*-0,5*	*0,5*	*1*	*1*	*-1*	*0*
y	*1,2*	*-1,3*	*1,3*	*-1*	*1*	*1*	*-1,4*

i	*0*	*1*	*2*	***3***	***4***	*5*	*6*
x	*-0,7*	***-1***	*0,5*	*1*	*1*	***-0,5***	*0*
y	*1,2*	***1***	*1,3*	*-1*	*1*	***-1,3***	*-1,4*

i	*0*	*1*	*2*	*4*	*3*	*5*	*6*
x	*-0,7*	*-1*	*0,5*	***1***	***1***	*-0,5*	*0*
y	*1,2*	*1*	*1,3*	***1***	***-1***	*-1,3*	*-1,4*

6. Ecrire la fonction *void permute(int i, int j)* qui permute deux points de *M* d'indices *i* et *j*.
7. Ecrire la fonction *void range()* qui permet de réaliser le rangement de *M* selon le principe précédent.
8. Ecrire la fonction *int indic()* qui donne *inf* indice du 1er point du demi plan inf après rangement de *M*.

Le tri de *M* donne :

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*
x	*-1*	*-0,7*	*0,5*	*1*	*1*	*0*	*-0,5*
y	*1*	*1,2*	*1,3*	*1*	*-1*	*-1,4*	*-1,3*

9. Ecrire la fonction *void triCercle()* qui trie la matrice *M* dans l'ordre cité ci-dessus.

III- Après avoir trié la matrice *M*, on cherche à vérifier si un point *P* de coordonnées *x1* et *y1* appartient à ce cercle. Le fait que la matrice soit triée accélère la recherche qui se fait en deux étapes :

- La désignation d'abord dans *M* de la zone correspondant au demi plan où la recherche sera effectuée.

- Dans la zone du demi plan concerné par la recherche, on compare *x1* et *y1* avec *x* et *y* du point se trouvant à l'indice milieu *im* de cette zone, si ils sont égaux alors *P* appartient au cercle sinon on recommence la recherche dans la zone à gauche ou à droite de *im* selon le résultat de la comparaison cela jusqu'à la fin de la recherche.

En supposant que la recherche se fait dans le demi plan sup

10. Ecrire la fonction récursive *int rechSup(float x1, float y1, int ideb, int ifin)* qui recherche si le point *P* appartient à ce demi plan ou pas et cela en respectant le principe de recherche cité ci-dessus.

En supposant que la recherche se fait dans le demi plan inf

11. Ecrire la fonction itérative *int rechInf(float x1, float y1, int ideb, int ifin)* qui recherche si le point *P* appartient à ce demi plan ou pas et cela en respectant le principe de recherche cité ci-dessus.

12. Ecrire la fonction *recherche(float x1, float y1)* qui vérifie si le point *P* appartient à *M* ou pas.

***

Compression et Brouillage d'images

On veut construire un outil de compression et de brouillage d'images (appelé encodeur) ainsi que l'outil capable de les décoder et de les décompresser (appelé décodeur).

I- Représentation d'une image par une matrice de caractères.

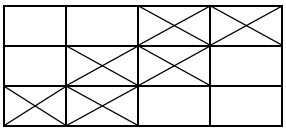
Pour simplifier le problème, nous considérons des images en noir et blanc. Une image est décrite ligne par ligne par des points (pixels) représentés par des caractères (caractère *'b'* pour les points blancs et caractère *'n'* pour les points noirs). Ces lignes ont le même nombre de points.

Exemple : La matrice de caractères *{"nbnn","bnbb","bbbb"}* représente l'image *3*x*4* suivante :

'n'	*'b'*	*'n'*	*'n'*	*'\0'*
'b'	*'n'*	*'b'*	*'b'*	*'\0'*
'b'	*'b'*	*'b'*	*'b'*	*'\0'*

Le caractère *'\0'* ne fait pas partie de l'image mais indique la fin d'une chaîne de caractères (fin d'une ligne).

1- Donnez la matrice de caractères correspondant à l'image suivante :



2- Dessinez l'image correspondant à la matrice suivante : *{"bnb","nnn","nbn"}*

II- Représentation d'une image par une séquence d'entiers.

L'objectif de la compression d'image est de représenter une image par une séquence d'entiers plus courte que la matrice de caractères *'n'* et *'b'*.

Pour compresser une image, on suit les étapes suivantes :

- On forme d'abord une matrice d'entiers à partir de l'image où à chaque ligne de l'image correspond une ligne de la matrice d'entiers délimitée par l'entier ***–1***.

La 1ère valeur d'une ligne indique le nombre de points blancs du début de la ligne (le nombre vaut *0* si la ligne ne commence pas par des points blancs) ;
La 2nd valeur indique le nombre de points noirs qui suivent, la 3ème valeur indique le nombre de points blancs qui suivent, et ainsi de suite jusqu'à la fin de la ligne.

- Puis, on transforme la matrice d'entiers en séquence d'entiers où ***–1*** indique la fin de chaque ligne et la suite ***–1 –1*** indique la fin de l'image (fin de la séquence d'entiers).

Les valeurs de la séquence sont stockées dans un tableau.

De l'exemple précédent, on forme la matrice d'entiers suivante :

0	*1*	*1*	*2*	*-1*
1	*1*	*2*	*-1*	
4	*-1*			

Ainsi, la séquence est :

0	*1*	*1*	*2*	*-1*	*1*	*1*	*2*	*-1*	*4*	*-1*	*-1*

3- Donnez la matrice de caractères de l'image correspondant à la séquence d'entiers :

{0 , 2 , 1 , -1 , 2 , 2 , 2 , -1 , 0 , 1 , 2, -1 , -1}

4- Donnez la séquence d'entiers correspondant à l'image *{"bnb","nnn","nbn"}*

5- Définir les constantes globales suivantes : *NL* (nombre de lignes de l'image), *NC* (nombre de colonnes de l'image) et *MAX* (taille maximale du tableau d'entiers contenant la séquence).

6- Pourquoi *MAX* doit être égal à *NL*(NC+2)* ?

7- Ecrire la fonction *void afficheSeq(int seq[])* qui affiche les entiers de la séquence séparés par des virgules jusqu'au marqueur de fin de séquence.

III- Compression d'images

8- Ecrire la fonction *void compresseImage*(*char image[][NC+1], int seq[])* qui permet de compresser une image de taille *NL*x*NC*.

Voici quelques fonctions à développer qui peuvent vous aider.

9- La fonction *void compresseL(char limage[], int lmat[])* qui permet de compresser un tableau de caractères.

10- La fonctionre *compresserMat (char image[][NC+1], int matEnt[][NC+2])* qui permet de compresser une image dans une matrice d'entiers.

11- La fonction *void compresseSeq(int matEnt[][NC+2], int seq[])* qui forme la séquence d'entiers compressée à partir d'une matrice d'entiers.

IV- Brouillage et décodage d'une image

Le principe du brouillage D.E.S. consiste à faire une rotation de 90°, dans le sens des aiguilles de montre, d'un carré de quatre points comme indiqué ci-dessous :

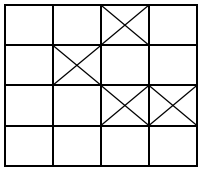
1	*2*
4	*3*

Avant rotation

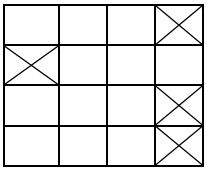
4	*1*
3	*2*

Après rotation

Pour brouiller une image on applique cette permutation à chacun des carrés de quatre points de l'image comme indiqué sur l'exemple suivant :



Avant rotation



Après rotation

On suppose que le nombre de lignes et de colonnes d'une image est toujours pair.

12- Ecrire la fonction *void brouillerImage* (*char image[][NC+1])* qui permet de brouiller une image suivant le sens indiqué précédemment.

Voici une procédure, à développer, qui peut vous aider.

13- La fonction *void brouillerCarre (char image[][NC+1], int l, int c)* permet de brouiller un carré d'image d'indice ***l*** et ***c***.

14- Ecrire une procédure qui permet de brouiller et de compresser une image noir et blanc.

V- Décodage et décompression d'une image

15- Proposer la fonction *void decompresserImage (int seq[], char image[][NC+1])* qui réalise le décodage et la décompression d'une image brouillée et compressée. Son prototype est :

Indication. En brouillant une image déjà brouillée, obtient-on une image encore brouillée ?

***

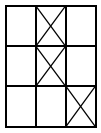
Cryptage d'image

On cherche à crypter une image numérique en couleur de taille *100*x*100* pixels. Un pixel (ou point) est une couleur et une intensité lumineuse.

Dans ce problème, on ne considérera que les couleurs des pixels. Une couleur est un entier codé sur *8* bits, soit *256* couleurs différentes **représentées par des entiers compris entre *0* et *255***.

Ainsi, les couleurs seront numérotées de *0* à *255* avec *0* représentant le blanc et *255* représentant le noir.

Exemple : L'image de la figure*1* de *3*x*3* pixels est représentée par la matrice *M* de taille *3*x*3* (figure*2*)



0	255	0
0	255	0
0	0	255

0	255	0	0	255	0	0	0	255

Figure1 Figure2 Figure3

Soit une image de ***N***x***N*** pixels (***N = 100***) et une fonction ***remplirImage(M)*** qui permet, en l'appelant, de remplir une matrice ***M*** d'entiers de taille ***N***x***N*** à partir cette image.

Pour crypter cette image, on utilise le code de César. Le principe est de modifier les couleurs de l'image par rapport à une clé qui est aussi une couleur.

La clé ***K*** (représentée par un entier entre ***0*** et ***255***) donne le décalage ***d*** entre la couleur de la clé ***K*** et le blanc de numéro ***0***.

Par exemple, en décalant les couleurs de ***1*** position, le blanc devient la couleur numéro ***1***, la couleur numéro ***0 < i < 255*** devient la couleur numéro ***i+1***, le noir devient le blanc.

1. Que donne le codage de l'image précédente en utilisant un décalage de ***10*** ?

2. Ecrire un programme (fonction main) qui permet de :

- ***2.1.*** Déclarer et initialiser une matrice ***M*** à partir d'une image non cryptée.
- ***2.2.*** Lire et contrôler un nombre entier ***K*** positif représentant la clé (***0 ≤ K < 256***)
- ***2.3.*** Crypter l'image ***M***.

Pour réaliser le décryptage, il faut connaître la valeur du décalage ***d***. Une manière de la déterminer est d'essayer de deviner cette valeur. L'approche proposée est de chercher la couleur dominante dans l'image cryptée (on suppose qu'elle est unique). En effet, la couleur dominante dans une image non cryptée est la couleur ***X*** de numéro *100*.

3. Ecrire un programme (main) qui permet de :

- ***3.1.*** Déclarer et initialiser une matrice *M* à partir d'une image cryptée.
- ***3.2.*** Chercher la couleur dominante dans cette image.
- ***3.3.*** Déduire la clé du cryptage pour que la couleur *X* soit la plus fréquente dans l'image décrypté.
- ***3.4.*** Décrypter l'image *M*.

***

Etude d'une carte géographique

- ✓ On attachera une importance à la concision, à la clarté, et à la précision de la rédaction en langage C.
- ✓ Aucune fonction prédéfinie n'est autorisée.

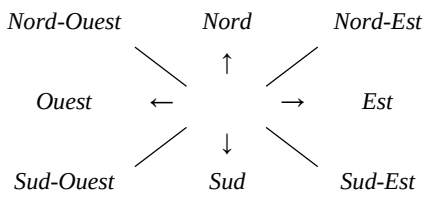
***

Soit une carte géographique d'une région sous forme d'une grille *T* (matrice de taille *N*x*M*) représentant les coordonnées de *N*x*M* points géographiques (abscisses, ordonnées et altitudes).

La carte *T* est orientée (Nord, Sud, Est et Ouest) et *N* et *M* sont des constantes globales.

Exemple : *T* de taille *6*x*9* :

i\j	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
0	*6*	*4*	*4*	*4*	*4*	*2*	*1*	*0*	*2*
1	*5*	*4*	*3*	*2*	*1*	*5*	*5*	*4*	*4*
2	*5*	*5*	*5*	*1*	*0*	*1*	*5*	*6*	*6*
3	*4*	*6*	*7*	*6*	*7*	*6*	*7*	*9*	*8*
4	*2*	*3*	*8*	*10*	*8*	*8*	*6*	*6*	*4*
5	*1*	*4*	*5*	*8*	*8*	*8*	*4*	*4*	*5*



Les points de *T* sont identifiés par leur position *h* dans *T* (*1* ≤ *h* ≤ *N*x*M*). *h* est ordonné du Nord-Ouest *(i=0, j=0)* vers le Sud-Est *(i=N–1, j=M–1)* en passant ligne par ligne.

La relation entre *h* et les indices ligne *i* et colonne *j* de *T* est *h=i* x *M+j+1*.
Ainsi, *i=(h–1)/M* et *j=(h–1)%M*.

Le type de *T* est :

```
typedef struct {
        int a ;
        char c ;
} elt ;
```

a est un entier positif représentant l'altitude d'un point de *T*.
c sera utilisé pour marquer un point de *T*. Sa valeur par défaut est *' '*. (espace)

Un point de *T* est de coordonnées *(i, j, T[i][j].a)*, *i* est son abscisse, *j* son ordonnée et *T[i][j].a* son altitude.

Exemple : les points *1*, *9* et *10* ont respectivement les coordonnées *(0, 0, 6)*, *(0, 8, 2)* et *(1, 0, 5)*.

Les points de la matrice *T* sont de deux types :

- des points bords correspondant aux indices *i = 0*, *i = N–1*, *j=0 ou j = M–1*.
- des points centres correspondant aux indices *1* ≤ *i* < *N–1* et *1* ≤ *j* < *M–1*.

Un point centre ne peut pas être un point bord et inversement.

Dans ce problème, on veut étudier les descentes de la carte *T* à partir de son sommet le plus haut. Pour cela, on développera des fonctions permettant de tracer les descentes et étudier une d'elle.

Définition : Une descente de *T* est le passage d'un point d'altitude *a1* à un de ses points voisins d'altitude *a2* tel que *a1 > a2*. La descente continue jusqu'à ne plus pouvoir descendre.

Partie 1 : Tracé de descentes

I- On cherche d'abord à tracer une des descentes de *T*. Le tracer consiste à marquer la variable *c* des points qui composent la descente par la lettre *'d'* (*T[i][j].c = 'd'*).
Pour cela, il faut marquer le sommet le plus haut de *T* puis marquer à chaque fois un point descente tant qu'il existe ou atteindre un point bord de *T*.

1. Ecrire la fonction *f(i, j)* qui retourne *h* la position d'un point de *T* à partir de ses indices *i* et *j*.
2. Ecrire la fonction *abs(x)* qui retourne la valeur absolue de l'entier *x*.
3. Ecrire la fonction *bord(i, j)* qui vérifie si le point d'indices *i* et *j* est un point bord de *T* ou pas.

Un point centre est un sommet si $T[i][j].a > T[k][l].a$ pour $i-1 \leq k \leq i+1$ et $j-1 \leq l \leq j+1$ avec $k \neq i$ et $l \neq j$.

Exemple : Dans *T*, le point de coordonnée *(3, 7, 9)* est un sommet.

4. Ecrire la fonction *sommet(T, i, j)* qui vérifie si le point centre d'indices *i* et *j* est un sommet ou pas.

Un point descente est un point voisin d'un point centre d'indices *i* et *j* permettant une descente. Dans le cas de plusieurs points descentes alors on retourne le 1er point, dans l'ordre de *h* présenté précédemment.

Exemple : Dans *T*, la 1ère descente à partir du point de coordonnées *(2, 7, 6)* est le point *(1, 6, 3)*.

5. Ecrire la fonction *point(T, i, j)* qui retourne *h* la position du 1er point descente s'il existe sinon *0*.

II- On trace ici la descente à partir du sommet le plus haut en passant à chaque fois par le 1er point descente.

Exemple : La figure suivante présente les tracés de la 1ère descente de *T* à partir du point *(4, 3, 10)* et celle à partir du point *(3, 7, 8)*.

i\j	0	1	2	3	4	5	6	7	8
0	6	4	4	4	4	2	1 'd'	0	2
1	5	4 'd'	3 'd'	2 'd'	1 'd'	5	5	4 'd'	4
2	5	5 'd'	5	1	0 'd'	1	5 'd'	6	6
3	4	6	7 'd'	6	7	6	7	9 'd'	8
4	2	3	8	10 'd'	8	8	6	6	4
5	1	4	5	8	8	8	4	4	5

6. Ecrire la fonction *sommetMax(T)* qui retourne *h* la position du sommet le plus haut de *T* s'il existe.
7. Ecrire la fonction *descente(T, i, j)* qui marque le point sommet le plus haut d'indices *i* et *j* et tous les points de sa 1ère descente par la lettre *'d'*.

Sachant que le tracé d'une descente composée de *k* points est le marquage du 1er point et le tracé d'une descente de $k - 1$ points.

8. Ecrire une version récursive de la fonction *descente(T, i, j)*.

On veut tracer toutes les descentes à partir du sommet le plus haut de la carte. Le tracé consiste à marquer le point sommet et toutes ses descentes.
Remarque importante : Chaque point de ce tracé peut être vu avec les descentes qui le traversent comme un tracé de descentes.

9. Ecrire la fonction récursive *lesDescentes(T, i, j)* qui marque le point d'indices *i* et *j* et toutes ses descentes par la lettre *'d'*.

Partie 2 : Etude d'une descente

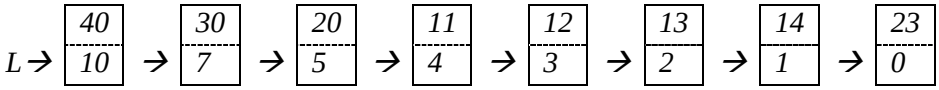
Les points d'une descente partant du sommet le plus haut de *T* forment la liste chaînée *L*. Le point sommet de *T* se trouve au début de la liste *L*.

Le type de *L* est :

```
typedef struct tliste{
        int h, a ;
        struct tliste *s ;
} liste ;
```

h est la position du point dans *T* et *a* son altitude.

Exemple de *L* :



III- Dans cette partie, on veut étudier les caractéristiques de la descente *L* afin de l'exploiter pour des activités sportives (ex. ski).

Quelques caractéristiques de *T* : La distance entre le point *30* et le point *20* est *2*. La longueur de *L* est 10.

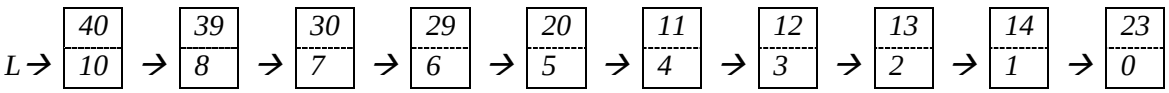
10. Sachant que la distance entre *2* points voisins de *L* est la différence entre leur altitude, écrire la fonction *dist2(p)* qui retourne la distance *d* entre le point **p* et son suivant s'il existe sinon *0*.
11. Ecrire la fonction *longueur(L)* qui retourne la longueur de la descente *L*.
12. Ecrire la fonction *moyen(L)* qui retourne la moyenne des distances *d* entre les points successifs de *L*.

Sachant que la longueur d'une descente à partir du point *p* est égale à la somme de *dist2(p)* et la longueur de la descente à partir de son suivant.

13. Ecrire une version récursive de la fonction *longueur*.

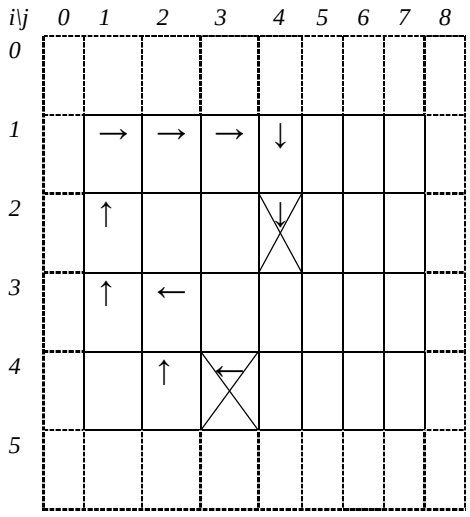
IV- On désire allonger la descente *L* tant que c'est possible afin d'affaiblir sa force de descente. Pour cela, on cherche à augmenter le nombre de points de *L* tout en gardant sa longueur. Pour cela, on adopte la démarche suivante :
Soit le point **p* de *L* et son suivant (le point suivant est la descente de **p*), alors on cherche un point *h* de *T* intermédiaire entre eux de sorte qu'il maintient la descente. *h* peut être un point quelconque (bord ou centre). S'il y'a plusieurs points intermédiaires *h*, alors on prend le 1er point dans l'ordre de *h*.

Exemple : *L* ajustée:



La descente représentée dans *T* (figure à gauche). La figure à droite représente le sens de la descente.

i\j	0	1	2	3	4	5	6	7	8
0	6	4	4	4	4	2	1	0	2
1	5	4 'd'	3 'd'	2 'd'	1 'd'	5	5	4	4
2	5	5 'd'	5	1	0 'd'	1	5	6	6
3	4	6 'd'	7 'd'	6	7	6	7	9	8
4	2	3	8 'd'	10 'd'	8	8	6	6	4
5	1	4	5	8	8	8	4	4	5

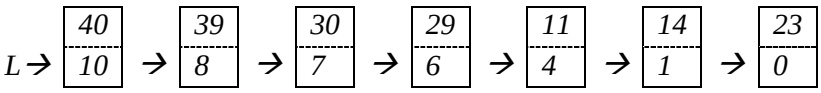


14. Ecrire la fonction *interim(T, p)* qui retourne le point intermédiaire *h* s'il existe sinon *0*.
15. Ecrire la fonction *ajuster(T, L)* qui ajuste tant que c'est possible le tracer de la descente *L*.

III- Les directions de la carte *T* sont Est/Ouest, Nord/Sud, Nord-Ouest/Sud-Est et Nord-Est/Sud-Ouest (voir l'orientation de la carte *T*).

Dans cette partie, on veut factoriser la descente *L*. La factorisation consiste à garder dans *L* que les points présentant un changement de direction. Pour cela, on supprime de chaque séquence de points continue de *L* se trouvant dans la même direction tous les points milieu et on ne garde que les deux points d'extrémités.

Exemple : *L* précédente factorisée



16. Ecrire la fonction *factor(L)* qui factorise la liste *L*.

***

Organisation d'élections

- ✓ On attachera une importance à la concision, à la clarté, et à la précision de la rédaction en langage C.
- ✓ La fonction prédéfinie autorisée est *printf*.

***

Dans un pays, *N* parties politiques ont présenté leur liste de candidats aux élections (listes électorales numérotées de *1* à *N*). Ces élections ont vu la participation de *M* votants, chacun inscrit un numéro de liste dans un bulletin de vote saisi dans un tableau de votes *V* de taille *M*. *V[i] = j* signifie que le vote *i* ($0 \leq i < M$) est partie pour la liste *j* ($1 \leq j \leq N$). Un vote blanc ou annulé n'est pas considéré vote valide et *V[i]=0*.

N et *M* sont des constantes globales.

Exemple de *25* votants sur *6* listes :

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*
V	*3*	*3*	*0*	*2*	*5*	*2*	*3*	*0*	*2*	*1*	*3*	*5*	*5*	*0*	*2*	*3*	*4*	*0*	*0*	*6*	*5*	*5*	*5*	*6*	*6*

Les résultats des listes sont stockées dans un tableau *P* de taille *N+1* où *P[i]=j* signifie que le nombre de votes valides obtenus par la liste *i* est *j*.
P[0] contient le total des votes blancs ou annulés.

Exemple : Les résultats de l'exemple précédent :

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*
P	*5*	*1*	*4*	*5*	*1*	*6*	*3*

Le problème consiste à aider la commission d'organisation des élections à appliquer les règles électorales pour décompter les résultats, nommer le gagnant majoritaire ou prévoir la coalition de listes idéale.

I- Une liste obtenant plus de *50%* des voix valides est considérée majoritaire et désignée vainqueur des élections. De l'exemple précédent, la liste qui obtient au moins *11* voix valides gagnera les élections.

1. Ecrire la fonction *init(T, L)* qui initialise le tableau d'entiers *T* de taille *L* à *0*.
2. Ecrire la fonction *scores1(V, P)* qui remplit *P* tableau des résultats des listes et les votes blancs.
3. Ecrire la fonction *nbVoix(P)* qui retourne *nvv* le nombre de voix valides des élections.
4. Ecrire la fonction *gagnant1(P)* qui retourne le gagnant majoritaire s'il existe sinon -1.

Après décompte des voix, on élimine de *V* les voix non valides en préservant l'ordre des votes.

Exemple : *V* précédent devient :

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	...	*24*
V	*3*	*3*	*2*	*5*	*2*	*3*	*2*	*1*	*3*	*5*	*5*	*2*	*3*	*4*	*6*	*5*	*5*	*5*	*6*	*6*	...	...	...

5. Ecrire la fonction *purger(V)* qui réalise l'élimination des voix non valides de *V*.
6. Si *V* est trié (ex. tri décroissant), écrire la fonction *gagnant2(V, nvv)* qui retourne le gagnant majoritaire.

II- En cas d'absence de vainqueur majoritaire, un 2nd décomptage est effectué pour le désigner. Sa raison, c'est que la participation de nombreuses listes empêche une des listes principales d'une victoire majoritaire.

Les listes qui ont moins de *10%* des voix sont éliminées (remise de leur score à *0*). Les voix de ces listes *nva* sont redistribuées totalement sur les listes restantes, en proportion de leur score, grâce à la fonction *f* supposée existée.
Si le score d'une liste est *j* alors *j* est augmenté de *f(j, nva, nvv, nlr)* où *nlr* est le nombre de listes restantes.
Exemple : Le 2nd décomptage de l'exemple précédent donne :

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*
P	*5*	*0*	*4*	*6*	*0*	*7*	*3*

7. Ecrire la fonction *scores2(P)* qui mettra à jour les scores des listes après 2nd décomptage.

III- Les listes restantes sont stockées dans un tableau *T* de taille *nlr* contenant leur numéro *nl*, leur score *sl*, leur courant politique *cl* et la liste des noms des candidats *L*.

La valeur de *cl* appartient aux *3* courants politiques classiques : gauche, droite et centre représentés respectivement par les *3* lettres *'g'*, *'d'* et *'c'*.

Le type de *T* est :
```
struct element{
        int nl, sl;
        char cl ;
        listeC *L ;
} ;
```

Avec le synonyme suivant : `typedef struct element elt ;`

Le type *listeC* est :
```
typedef struct listeCandidat{
        char nom[20];
        struct listeCandidat *s ;
} listeC ;
```

Exemple : *T* correspondant à l'exemple précédent :

i	*nl*	*sl*	*cl*	*L*	
0	*2*	*4*	*'g'*		→…
1	*3*	*6*	*'c'*		→…
2	*5*	*7*	*'g'*		→…
3	*6*	*3*	*'d'*		→…

On cherche le courant politique dominant des élections (celui qui obtient le maximum de voix). Dans le cas de plus d'un, on dit qu'il n'y pas de courant dominant.

Exemple : D'après le tableau *T*, le courant politique gauche a dominé les élections par *11* voix.

8. Ecrire la fonction *dominant(T, nlr)* qui donne le courant politique dominant s'il existe sinon *'_'*.

Dans le cas d'absence de vainqueur majoritaire après 1er et 2nd décomptage, une liste principale, ayant le maximum de voix, est chargée de former une coalition de listes réussie.
Une liste est réussie si elle est composée autour de la liste principale et totalise un score qui dépasse *50%* des voix.
S'il y'a plus d'une liste de score maximum alors celle qui appartient au courant politique obtenant le plus de voix est désignée liste principale.

Exemple : dans *T*, la liste n°*5* est la liste principale. Le score de son courant politique (*'g'*) totalise *11* voix.

9. Ecrire la fonction *nvCourant1(T, nlr, j)* qui retourne le total des voix du courant politique de la liste *j*.
10. Ecrire la fonction *listePrincipale(T, nlr)* qui retourne le numéro de la liste principale.

Une coalition est représentée par un tableau *C* de taille *nlr* (nombre de liste restantes).
C[i]=1 indique que la liste *T[i].nl* participe à la coalition et *C[i]=0* indique le contraire.

Exemple : La coalition formée des listes n°*3* et n°*5* suivante totalise *13* voix :

i	*0*	*1*	*2*	*3*
C	*0*	*1*	*1*	*0*

11. Écrire la fonction *score(T, C, nlr)* qui retourne *sc* le score de la coalition représentée dans *C*.
12. Écrire la fonction *reussir(T, C, nlr, nvv)* qui vérifie si la coalition représentée par *C* est réussie.

On calcule le total des voix obtenu par toutes les listes de coalition *C* de même courant politique que la liste *j*.

13. Ecrire la fonction *nvCourant2(T, C, nlr, j)* qui retourne le score du courant politique de la liste *j* dans *C*.

Pour le maintien d'une coalition, celle-ci doit être stable. Une coalition est stable si elle est réussie et que toutes les listes participantes sont nécessaires pour dépasser les *50%* des voix nécessaires.

Exemple : La coalition précédente est stable par contre la coalition suivante ne l'est pas car elle reste réussie même sans la liste n°*2* ou n°*3* :

i	*0*	*1*	*2*	*3*
C	*1*	*1*	*1*	*0*

14. Écrire la fonction *stable(T, C, nlr, nvv)* qui vérifie si la coalition dans *C* est stable ou pas.

IV- On veut aider la liste principale à former une coalition stable et le plus homogène possible, c'est-à-dire que la coalition est dominée par le courant politique de la liste principale (le maximum de voix de son courant politique).

Exemple : A partir de *T*, une coalition sera parfaite si la liste n°*5* s'associe avec la liste n°*2*: *C = {1, 0, 1, 0}*.

On développe d'abord un générateur de coalitions sans condition préalable sur sa formation puis on sélectionne celle, supposée unique, répondant aux conditions et critères précédents (coalition parfaite).

Exemple : Les coalitions possibles, sans conditions préalable sur leur formation, relatives aux *4* listes précédentes sont au nombre de *15* :

{0, 0, 0, 1}, {0, 0, 1, 0}, {0, 0, 1, 1}, {0, 1, 0, 0},…, {1, 1, 1, 0}, {1, 1, 1, 1}

On peut constater qu'une coalition peut être formée par la conversion en base *2* d'un entier *x* avec $1 \leq x \leq 15$

Exemple : Si *x = 3* alors *C = {0, 0, 1, 1}* et la coalition est formée des listes n°*5* et n°*6* et obtenant *10* voix.

15. Ecrire la fonction *g(nlr)* qui donne le nombre de coalitions, sans condition, formées à partir de *nlr* listes.

16. Ecrire la fonction *generer(x, C, nlr)* qui remplit *C* par la coalition correspondant à l'entier *x* (*1≤x≤15*).
17. Ecrire la fonction *coalParfaite(T, C, nlr, nvv)* qui remplit *C* par la coalition parfaite.

V- Après formation d'une coalition, un conseil de *Y* membres est à constituer à partir des candidats de ses listes. Chaque liste électorale est composée de *Y* candidats présentés par le partie correspondant.
L'ordre des candidats dans la liste reflète leur importance et par conséquent augmente leur chance de devenir membre du conseil.
Le 1er candidat d'une liste est tête de liste (le plus important), le suivant est le 2nd d'importance dans la liste, etc.
Exemple : Soit *Y=5* et le tableau *T* précédent complété par les *Y* noms de candidats de chaque liste.

i	*nl*	*sl*	*cl*	*L*									
0	*2*	*4*	*'g'*	→	'xx01'	→	'xx02'	→	'xx03'	→	'xx04'	→	'xx05'
1	*3*	*6*	*'c'*	→	'xx11'	→	'xx12'	→	'xx13'	→	'xx14'	→	'xx15'
2	*5*	*7*	*'g'*	→	'xx21'	→	'xx22'	→	'xx23'	→	'xx24'	→	'xx25'
3	*6*	*3*	*'d'*	→	'xx31'	→	'xx32'	→	'xx33'	→	'xx34'	→	'xx35'

Pour faciliter l'exemple, on représente les noms des candidats par la chaîne de caractères 'xx..'.
Les candidats 'xx01', …, 'xx31' ont plus de chance de siéger au conseil (devenir membre du conseil) qu'au reste des candidats.

Pour former le conseil, si une liste est vainqueur majoritaire alors ses *Y* candidats siègeront au conseil. Dans le cas de formation d'une coalition de liste, de chaque liste des candidats siègeront au conseil dans la limite des *Y* membres du conseil. Le nombre de candidats par liste participante à la coalition est proportionnel à son score *sl*.

Si la liste *i* de score *j* participant à la coalition *C* de score *sc*, alors *(j*Y)/sc* de ses candidats siègeront au conseil. L'application de ce calcul à toutes les listes de *C* donnera un total de sièges *Y' < Y* du fait de la division entière de *(j*Y)/sc*.
Les *Y–Y'* sièges restants seront attribués un siège par liste aux listes qui ont le plus grand reste *(j*Y)%sc*.

Les nombres de sièges attribués aux listes de la coalition sont représentés par un tableau d'entiers *R* de taille *nlr*.
Le conseil est représenté par une liste chaînée *LC* composée des noms de ses membres.
LC est déclarée globale. Sa déclaration est :

*listeC *LC = NULL ;*

Exemple : A partir du tableau *T* précédent et la coalition *C = {1, 0, 1, 0}*, la 1ère attribution de sièges donnera *1* siège à la liste n°*2* et *3* sièges à la liste n°*5*. Cependant, *1* siège restera ; celui-ci sera attribué à la liste n°*2* qui a le plus grand reste. Ainsi, *R = {2, 0, 3, 0}*

Dans le conseil *LC*, on représente d'abord les têtes de listes participantes à la coalition puis le reste des candidats de chaque liste et tout cela dans l'ordre de leur apparition dans *T* (voir exemple)

Exemple : La liste du conseil *LC* est :

LC → 'xx01' → 'xx21' → 'xx02' → 'xx22' → 'xx23'

18. Ecrire la fonction *sieges(T, C, R, Y, nlr)* qui remplit *R* par les nombres de sièges attribués aux listes de *C*.
19. Ecrire la fonction *conseil(T, R)* qui forme la liste des membres du conseil *LC*.

***

Evaluation d'une expression algébrique

La notation habituelle des expressions algébriques, sous forme dite infixe, où les opérateurs +, *, –, / figurant entre leurs 2 opérandes, souffre a priori d'une ambiguïté si l'on n'introduit pas de priorités entres les opérateurs. C'est ainsi que la notation *2 + 3 * 4* peut aussi bien désigner *2 + (3*4) = 14* que *(2+3) * 4 = 20.*

Des parenthèses ou des règles de priorité sont donc nécessaires pour lever cette ambiguïté. Nous allons étudier ici une autre notation, appelée notation algébrique post-fixée ou encore notation polonaise inversée qui ne souffre pas de ces inconvénients. Cette notation est utilisée par certains langages de programmation ou certaines calculatrices.

Exemple :

*2 3 + sin 4 * 5 6 + –*

Son équivalent en notation infixée :

*(sin(2+3) * 4) – (5 + 6)*

Les étapes de son évaluation :

+	opérateur binaire	*r = 2+3*
sin	un seul paramètre	*r = sin(r)*
*	opérateur binaire	*r = r*4*
+	opérateur binaire	*s = 5+6*
–	opérateur binaire	*r = r – s*

On suppose dans ce problème une expression algébrique sous forme d'une chaîne de caractère *C* composée de : 4 opérateurs *'+', '–', '*', '/'*, des opérandes caractères appartenant à l'ensemble *{'0','1',…'9'}* représentant des entiers entre *0* et *9* et des espaces de séparation.

La transformation d'une opérande caractère *'0' ≤ C[i] ≤ '9'* en entier équivalent se fait par la soustraction du caractère *'0'* de *C[i]* (ex. *'1' – '0' = 1*)

Exemple :

*5 2 3 * + 5 6 + –*

Les étapes de son évaluation :

*r = 2*3*
r = r + 5
s = 5+6
r = r – s

Des étapes de l'évaluation, on remarque que les opérandes de *C* sont mémorisées puis évaluées par les opérateurs de C suivant le principe LIFO (Last In First Out) implémenté par une pile d'entiers *L*.

1. Ecrire les fonctions *empiler(L, val), depiler(L)* et *sommet(L)* qui respectivement ajoute à la pile une opérande entier, supprime une opérande de la pile et donne l'opérande se trouvant à son sommet.

N.B. vous pouvez définir les fonctions *empiler(…)* et *depiler(…)* autrement.

On supposant que l'expression *C* est rédigée correctement selon la notation algébrique post-fixée.

2. Ecrire la fonction *int eval(C)* qui donne la valeur de l'expression algébrique *C*.

***

Matrice creuse

- ✓ On attachera une importance à la concision, à la clarté, et à la précision de la rédaction en langage C.
- ✓ La fonction prédéfinie autorisée est *printf*.

***

Une matrice creuse est une matrice contenant plus de 50% de valeurs nulles. La représentation d'une matrice par un tableau à *2* dimensions, conduit à une occupation inutile de l'espace mémoire par des *0*.

Dans ce problème, on travaille sur d'autres moyens de représentation des matrices creuses.

Soit une matrice d'entiers *M* de taille *N*x*N* déclarée et initialisée dans la fonction *main()*.
N est une constante globale.

Exemple : matrice *M* de taille *5*x*5* :

5	*4*	*0*	*0*	*1*
0	*5*	*0*	*0*	*0*
0	*1*	*0*	*7*	*0*
0	*0*	*0*	*0*	*0*
0	*0*	*3*	*0*	*0*

Proposition 1

La proposition 1 consiste à représenter la matrice *M* par deux tableaux *Tv* et *Tn*. Le tableau *Tv* contient les couples : valeur non nulle de *M* et son numéro de colonne. Le tableau *Tn* contient le nombre de valeurs non nulles de *M* par ligne.

Exemple : D'après l'exemple précédent *Tv* est :

5	*4*	*1*	*5*	*1*	*7*	*3*
0	*1*	*4*	*1*	*1*	*3*	*2*

et Tn est :

3	*1*	*2*	*0*	*1*

1. Définir le type du tableau d'enregistrements *Tv*.
2. Sans programmer, que pouvez-vous dire sur la dimension du tableau *Tv* et du tableau *Tn* ?

On cherche à transformer *M* en *Tv* et *Tn*.

3. Ecrire une fonction *nombreV(M)* qui donne le nombre de valeurs non nulles de *M*.
4. Ecrire la fonction *remplirTv(M)* qui retourne le tableau *Tv* obtenu à partir de *M*.
5. Ecrire la fonction *remplirTn(M, Tn)* qui remplit le tableau *Tn* à partir de *M*.

On veut exploiter la matrice *M* à travers sa représentation équivalente *Tv* et *Tn*.
Répondre aux questions suivantes à partir de *Tv* et *Tn* au lieu de *M*.

6. Ecrire la fonction *sommeC1(Tv, Tn, j)* qui retourne la somme de la colonne *j*.
7. Ecrire la fonction *sommeL1(Tv, Tn, i)* qui retourne la somme de la ligne *i*.
8. Ecrire la fonction *valeur1(Tv, Tn, i, j)* qui donne la valeur de *M[i][j]*.
9. Ecrire la fonction *trace1(Tv, Tn)* qui donne la trace de la matrice *M*.

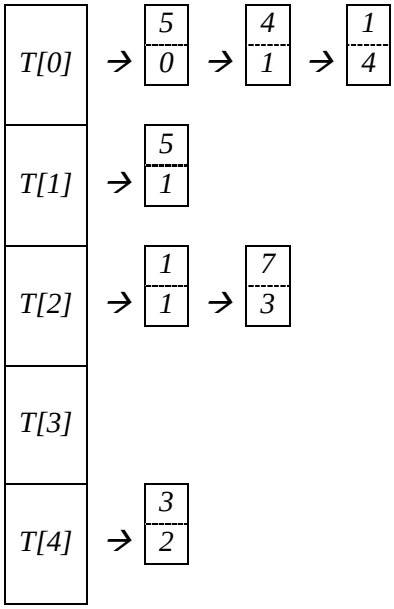
On veut afficher les valeurs de la ligne *i* de *M*. Si *i = 2* alors on affiche *0 1 0 7 0*

10. Ecrire la fonction *afficheL1(Tv, Tn, i)* qui affiche les valeurs de la ligne *i* de *M*.

Proposition 2

Consiste à représenter la matrice *M* par un tableau de listes chaînées *T* de taille *N* où chaque liste chaînée *T[i]* est composée des valeurs non nulles de la ligne *i* de *M* et les numéros de colonnes correspondantes.
T est déclaré dans la fonction *main()* et initialisé à la valeur *NULL* (*T[i] = NULL*).

Exemple : D'après l'exemple précédent *T* est :



11. Définir le type du tableau de listes *T* et donner la déclaration du tableau *T*.

On veut transformer la matrice *M* en tableau de listes *T*.

12. Ecrire la fonction *remplirT(M, T)* qui remplit le tableau *T* à partir de *M*.

On cherche à exploiter la matrice *M* à travers sa représentation équivalente *T*.
Répondre aux questions suivantes à partir de *T* au lieu de *M*.

13. Ecrire la fonction *sommeL2(T, i)* qui retourne la somme de la ligne *i*.
14. Ecrire la fonction *sommeC2(T, j)* qui retourne la somme de la colonne *j*.
15. Ecrire la fonction *valeur2(T, i, j)* qui retourne la valeur de *M[i][j]*.
16. Ecrire la fonction *trace2(T)* qui retourne la trace de la matrice *M*.

On veut afficher les valeurs de la ligne *i* de *M*.
Exemple : Si *i = 2* alors on affiche *0 1 0 7 0*

17. Ecrire la fonction *afficheL2(T, i)* qui affiche les valeurs de la ligne *i* de *M*.

On développe les fonctions permettant la transformation de *T* en *Tv* et *Tn* et inversement sans passer par *M*.

18. Ecrire la fonction *tabListe(Tv, Tn, T)* qui remplit *T* à partir de *Tv* et *Tn*.
19. Ecrire la fonction *listeTab(T, Tv, Tn)* qui remplit *Tv* et *Tn* à partir de *T*.

***

Le jeu du SuDoKu

- ✓ On attachera une importance à la concision, à la clarté, et à la précision de la rédaction en langage C.
- ✓ Aucune fonction prédéfinie n'est autorisée.
- ✓ *a* et *b* deux entiers positifs, *a/b* donne le quotient de la division et *a%b* donne son reste.

*****

On dispose d'une grille SuDoKu sous forme d'une matrice d'entiers *T* de taille *N*×*N* partiellement remplie par des valeurs entre *1* et *9*. *T* est subdivisée en *9* carrés de taille *3*×*3*.
Un carré est l'intersection entre *3* lignes et *3* colonnes de *T* (voir exemple).

N est une constante globale égale *9*.

I- Description du jeu

Le jeu consiste à remplir les cases vides de *T* avec un entier entre *1* et *9* de sorte que l'entier n'apparaît que :

- une seule fois dans une ligne de *T*
- une seule fois dans une colonne de *T*.
- une seule fois dans un carré de *T*.

Exemple de grille :

i/j	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
0	5	3			7				
1	6			1	9	5			
2		9	8					6	
3	8				6				3
4	4			8		3			1
5	7				2				6
6		6					2	8	
7				4	1	9			5
8					8			7	9

Exemple : Le carré obtenu par l'intersection des *3* dernières lignes et les *3* dernières colonnes est :

i/j	*6*	*7*	*8*
6	2	8	
7			5
8		7	9

1. Compléter le remplissage des cases vides ce carré. Les entiers possibles sont : *1, 3, 4* et *6*.

Dans ce problème nous développons des fonctions, qui permettent de compléter le remplissage de *T*.

II- Manipulation de *T*

T est une matrice déclarée et initialisée dans la fonction *main()*.
Les cases vides de *T* sont initialisées à la valeur *0*.
On suppose que l'initialisation mène à une solution de la grille SuDuKu *T*.

Exemple :

i/j	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
0	*5*	*3*	*0*	*0*	*7*	*0*	*0*	*0*	*0*
1	*6*	*0*	*0*	*1*	*9*	*5*	*0*	*0*	*0*
2	*0*	*9*	*8*	*0*	*0*	*0*	*0*	*6*	*0*
3	*8*	*0*	*0*	*0*	*6*	*0*	*0*	*0*	*3*
4	*4*	*0*	*0*	*8*	*0*	*3*	*0*	*0*	*1*
5	*7*	*0*	*0*	*0*	*2*	*0*	*0*	*0*	*6*
6	*0*	*6*	*0*	*0*	*0*	*0*	*2*	*8*	*0*
7	*0*	*0*	*0*	*4*	*1*	*9*	*0*	*0*	*5*
8	*0*	*0*	*0*	*0*	*8*	*0*	*0*	*7*	*9*

2. Ecrire la fonction *nbVide(T)* qui retourne le nombre de cases vides dans *T*.

Pour identifier les lignes et les colonnes de *T*, on utilise respectivement les indices *i* et *j*.

Une case de *T* est identifiée soit par ses indices *i* et *j* (*T[i][j]*) ou par son ordre *k* dans *T* ($0 \leq k < 9$x9) où *k* est ordonné de gauche à droite et de haut vers le bas.

La relation entre *k* et *i* et *j* est $k=i$x$N+j$.
Ainsi, $i=k/N$ et $j=k\%N$.

Exemple : Les cases *0, 8, 9, 80* de *T* ont respectivement les indices (*i, j*) suivants : (*0, 0*), (*0, 8*), (*1, 0*), (*8, 8*).

3. Ecrire la fonction *f(i, j)* qui retourne *k* correspondant aux indices *i* et *j* d'une case de *T*.

Un carré de *T* est identifié par la position *k* de la case de *T* qui se trouve à son centre.

Exemple : Les carrés *1, 2, …, 9* de *T* sont identifiés par les valeurs de *k* suivantes : *10, 13, 16, 37, 40, 43, 64, 67, 70* correspondant aux indices *i* et *j* suivants : (*1, 1*), (*1, 4*), (*1, 7*), (*4, 1*), (*4, 4*), (*4, 7*), (*7, 1*), (*7, 4*), (*7, 7*).

4. Ecrire la fonction *carre(T, i, j)* qui retourne *k* le numéro du carré où se trouve la case d'indices *i* et *j*.

III-Remplissage de *T*

Un entier *x* entre *1* à *9* est possible pour *T[i][j]=0*, si *x* n'est pas déjà dans la ligne *i*, ou dans la colonne *j*, ou dans le carré *k* contenant *T[i][j]*.

5. Ecrire la fonction *verifL(T, i, x)* qui vérifie si *x* se trouve dans la ligne *i* ou pas.
6. Ecrire la fonction *verifC(T, j, x)* qui vérifie si *x* se trouve dans la colonne *j* ou pas.
7. Ecrire la fonction *verifR(T, i, j, x)* qui vérifie si *x* se trouve dans le carré contenant la case *T[i][j]*.
8. Ecrire la fonction *estPossible(T, i, j, x)* qui vérifie si *x* est possible dans *T[i][j]*.

Pour mémoriser les entiers *x* possibles dans chaque *T[i][j]*, on utilise une matrice d'entiers *P* de taille *M*x*N* remplie par des *0* et *1*.
M est une constante globale égale à $81=9$x9.

L'indice ligne de *P* est *k* (position de *T[i][j]* dans *T*) et son indice colonne est $l=x-1$.

Exemple : *P[k][x-1]=1* signifie que l'entier *x* est possible dans *T[i][j]* et *P[k][x-1]=0* signifie le contraire.

P est déclarée dans la fonction *main()*.

Exemple : La Grille *T* (à gauche) et sa matrice de possibilité *P* (à droite). En raison de la taille de *P* qui est grande, on ne représente que ses lignes qui correspondent au carré *9* de *T*.

i/j	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
0	5	3			7				
1	6			1	9	5			
2		9	8					6	
3	8				6				3
4	4			8		3			1
5	7				2				6
6		6					2	8	
7				4	1	9			5
8					8			7	9

k/l	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
0	0	0	0	0	0	0	0	0	0
…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…
60	0	0	0	0	0	0	0	0	0
61	0	0	0	0	0	0	0	0	0
62	0	0	0	1	0	0	0	0	0
…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…
69	0	0	1	0	0	1	0	0	0
70	0	0	1	0	0	0	0	0	0
71	0	0	0	0	0	0	0	0	0
…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…
78	1	0	1	1	0	1	0	0	0
79	0	0	0	0	0	0	0	0	0
80	0	0	0	0	0	0	0	0	0

T[7][7] correspondant à la ligne *70* de *P* ne peut recevoir que l'unique valeur *3*.
T[8][6] correspondant à la ligne *78* de *P* peut recevoir les valeurs *1, 3, 4* et *6*.
T[8][8] correspondant à la ligne *80* de *P* ne peut rien recevoir. Elle est déjà remplit.

9. Ecrire la fonction *init(P)* qui initialise la matrice *P* à la valeur *0*.
10. Écrire la fonction *remplirP(T, P)* qui remplit *P* par les entiers possibles dans les cases vides de *T*.

Afin de compléter le remplissage de *T*, on suit la stratégie suivante :

- Etape1 : On remplit la matrice des possibilités *P* à partir de la grille *T*.
- Etape2 : On commence par chercher dans *P*, les valeurs possibles uniques puis on les affecte aux cases vides de *T* correspondantes.
- On recommence le remplissage de la nouvelle grille *T* obtenue après les étapes *1* et *2* et cela jusqu'au remplissage complet de *T*.

11. Ecrire la fonction *val(P, T, i, j)* qui retourne l'entier possible de *T[i][j]* s'il est unique sinon *0*.
12. Écrire la fonction *remplirT(T, P)* qui complète le remplissage de *T*.
13. Donner une version récursive de la fonction *remplirT*.

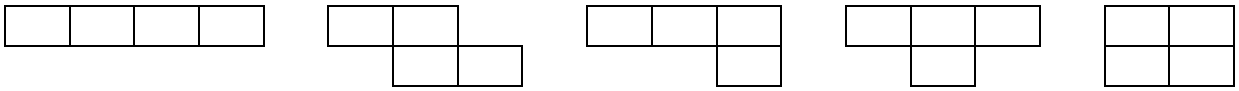
On veut optimiser le temps de réponse de cette stratégie. Pour cela, et au lieu de répéter le remplissage de *P* à chaque fois qu'on affecte un entier à une case vide de *T*, on ne modifiera que les lignes de *P* concernées par le remplissage de la case vide de *T*.

14. Sans programmer, proposer une stratégie alternative qui se base sur les remarques précédentes.
15. Écrire la fonction *remplirT2(T, P)* qui complète le remplissage de *T* selon cette stratégie.

***

Le jeu de Tetris

Tetris est un jeu vidéo célèbre qui utilise les 5 pièces suivantes:



Pièce 1 Pièce 2 Pièce 3 Pièce 4 Pièce 5

Dans ce problème, nous avons une grille de *N*x*N* entiers ($4 \leq N \leq 100$) (exemple Figure 1 pour $N=4$). Nous voulons placer une seule pièce de Tetris sur la grille telle que la somme des nombres au-dessous de la pièce est maximale.

Noter que toutes les pièces de Tetris, sauf la dernière, peuvent être tournées de 90°. Quelques pièces peuvent avoir jusqu'à quatre orientations différentes.

70	*2*	*1*	*7*
7	*1*	*30*	*6*
4	*30*	*30*	*5*
3	*1*	*30*	*2*

Figure 1

70	*2*	*1*	*7*
7	*1*	*30*	*6*
4	*30*	*30*	*5*
3	*1*	*30*	*2*

Figure 2

70	*2*	*1*	*7*
7	*1*	*30*	*6*
4	*30*	*30*	*5*
3	*1*	*30*	*2*

Figure 3

70	*2*	*1*	*7*
7	*1*	*30*	*6*
4	*30*	*30*	*5*
3	*1*	*30*	*2*

Figure 4

N'importe quelle orientation est acceptable tant que la pièce est entièrement à l'intérieur de la grille. Par exemple, la pièce 1 peut être placée sur la 1ère ligne de la grille (Fig2), avec une somme de 80.
Elle peut également être placée, par exemple, sur la 3ème colonne, rapportant une somme de 91 (Fig3).

En fait, dans une grille 4x4, nous pouvons avoir 77 manières de placer les pièces de Tetris.
Dans l'exemple, la plus grande somme est réalisée grâce à la pièce 4 avec une somme de 120 (Fig4).

On considère une grille *T* de taille *N*x*N*. *N* est une constante globale.

1. Ecrire les fonctions *f1, f19, f2, f29, f3, f39, f318, f327, f4, f49, f418, f427* et *f5* qui retournent respectivement les sommes obtenues par l'application des pièces : pièce *1* et sa rotation de *90°*; pièce *2* et sa rotation de *90°* ; pièce *3* et ses rotations de *90°*, *180°* et *270°*, pièce *4* et ses rotations de *90°*, *180°* et *270°* et enfin la pièce *5* sur la grille.

2. Écrire une fonction qui retourne la plus grande somme de *T*.

***

Tri de chaînes de caractères

On considère *N* chaînes de caractères de taille maximale *M* stockées dans une matrice *T*.
N et *M* sont des constantes globales.
Ecrire une fonction qui prend en paramètre *T* et qui permet de trier et d'afficher les chaînes de la manière suivante :
On trie les caractères de chaque chaîne puis on trie les chaînes entre elles.

Exemple 1 :

Avant tri :	Après tri :
"znb"	"ar"
"cd"	"bnz"
"z"	"cd"
"ra"	"z"

Exemple 2 :

Avant tri :	Après tri :
"elk"	"ay"
"rmoze"	"efz"
"ezf"	"ekl"
"ya"	"emorz"
"umh"	"hmu"
"ijlk"	"ijkl"

***

Voyage à prix bas

Les prix de voyages par avion sont fous ! Le prix d'un billet est déterminé par plusieurs facteurs, et n'est pas d'habitude directement lié à la distance du voyage. Les agences de voyages essayent d'être innovatrices, utilisant parfois des billets pouvant servir à différentes destinations afin de proposer des voyages à prix bas.

Pourtant, les compagnies aériennes sont conscientes de ce comportement, et d'habitude exigent qu'un billet couvrant un voyage le soit dans l'ordre indiqué des villes.

Par exemple, si vous avez un billet pour voyager d'une ville 1 à une ville 2 puis à la ville 3, vous n'êtes pas autorisé à utiliser seulement le tronçon de billet de la ville 2 à la ville 3. Vous devrez toujours commencer de la première ville mentionnée sur le billet. En plus, vous n'êtes pas autorisé à voyager de la ville 1 à la ville 2, voler quelques parts ailleurs, revenir puis continuer votre vol de la ville 2 à la ville 3.

Prenons un exemple, supposons qu'une compagnie aérienne propose trois types de billets :

Billet #*1* : Ville *1* à Ville *3* à Ville *4*	225.00 Dh
Billet #*2* : Ville *1* à Ville *2*	200.00 Dh
Billet #*3* : Ville *2* à Ville *3*	50.00 Dh

Supposons que vous voulez voyager de la ville 1 à la ville 3, il y a deux façons d'y arriver en utilisant seulement les choix possibles de billets.

1er choix : Billet #1 au prix de 225.00 Dh en utilisant que le premier tronçon du billet.
2ème choix : Billet #2 au prix de 200.00 Dh et Billet #3 au prix de 50.00 Dh

Le 1er choix est le moins cher.

1. Ecrire une fonction qui permet de remplir une matrice *M*, à partir des données du fichier d'entrée.

2. Donnons une suite d'offres de billets, un ou plusieurs itinéraires de voyages, vous devez déterminer comment avoir des billets à prix bas.

Fichier d'entrées :

Les données d'entrées consistent dans de multiples tests, chacun décrit une suite d'offres de billets et une suite d'itinéraires de voyages.

Chaque test commence par une ligne contenant nombre d'offres de billets (*NB*), suivi par *NB* descriptions d'offres, une par ligne.
Chaque description est composée d'un nombre entier positif spécifiant le prix du billet, le nombre de villes traversées, et les villes. Chaque ville est identifiée par un nombre entier unique.

La ligne qui suit les itinéraires contient le nombre de voyages à vouloir à prix bas (*NV*), suivie par *NV* descriptions de voyage. Chaque ligne comprend le nombre de villes dans le voyage y compris la ville de départ, suivi par les numéros des villes, donnés dans l'ordre de leur visite.

Dans le cas du test, les offres de billets et de voyage ne dépasseront pas *20*. Chaque voyage ou billet aura une liste de villes de *1* à *9*.

Le dernier cas est suivi par une ligne contenant *0*.

Fichier de sortie

Pour chaque voyage, les deux lignes de sortie contiennent le numéro de test, le numéro de voyage, le prix minimum, et les numéros de billets utilisés dans le voyage dans l'ordre de leur utilisation.

Suivre le format de sortie indiqué ci-contre.

Exemple de données d'entrées	Exemple de données de sorties correspondant
3 *225 3 1 3 4* *200 2 1 2* *50 2 2 3* *1* *2 1 3* *3* *100 2 2 4* *100 3 1 4 3* *200 3 1 2 3* *2* *3 1 4 3* *3 1 2 4* *0*	Test *1*, Voyage *1*: Prix = *225.00 Dh* Billets utilisés : *1* Test *2*, Voyage *1*: Prix = *100.00 Dh* Billets utilisés : *2* Test *2*, Voyage *2*: Prix = *300.00 Dh* Billets utilisés : *3 1*

***

Filtre non causal

I- Lissage d'un signal :

L'objectif de cet algorithme est de développer un filtre non causal, c'est-à-dire une fonction qui lisse un signal (considéré comme un tableau de valeurs) en utilisant une fenêtre glissante d'une taille *n* pour moyenner les valeurs du signal. Une valeur filtrée *p* est la moyenne des valeurs non encore filtrées se trouvant dans une fenêtre centrée en *p*.

On considère un tableau non filtré *Tnf* et un tableau filtré *Tf* de taille *N* (déclaré constante globale)

Exemple :

Un signal de valeurs :

2	1	4	5	3	6	3	7

Devient après filtrage:

1.5	2.3	3.3	4	4.7	4	5.3	5

1. Ecrire la fonction *filtreSig(Tnf, Tf)* qui filtre le signal *Tnf* dans *Tf*.

II- Lissage d'une image

A l'image du filtre non causal, cette fonction calculera la couleur d'un point *p* de la nouvelle image en faisant la moyenne des couleurs des points de l'image source se trouvant dans une fenêtre carrée centrée en *p*.

La figure ci-dessous propose un exemple de calcul d'une image de largeur *4* et de hauteur *5* avec un filtre de taille *3*.

147	*14*	*97*	*89*
234	*95*	*1*	*187*
153	*17*	*156*	*137*
230	*73*	*135*	*79*
37	*232*	*32*	*48*

Image source

122.5	*98*	*80.5*	*93.5*
110	*101.6*	*88.1*	*111.2*
133.7	*121.6*	*97.8*	*115.8*
123.7	*118.3*	*101*	*97.8*
143	*123.2*	*99.8*	*73.5*

Image filtrée

Soit une matrice non filtré *Wnf* et une matrice filtré *Wf* de taille *N*x*M* (déclarés constantes globales)

2. Ecrire la fonction *filtreImage(Wnf, Wf)* qui filtre l'image *Wnf* dans *Wf*.

III- Compression d'un signal

On peut considérer qu'un signal est une suite de valeurs représentant les amplitudes du signal. Une méthode de compression simple est de remplacer cette suite de valeurs par une suite de couples de valeur (occurrence, amplitude).

Par exemple la suite *10, 10, 10, 4, 4, 5, 5, 5, 5, 8* peut donner la suite *3, 10, 2, 4, 4, 5, 1, 8* (on a ici économisé *2* valeurs). La taille du tableau *Tc*, représentant le signal, est très grande. La fin du signal est identifiée par la valeur *-1*.

3. Ecrire la fonction *compresse(Tc)* qui permet de compresser le signal *Tc* dans le même tableau.

IV- Qualité du signal et compression.

La fonction précédente permet de récupérer le signal identique à l'original après décompression.

4. Récrire la fonction *compresse(Tc)* qui garde une qualité de *80%* du signal d'origine après décompression (c'est-à-dire que si *2* points successifs qui diffèrent de *20%* alors ils sont supposés égaux et la 1ère valeur est compressée).

***

Réseau d'ordinateurs

Partie 1 : Modélisation du réseau

I- On considère un réseau informatique composé d'un ensemble d'ordinateurs reliés entre eux. Chaque ordinateur est caractérisé par un nom et possède *N* ports de Liaison qui y sont rattachés (ex. ports USB) et qui permettent de connecter l'ordinateur à d'autres ordinateurs.
Ainsi, une Liaison relie deux ordinateurs. Elle est caractérisée par un débit (entier) et ses deux ordinateurs d'extrémités rattachés.

Nous allons définir un type Ordinateur et un type Liaison.

D'un point de vue de la représentation mémoire, il est impossible que l'Ordinateur contienne les Liaisons, et qu'une Liaison contienne les Ordinateurs.
Un Ordinateur contient donc un ensemble de pointeurs sur des variables de type Liaison et une Liaison aura des pointeurs sur ces deux extrémités de type Ordinateurs.

Chaque ordinateur ne peut être connecté qu'au maximum *N* autres ordinateurs. Un port non connecté est de valeur *NULL*.

1. Proposer les types de données utilisées dans ce problème.
2. Proposer un exemple d'implémentation mémoire du réseau.

On supposant un tableau d'ordinateurs *T* de taille *M* (*M* ordinateurs).

3. Écrire une fonction qui retourne le nombre de connexion effective dans le réseau.

***

Partie 2 : Détection d'un Fanion Début et Fin

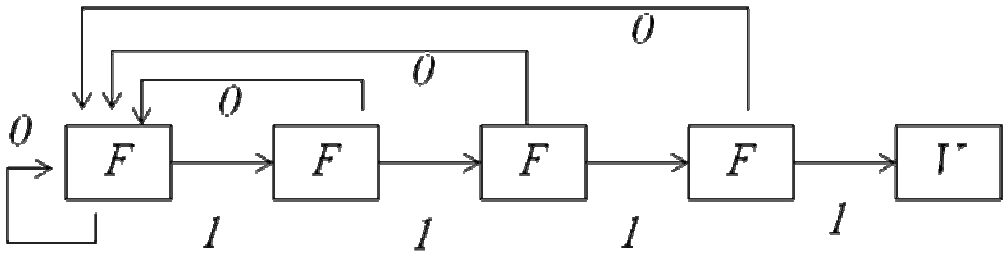
Une liaison permet une transmission de données (des bits *0* et *1*) entre ses deux ordinateurs d'extrémités. La liaison est de type série permettant de transmettre bit par bit les données.
La séquence de bits qui transite, sur la liaison, d'un ordinateur *X* à un ordinateur *Y* est représentée par un tableau d'entiers *T* de très grande taille de valeurs *0* et *1*.

T contient, au début, une séquence continue de bits inutiles (absence de données à transmettre de *X* vers *Y*), suivie d'une séquence de bits pour détecter le début de la transmission (appelée Fanion Début), suivie d'une séquence de bits utiles (début de transmission), suivie d'une séquence de bits pour détecter la fin de la transmission de donnée (Fanion Fin), suivie de séquence de bits inutiles, etc.

Le Fanion début est identique au Fanion fin.
La suite de bits permettant de détecter le début et la fin de la transmission est *{1, 1, 1, 1}*

Remarque : pour éviter de confondre une suite de bits utile identique à celle du Fanion, l'émetteur code ses données à transmettre de sorte de ne pas avoir une telle suite au sein de ses données.

Pour détecter le Fanion, on propose l'automate *A* suivant : (partie concernant le Fanion début)



F : indique Faux (Fanion début non détecté)
V : indique Vrai (détection du Fanion début).
0 et *1* sont les bits du tableau *T* transits de l'ordinateur *X* vers *Y*.

N.B. L'ordre de transmission de bits est l'ordre croissant des indices de *T*.

1. Proposer le type de *A*.
2. Ecrire une fonction qui permet de l'implémenter en mémoire.
3. Ecrire une fonction qui prend en paramètre le tableau *T* et l'automate *A* et retourne *1* s'il détecte le Fanion sinon *0*.
4. Sans programme, proposer le schéma complet de l'automate *A* qui permet de détecter le début et la fin de la transmission.

***

Bonne chance

Les listes chaînées : Un Essai didactique

Proposé par Pr. D. El Ghanami

Ecole Mohammadia d'Ingénieurs

Mai 2011

Des fonctions de manipulation d'une Liste

1. Définition du nœud :

```
typedef struct un_noeud{
        int x;
        struct un_noeud *s;
} nœud ;
```

Déclaration d'une liste ***L*** vide *:*

*nœud *L = NULL ;*

L	Signifie la liste (pointeur sur le 1er nœud s'il existe ou *NULL* sinon).
**L*	1er nœud de la liste s'il existe sinon cette utilisation est fausse.
**L.x et *L.s*	Entier et pointeur du 1er nœud (autre écriture *L→s*, ce dernier pointe sur le 2ème nœud)
**(*L.s).s*	ou *L→s→s* : Pointeur sur le 3ème nœud
L→	est équivalent à **L*. c-à-d qu'à la droite de → on ne peut avoir qu'un pointeur.

Cas d'une liste d'enregistrements (ex. des points de coordonnées *x* et *y*) alors accès aux variables d'un l'enregistrement: **L.p.x* (autrement *L→p.x*)

Si le passage de ***L*** à une fonction se fait par valeur (ex. ***f(L)*** **)**, alors les modifications de *L* seront fait sur la copie.

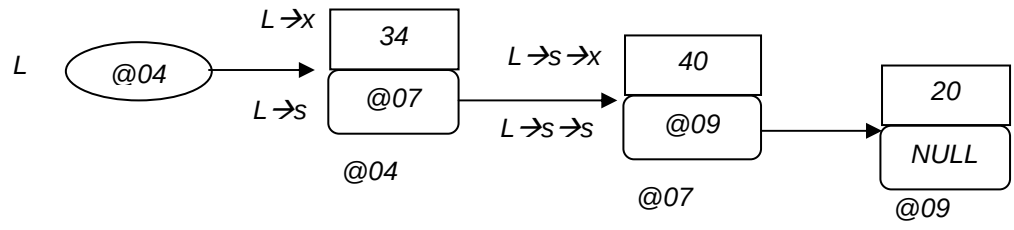
Dans le cas de modification de *L* au sein de ***f*** (cas : ajout, suppression d'un nœud au début) alors la nouvelle valeur de *L* doit être retournée. Attention au cas où la modification se fera sous condition *if*.

Autre manière est le passage par adresse de *L* (ex. *f(&L)*), alors le prototype de la fonction sera : *f(nœud **pL)*. La manipulation la plus simple au sein de *f* est la suivante :
*nœud *p ;*
*p=*pL ;*
La manipulation de la liste (accès aux nœuds se feront par le pointeur *p*).
Si modification de *L* (ex. suppression du nœud début) alors : **pL = p→s*

Si la liste *L* est déclarée variable globale, alors elle n'est ni passée en paramètre ni retournée et sa modification dans une fonction est valable partout.

L'allocation des nœuds ne doit pas se faire dans la PILE au risque d'avoir une rupture dans le chaînage de la liste mais dans le TAS grâce à l'instruction malloc.
La libération de la mémoire (instruction free) n'est pas importante dans le programme info-CPGE (rappel : intérêts de malloc et les pointeurs)

Soit la liste suivante :



Distance entre nœuds :
Soit *p1=@04*, *p2=@07*, *p3=@09* et *p*

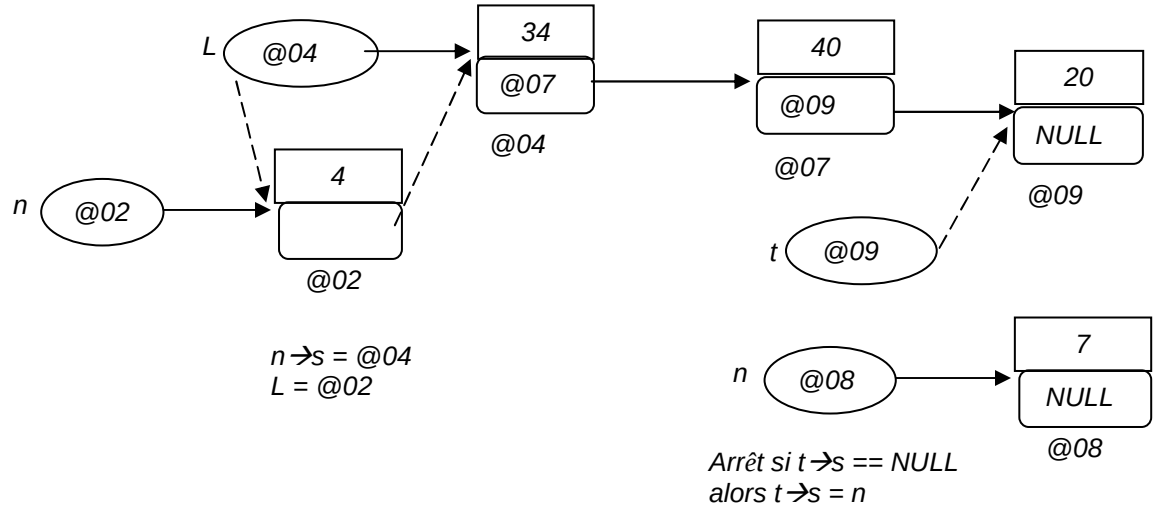
Pour que *p* indique le 1er nœud, il faut que : *p == p1* : distance *0* entre **p* et **p1* (même nœud)

p→s==p2 : distance *1* entre **p* et **p2* (deux nœuds voisins)
p→s→s==p3 : distance *2* entre **p* et **p3*

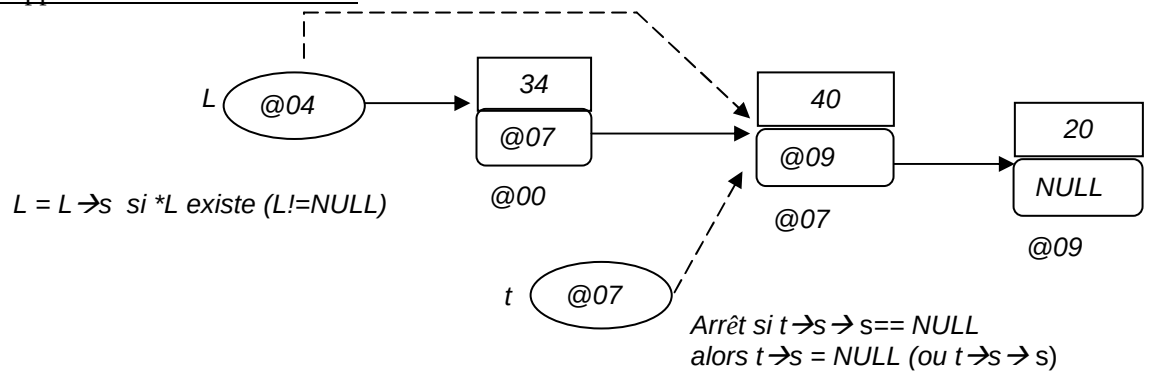
Exemples:

- Pour supprimer le dernier nœud, il faut que *p* pointe sur l'avant dernier nœud, alors la distance entre *p* et *NULL* est de *2* : *while(p→s→s !=NULL) p=p→s ;*
- Pour ajouter un nœud à la fin de la liste, il faut que *p* pointe sur le dernier nœud alors la distance entre *p* et *NULL* est de *1* : *while(p→s !=NULL) p=p→s ;*
- Pour traiter une liste, il faut la parcourir et pointer sur chaque nœud. A la fin du traitement le pointeur sera *NULL* : *while(p→s !=NULL){Traitement ; p=p→s ;}*
  Attention au cas particuliers : liste vide, liste avec un seul élément, ajout fin équivalent à ajout début.
- Le déplacement de *p (p=p→s)* peut se faire par rapport à une autre condition d'arrêt (pointeur *q* équivalent à *NULL*) suivant la même logique précédente.
- Le déplacement de *p* peut se faire d'une manière déterministe (un certain nombre de fois)

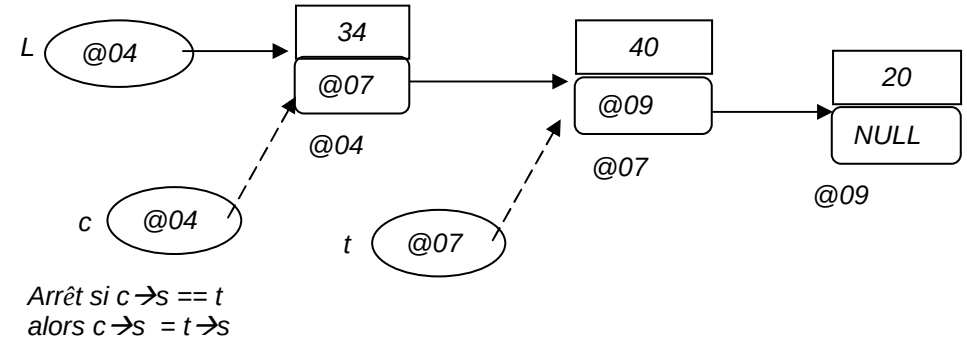
Ajout début et à la fin de la liste:



Supprimer au début et à la fin :



Supprimer une valeur :



noeud creerN(int i, nœud *p){* *noeud *n =(noeud*)malloc(sizeof(noeud));* *n→x = i;* *n→s = p;* *return n;* *}*	Création d'une case mémoire initialisée par les paramètres i et p et retour de son adresse.
noeud ajoutD(noeud *l, noeud* n){* *n→s = l;* *return n;* *}*	Vers2 (passage par @ de L) *void ajoutD(noeud **pl, noeud* n){* *noeud *p=*pl;* *n→s = p;* **pl=n;* *}*
noeud supprimeD(noeud *l){* *if(l != NULL) l = l→s;* *return l;* *}*	*void supprimeD(noeud **pl){* *noeud *p=*pl;* *if(p != NULL) *pl = p→s;* *}*
nœud sommetPile(noeud *l){* *return l;* *}*	Retour de l'@ du nœud de la tête de liste. L'avantage c'est le retour de plusieurs variables d'un nœud.
int nbNoeud(noeud l){* *int i = 0;* *while(l != NULL){* *i++; l = l→s;* *}* *return i;* *}*	
noeud seDeplacer(noeud* l, int i){* *int j;* *for(j = 0 ; j < i ; j++) l = l→s;* *return l;* *}* *void inverserListe(noeud *l){* *noeud *t1, *t2;* *int a, i, j, n = nbNoeud(l);* *for(i = 0; i < n/2; i++){* *t1 = seDeplacer(l, i);* *t2 = seDeplacer(l, n-1-i);* *a = t1→x;* *t1→x = t2→x;* *t2→x = a;* *}}*	l = l→s est le déplacement d'un nœud à l'autre i = 0 se déplacer 0 fois → le 1er nœud i = 1 se déplacer 1 fois → le 2ème nœud i = nbNoeud-1 → le dernier nœud i = nbNoeud → NULL donc : 0≤i<nbNoeud
noeud ajouterFin(noeud *l, noeud* n){* *noeud *c;* *if(l == NULL)* *return ajouterDeb(l, n);* *c = seDeplacer(l, nbNoeud(l) – 1);* *n→s = NULL ;* *c→s = n;* *return l;* *}*	Si liste est vide alors ajouter fin revient à ajouter au début (la nouvelle tête à retourner) sinon il faut déplacer *c* sur le dernier nœud et retourner quelque chose qui conserve *L*. *void ajouterFin(noeud **pl, noeud* n){* *noeud *c=*pl;* *n→s=NULL ;* *if(c == NULL) *pl=n ;* *else{ while(c→s!=NULL) c=c→s;* *c→s = n; }* *}*
noeud supprimerFin(noeud *l){*	Pour supprimer le dernier nœud, il faut mettre la valeur du pointeur de l'avant dernier nœud à NULL.

*noeud *c;* *if(l == NULL) return NULL;* *if(l→s == NULL)* *return supprimerD(l);* *c = seDeplacer(l, nbNoeud(l) – 2);* *c→s = NULL ;* *return l;* *}*	Il faut se déplacer jusqu'à ce nœud ou c = l ; while(c→s→s!=NULL) c=c→s; Cas particuliers : si liste vide : rien à supprimer Si la liste contient un seul nœud, ce nœud n'a pas de précédent et donc le faut le traiter comme supprimer début.
nœud sommetFile(noeud *l){* *noeud *c;* *if(l→s == NULL)* *return sommetPile(l);* *c = seDeplacer(l,nbNoeud(l) – 1);* *return c;* *}*	Déplacer le pointeur jusqu'au dernier nœud c = l ; while(c→s!=NULL) c=c→s;
noeud rechVal(noeud* l, int v){* *while(l != NULL && l→x != v)* *l = l→s;* *return l;* *}*	Rechercher une valeur dans la liste. Retour de NULL s'il n'est pas dans la liste ou sa position (pointeur qui l'indique) sinon. Si l est NULL alors l→x est interdite mais dans le while, l!=NULL la précède
void modifVal(noeud l, int v){* *noeud *t = rechVal(l,v);* *if(t != NULL) t→x++;* *}*	
noeud supprimerVal(noeud* l, int v){* *noeud *t,*c = l;* *t = rechVal(l, v);* *if(t == NULL) return l;* *if(t == l) return supprimerD(l);* *while(c→s != t)* *c = c→s;* *c→s = t→s;* *return l;* *}*	*void supprimerVal(noeud**p l, int v){* *noeud *t,*c = *pl;* *t = rechVal(c, v);* *if(t != NULL){* *if(t == c) *pl= c→s;* *else{* *while(c→s != t) c = c→s;* *c→s = t→s;* *}}* *}*
*void afficherDirect(noeud *l){* *while(l != NULL){* *printf("%d ", l→x);* *l = l→s;* *}* *}*	
*void afficherInverse(noeud *l){* *if(l != NULL){* *afficherInverse(l→s);* *printf("%d ", l→x);* *}* *}*	version itérative : *for(i= nbNoeud(l)-1 ; i>=0 ; i--){* *c = seDeplacer(l, i);* *printf("%d", c→x) ;* *}* *On peut utiliser une pile explicite*
*void triBulle(noeud *l){* *noeud *t;* *int i, j, a, n = nbNoeud(l);* *for(i=0; i <n–1; i++){* *t = l;* *for(j=0; j<n-1-i; j++){*	Tri à bulle d'une liste. Grâce à nbNoeud et le principe du tri qui consiste à comparer les valeurs 2 à 2 (valeur avec son suivant) et ceci jusqu'à la fin d'un parcours et reprendre le même principe, l'écriture devient similaire au cas de tableau. t=t→s (équivalent à j++) et t=l (éq. à remettre j à 0). Vers.2: *for(i=0; i <n–1; i++){* *t = l ;*

```
                    if(t→x > t→s→x){
                        a = t→x;
                        t→x = t→s→x;
                        t→s→x = a;
                    }
                    t = t→s;
                }
}}
```

```
                while(t→s→s != NULL{
                    if(t→x > t→s→x){
                        a = t→x;
                        t→x = t→s→x;
                        t→s→x = a;
                    }
                    t = t→s;
}}
```

```
void triSelect(noeud *l){
        int a;
        noeud *t = l, *pmax;
        while(t != NULL){
                pmax = maxListeP(t);
                a = t→x;
                t→x = pmax→x;
                pmax→x = a;
                t = t→s;
        }
}
```

Tri par sélection se base sur la recherche du max dans une liste paramétrée par le pointeur d'où commencer le permuter avant le premier. Dans le cas de recherche du max dans toute la liste, il faut supprimer t du paramètre et initialiser pmax à l.

```
noeud* maxListeP(noeud *t){
        noeud *pmax = t;
        while(t != NULL){
                if(t→x > pmax→x) pmax = t;
                t = t→s;
        }
        return pmax; }
```

```
noeud* convEntCh(noeud *l, int x){
   while(x != 0){
        l=ajouterDeb(l,creerN(x%10+'0'));
        x=x/10;
   }
  return l;
}
```

Conversion d'entier en chaîne en gérant une liste qui permet d'inverser les chiffres de l'entier. Ex. Récursive avec ajoutFin

```
noeud* convEntChR(noeud *l, int x){
    if(x!=0) l = convEntChR(l, x/10);
            return ajouterFin(l, creerN(x%10+'0'));  }
     else return NULL;        // liste encore vide au départ
}
```

```
int verifExpression(noeud *l, char T[]){
        int i = 0, b = 1;
        while(T[i] != '\0' && b == 1){
           if(T[i] == '(')
             l=ajouterDeb(l, creerN(T[i]));
             if(T[i] == ')')
                 if(sommetPile(l) == '(')
                     l = supprimerDeb(l);
                 else b = 0;
           i++;
        }
        if(l != NULL) b = 0;
        return b;
}
```

La vérification d'une expression si elle est bien parenthésée : Ex. ((a+b)+c)……

L'expression peut avoir d'autres symboles en plus de '(pour regrouper les termes de l'expression '{' , '[' , … Le principe de la solution reste le même.

La solution consiste à chaque fois qu'on rencontre une '(' de l'empiler dans la liste et si en rencontre une ')' alors en dépile (dans le cas de plusieurs symboles il faut que ça coïncide ex. '}' avec '{' en sommet de la liste)

A la fin, il faut que la liste soit vide sinon il y a un symbole supplémentaire dans l'expression qui reste non dépilé.

```
int maxListeV(noeud *l){
        int vmax;
        if(l == NULL)  return -1;
        vmax = l→x;
        while(l != NULL){
            if(l→x > vmax)  vmax=l→x;
            l = l→s;
        }
        return vmax;
}
```

Recherche de la valeur max dans une liste. La valeur -1 si la liste est vide. Rechercher le pointeur du max :

```
noeud* maxListeP(noeud *l){
        noeud *pmax = l;
        while(l != NULL){
                if(l→x > pmax→x)        pmax = l;
                l = l→s;
        }
        return pmax;
}
```

```
void ajoutApVal(noeud* l, noeud *n, int v){
        noeud *t = rechVal(l, v);
```

```
void supprimerApVal(noeud* l, int v){
        noeud *t = rechVal(l, v);
```

```
        if(t != NULL){
                n→s = t→s ;
                t→s = n;
}}
```

```
        if(t == NULL || t→s == NULL)
                printf("impossible");
        else    t→s = t→s→s;
}
```

```
nœud*  ajoutAvV(noeud* l,noeud *n,int v){
        noeud *c = l, *t = rechVal(l, v);
        if(t == NULL)  return l ;
        if(t == l)  return ajoutDeb(l, n) ;
        while(c→s != t) c = c→s;
        n→s = t ;
        c→s = n;
        return l;
}
```

```
nœud* supprimerAvVal(noeud* l, int v){
        noeud *c=l, *t = rechVal(l, v);
        if(t == NULL || t == l) return l ;
        if(t→s == NULL)
                return supprimerDeb(l);
        while(c→s→s != t)
                c = c→s;
        return l;
}
```

Récursivité

Présenté par D. ELGHANAMI
Pr. À l'EMI

Mai 2011

La récursivité

- Méthode de programmation puissante qui permet d'exprimer d'une manière élégante la solution de problèmes.
- Une fonction s'appelle elle-même
  - Une condition d'arrêt
  - Un appel
- Fonctionnement
  - La pile des appels
- Exemples :
  - Récurrences mathématiques classiques
  - Tour d'Hanoï
  - Tri rapide, tri fusion, …
  - Recherche dichotomique, …
  - ….

Les questions clé de la récursivité

1. Comment exprimer la solution du problème en terme de la solution du même problème ?
2. En quoi chaque appel récursif diminue-t-il la taille du problème ?
3. Quels cas du problème ont des solutions simples et serviront de cas de base?
4. Les appels récursifs convergent-ils vers un cas terminal ?
5. Les types de parcours et méthode de programmation.
6. Dans quel cas on peut utiliser l'Itération et/ou la récursivité.

- **Récursivité simple** : un appel récursif

  Exemple : La fonction puissance $x \rightarrow x^n$

$$x^k = \begin{cases} 1 & si\ k = 0 \\ x * x^{k-1} & sinon \end{cases}$$

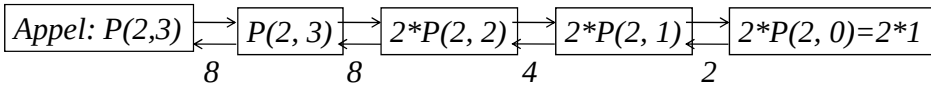
$x^k = 1/x * x^{k+1}$

Récursivité terminale ?

Programme en C :

```
int   puissance(float x, int k){
  if (k == 0)                      ← Condition d'arrêt
    return  1 ;
  else
    return  x*puissance(x, k–1);   ← Appel récursif
}
```

Code avant l'appel



- **Récursivité multiple** : plus d'un appel récursif.

Exemple : Calcul de combinaison

$$C_n^p = \begin{cases} 1 & si\ p = 0\ ou\ p = n \\ C_{n-1}^p + C_{n-1}^{p-1} & sinon \end{cases}$$

Récursivité imbriquée :

$$A(m,n) = \begin{cases} n+1 & si\ m = 0 \\ A(m-1, 1) & sinon\ \ si\ m > 0\ et\ n = 0 \\ A(m-1, A(m, n-1)) & sinon \end{cases}$$

Types de stratégies récursives

- Ascendante
- Descendante
- Par divisions successives (diviser pour régner)

Expemple : Somme des carrés de x à y

$x^2 + (x+1)^2 + \ldots. + y^2$

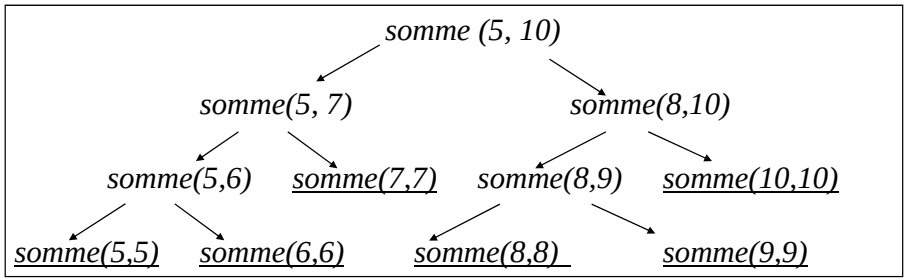
$$somme\ (x, y) = \begin{cases} x^2 & si\ x = y \\ x^2 + Somme(x+1, y) & sinon \end{cases}$$

$$somme\ (x, y) = \begin{cases} x^2 & si\ x = y \\ y^2 + Somme(x, y-1) & sinon \end{cases}$$

$$somme(x, y) = \begin{cases} x^2 & si\ x = y \\ somme(x, m) + somme(m+1, y) & sinon \end{cases}$$

avec $m = (x + y)/2$

Arbre de l'algorithme



Le mécanisme de la récursivité

Récursivité et boucle

- La récursivité peut simuler l'effet d'une boucle.
- Reçoit en paramètre le nombre d'itérations à réaliser (compteur)

```
void boucle (int i){
    if (i < N){
        Instructions;
        boucle(i+1);
    }
}
```

```
for(i=0; i<N; i++){
    Instructions;
}
```

Qu'arriverait-il si on inversait ces 2 opérations?

```
void boucle (int i){
    if (i < N){
        boucle(i+1);
        Instructions;
    }
}
```

Code exécuté après retour de l'appel

Récursivité et Itération

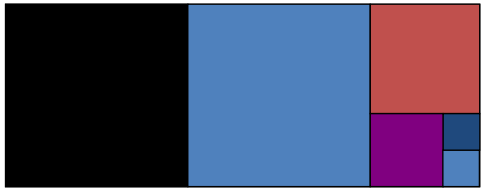
```
ALGORITHME P(U)
    si C(U) alors
        D(U)
        P(a(U))
    sinon
        T(U)
Fin.
```

```
ALGORITHME P(U)
    tant que C(U) faire
        D(U)
        U ← a(U)
    Fin tant que
    T(U)
Fin.
```


Algorithme d'Euclide pour le PGCD

- Trouver le plus grand carré permettant de le paver
- Découper en carré de côté = hauteur (largeur) du rectangle de façon à former un rectangle plus petit.
- Recommencer jusqu'à ce que la figure restante soit un carré
- Résultat : Le côté du carré est le pgcd de a et b.

Rectangle de cotés 65 et 25

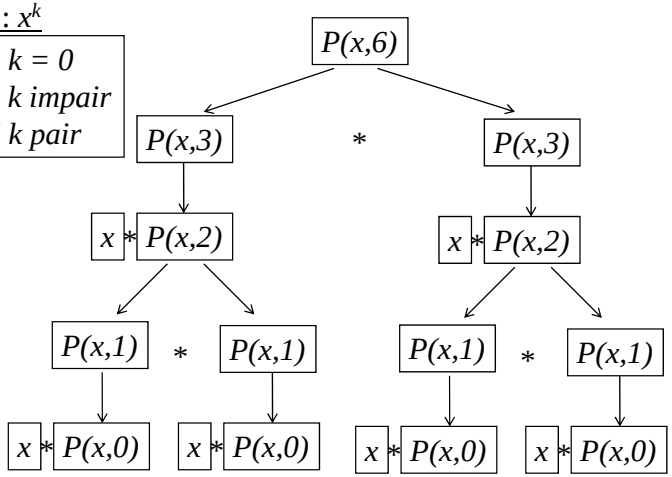


```
while (a != b) {
  if (a > b) a = a – b ;
  else b = b – a ; }
```

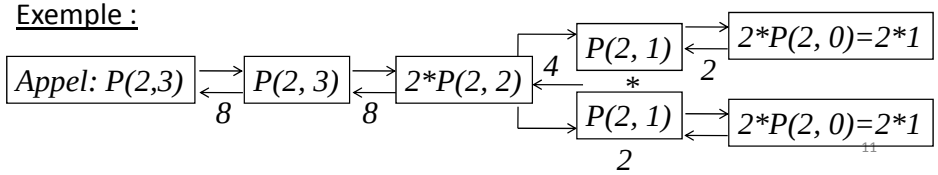
$$PGCD(a,b)=\begin{cases} PGCD(a-b,b) & si\ a>b \\ PGCD(a,b-a) & si\ a<b \\ a & sinon \end{cases}$$

Fonction puissance : x^k

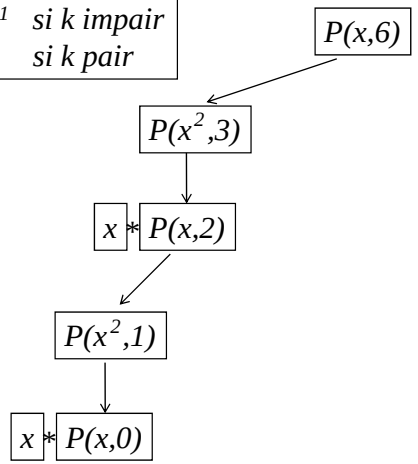
$$x^k=\begin{cases} 1 & si\ k=0 \\ x * x^{k-1} & si\ k\ impair \\ x^{k/2} * x^{k/2} & si\ k\ pair \end{cases}$$



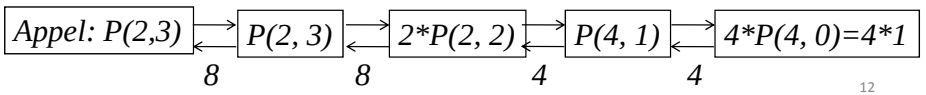
Exemple :



$$x^k=\begin{cases} 1 & si\ k=0 \\ x * x^{k-1} & si\ k\ impair \\ (x^2)^{k/2} & si\ k\ pair \end{cases}$$

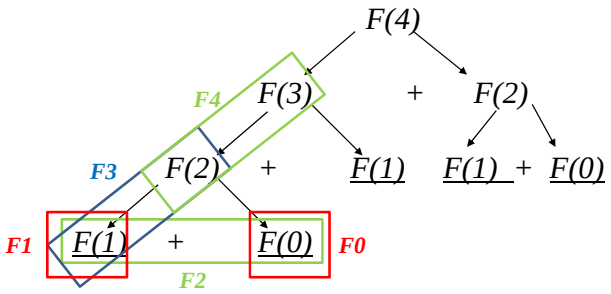


Exemple :



Suite de Fibonacci :

$F(0) = 1$
$F(1) = 1$
$F(n) = F(n-1) + F(n-2)$



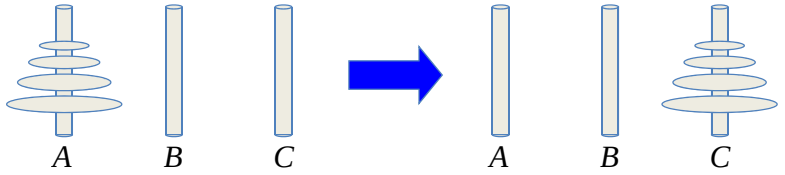
F[0]	F[1]	F[2]	F[3]	F[4]

Taille du tableau ?

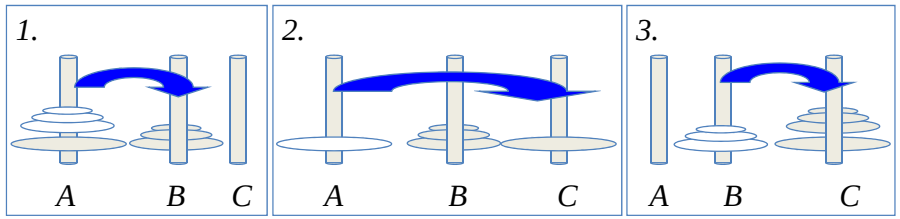
F[0]	F[1]	F[2]

Les Tours de Hanoï

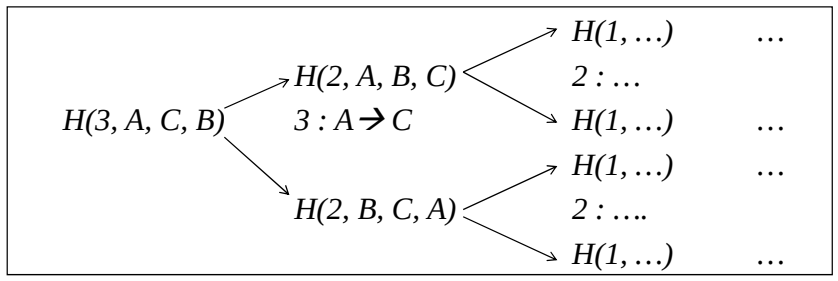
- Objectif : Déplacer tous les disques de la tige A à la tige C
  - La tige B sert d'emplacement temporaire
  - On ne peut empiler un disque sur un disque plus petit



- On décompose le problème en un problème plus simple
  1. On déplace les (n-1) disques, sauf le nème sur la tige B
  2. On déplace le nème disque sur la tige C
  3. On déplace tous les disques de la tige B sur la tige C



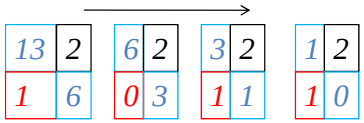

```
Hanoi (n, A, C, B)
   si (n >= 1)
     Hanoi (n – 1, A, B, C)
     déplacer n A → C
     Hanoi (n – 1, B, C, A)
   fin-si
```



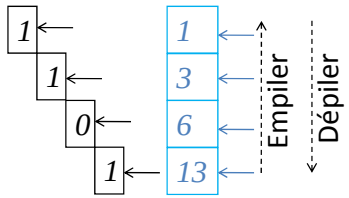
La pile des appels

- Une pile est utilisée pour mémoriser:
  - l'état des variables locales
- À chaque appel, les valeurs des variables locales sont empilées
  - l'instruction return ou fin provoque la descente de la pile

Exemple : convertir un entier décimal en base binaire : $(13)_{10} = (1011)_2$




```
void conversion (int  x){
  if( x != 0){
    conversion (x/2) ;
    printf("%d", x%2) ;
   }
}
```



Conversion d'un entier en chaîne de caractères :

Ver1

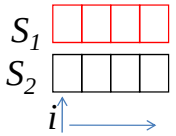
```
void EntCh (int  x, char T[], int* i){
  if( x != 0){
    EntCh (x/10, T, i) ;
    T[*i] = x%10 + '0' ;
    (*i)++;
    T[*i] = '\0' ;
  }
}
```

Ver2

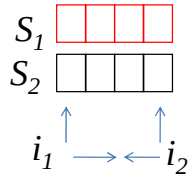
```
int EntCh (int  x, char T[], int i){
  if( x != 0){
    i = EntCh (x/10, T, i) ;
    T[i] = x%10 + '0' ;
    i++;
    T[i] = '\0' ;
  }
  return i ; }
```

Conversion d'une chaîne de caractères/entier en entier :

```
int ChEnt (char T[]){
  int i = 0, x = 0 ;
  while(T[i] != '\0'){
    x = x*10 + T[i] – '0' ;
    i++;
  }
  return x ; }
```

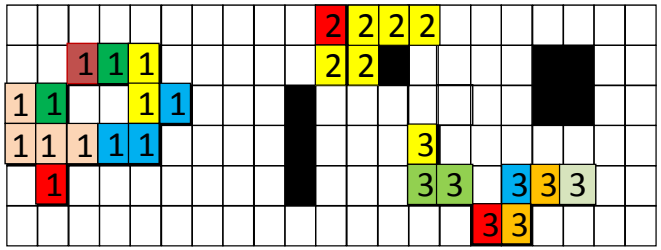
$$S_1 == S_2 \ ? \begin{cases} 0 & si\ l_1 \neq l_2 \\ 1 & sinon\ si\ i = l_1 \\ 0 & sinon\ si\ S_{1i} \neq S_{2i} \\ S_1 == S_2\ ?\ (\text{-1 case}) & sinon \end{cases}$$



Remarques:

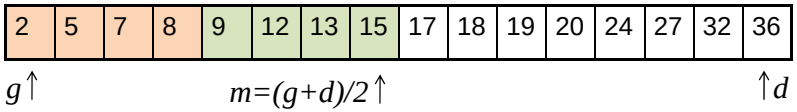
- Les programmes itératifs sont souvent plus efficaces,
- mais les programmes récursifs sont plus faciles à écrire.

Coloriage d'objets



```
void coloriageObjet( int  i , int  j , int  c ){
  if( i>=0  &&  i<N  &&  j>=0  &&  j<M  &&  T[i][j]==1 ){
     T[i][j] = c;
     for( k=i–1; k<= i+1; k++ )
       for( l=j–1; l<= j+1; l++ )
         if( k>=0  &&  k<N  &&  l>=0  &&  l<M  &&  T[k][l]==1 )
            coloriageObjet( k , l , c );
  }
}
```

Recherche dichotomique




```
int rechDicho (int x, int g, int d){
  int m ;
  if( g > d)  return –1;
  m = (g + d)/2;
  if (x == t[m])  return m;
  else
    if(t[m] < a)
      return rechercheDicho(x, m+1, d);
    else
      return rechercheDicho(x, g, m-1);
}
```

```
while( g <= d){
  m = (g + d)/2;
  if (x == t[m])
    return m;
  else
    if(t[m] < a)
      g = m+1 ;
    else
      d = m – 1 ;
}
return –1 ;
```

Tri rapide (QuickSort)

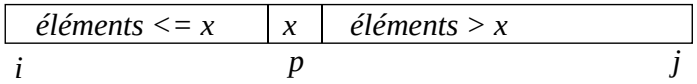
C'est un tri récursif comportant 3 étapes :

- Partition autour du pivot
  - Un élément pivot du tableau à trier est choisi
  - Le tableau est réorganisé afin que:
    - les éléments <= au pivot se retrouvent avant le pivot
    - Les éléments > pivot se retrouvent après le pivot

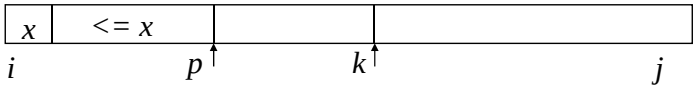
    Conséquence de cette étape : l'élément pivot est à sa position finale (lorsque le tableau est complètement trié)
- Un appel récursif pour trier les éléments avant le pivot
- Un appel récursif pour trier les éléments après le pivot

Fonctionnement de la fonction pivot

Objectif : x valeur du pivot p



À une certaine étape du processus

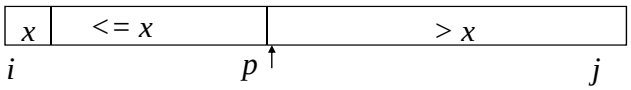


p pointe sur le dernier élément connu pour lequel T[p] >= x .

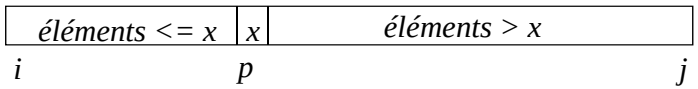
A sa droite, les éléments sont > x jusqu'à k qui pointe sur le premier élément non encore traité.

```
si T[k] <= x
        p ← p+1
        échanger(T[k], T[p])
fin-si
```


Après avoir parcouru le tableau



On échangea T[i] et T[p]




```
void  trier(int  t[ ], int  min, int  max){
    int  p, i;
    if(min<=max){
      p = min;
      for(i=p+1; i<=max; i++)
        if(t[i]<=t[min]){
          p++;
          permuter(t, i, p);
        }
      permuter(t, min, p);
      trier(t, min, p–1);
      trier(t, p+1, max);
    }
}
```

Exemples :

$$x^k = \begin{cases} 1 & si\ k = 0 \\ x * x^{k-1/2} * x^{k-1/2} & si\ k\ impair \\ x^{k/2} * x^{k/2} & si\ k\ pair \end{cases}$$

$$F(n) = \begin{cases} 0 & si\ n = 0 \\ 1 & si\ n = 1 \\ F(n-F(n-1))+F(n-1-F(n-2)) & sinon \end{cases}$$

Tri Fusion

tri : 69 85 34 24 40 77 64
tri : 69 85 34 24
tri : 69 85
tri : 69
tri : 85
fusion : 69 et 85 → 69 85
tri : 34 24
tri : 34
tri : 24
fusion : 34 et 24 → 24 34
fusion : 69 85 et 24 34 → 24 34 69 85
tri : 40 77 64
tri : 40 77
tri : 40
tri : 77
fusion : 40 et 77 → 40 77
tri : 64
fusion : 64 → 64
fusion : 40 77 et 64 → 40 64 77
fusion : 24 34 69 85 et 40 64 77 → 24 34 40 64 69 77 85