**Whether Linked List is linear or Non-linear data structure?**

According to Access strategies Linked list is a linear one. According to Storage Linked List is a Non-linear one

**What is the quickest sorting method to use?**

The Quick Sort,

The Merge Sort

The Radix Sort

**How can I search for data in a linked list?**

**What is the heap?**

The heap is where malloc(), calloc(), and realloc()get memory.

**How can I search for data in a linked list?**

**What is the difference between ARRAY and STACK?**

STACK follows LIFO. Thus the item that is first entered would be the last removed.

In array the items can be entered or removed in anyorder. Basically each member access is done using index. No strict order is to be followed here to remove a particularelement.

PILE suit LIFO. Ainsi, l'article qui est d'abord entré serait la dernière enlevé.   
Dans le tableau des éléments peuvent être saisis ou retirés dans AnyOrder. Fondamentalement chaque accès de membre se fait en utilisant l'index. Aucun ordre strict est à suivre ici pour supprimer un particularelement.

**What method removes the value from the top of a stack?**

The pop() member method removes the value from the top of a stack, which is then returned by the pop()member method to the statement that calls the pop() member method.

**What is a queue ?**

A Queue is a sequential organization of data. A queue is a first in first out type of data structure. An element is inserted at the last position and an element is always taken out from the first position

**Which process places data at the back of the queue?**

Enqueue is the process that places data at the backof the queue.

**What is Linked List ?**

Linked List is one of the fundamental data structures. It consists of a sequence of? nodes, each containing arbitrary data

fields and one or two (”links”) pointing to the next and/or previous nodes. A linked list is a self-referential datatype because it contains a pointer or link to another data of the same type. Linked lists permit insertion and removalof nodes at any point in the list in constant time, but do not allow random access

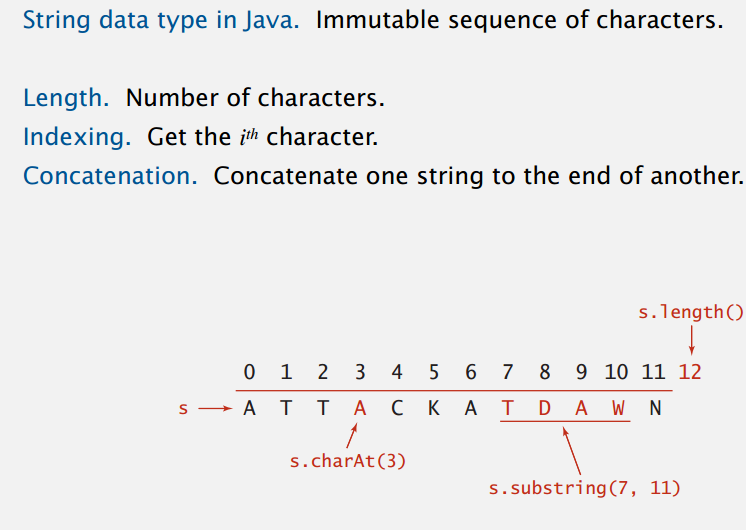
Difference between calloc and malloc ?

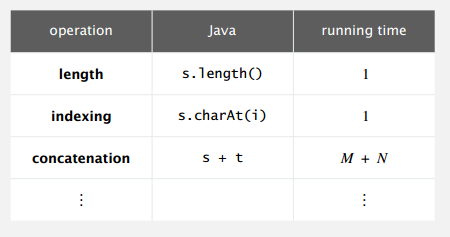
malloc: allocate n bytes

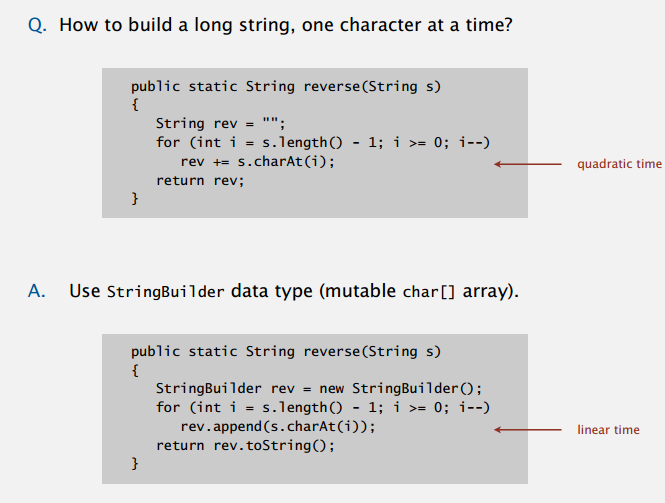
calloc: allocate m times n bytes initialized to 0

**What are the different storage classes in C?**

C has three types of storage: automatic, static andallocated.





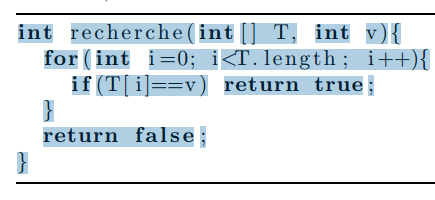


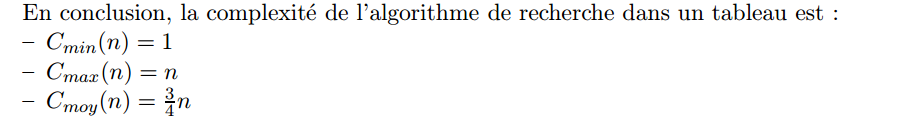
17) How is memory allocated by new ?

a)††† In a heap b) in a stack b) both a & b c) None of these.

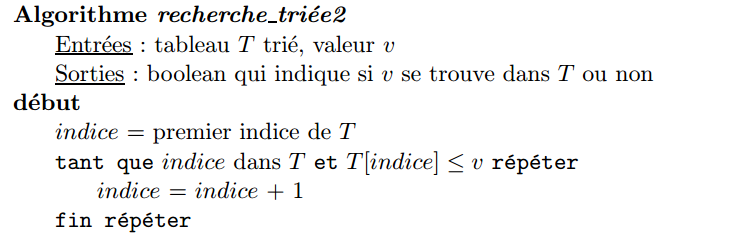
Voici par exemple la recherche dans un tableau

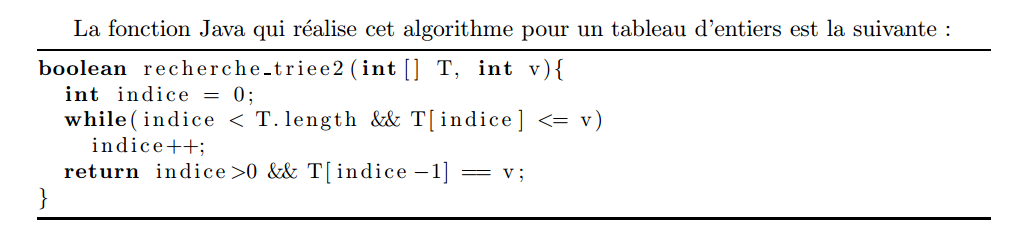
d’entiers, en Java :



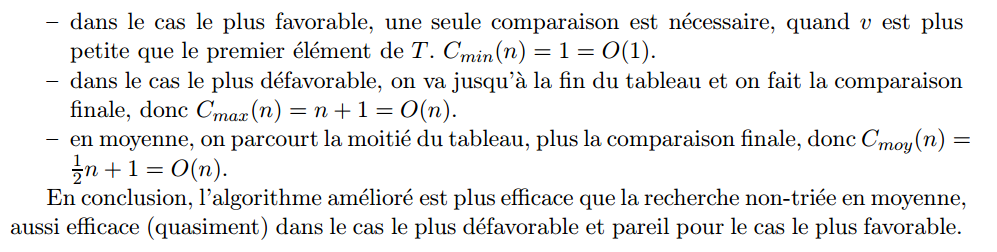


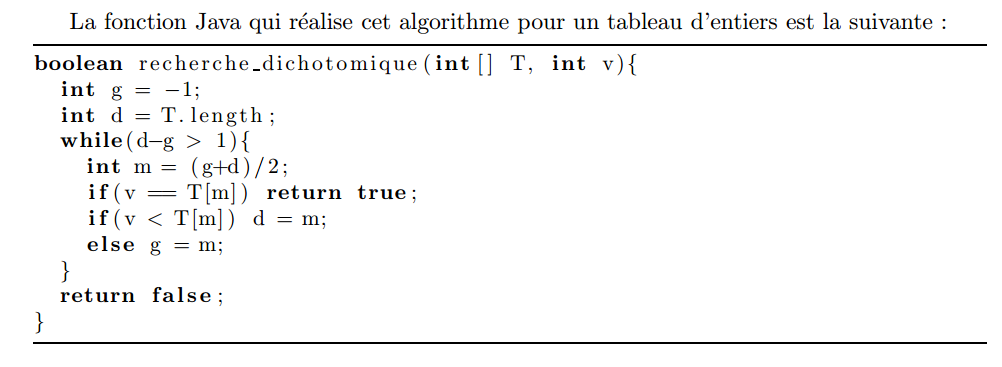
15.2.2 Recherche dichotomique dans un tableau tri´e

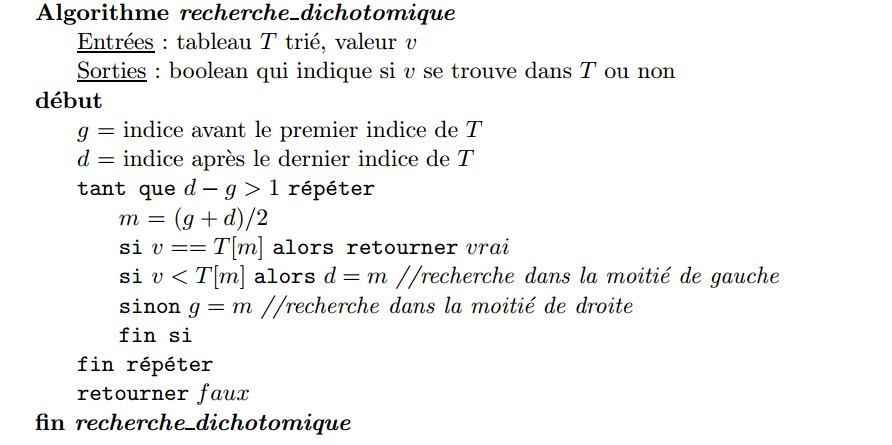


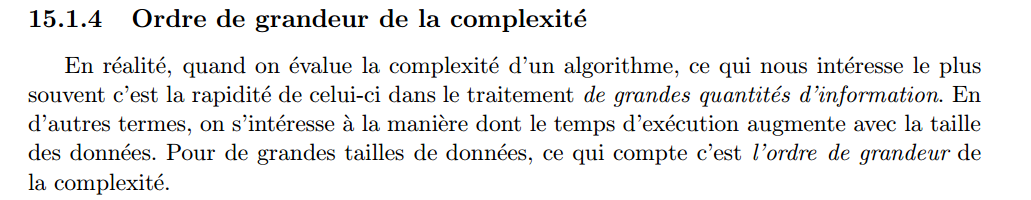


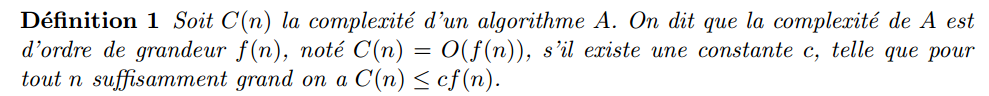
Complexit´e de l’algorithme am´elior´e

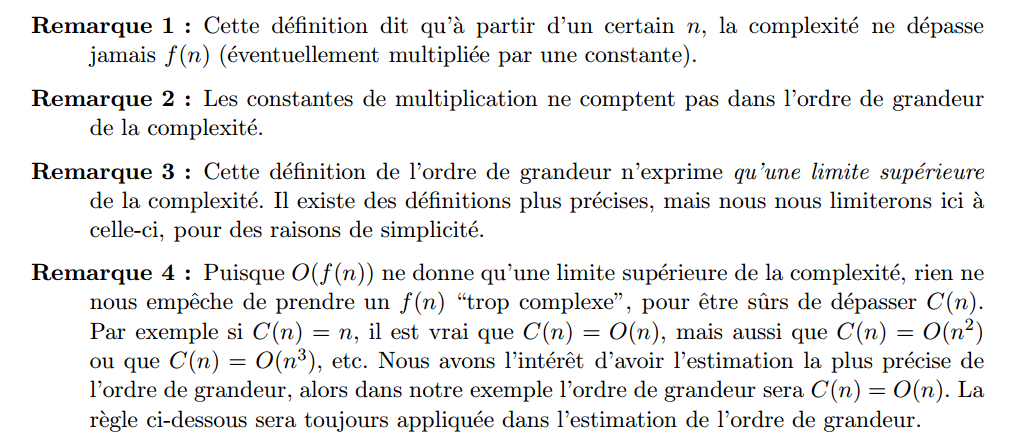


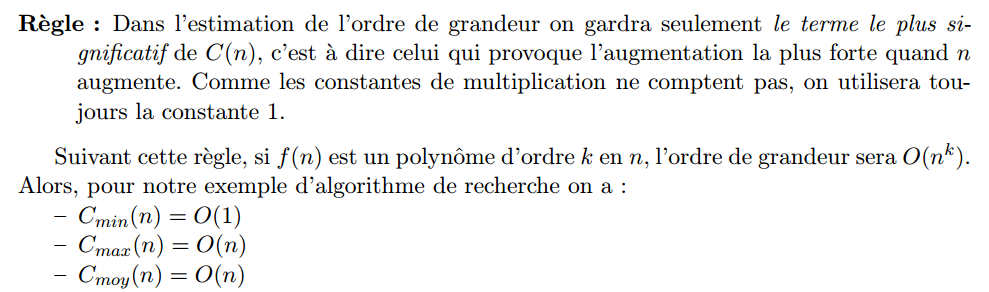


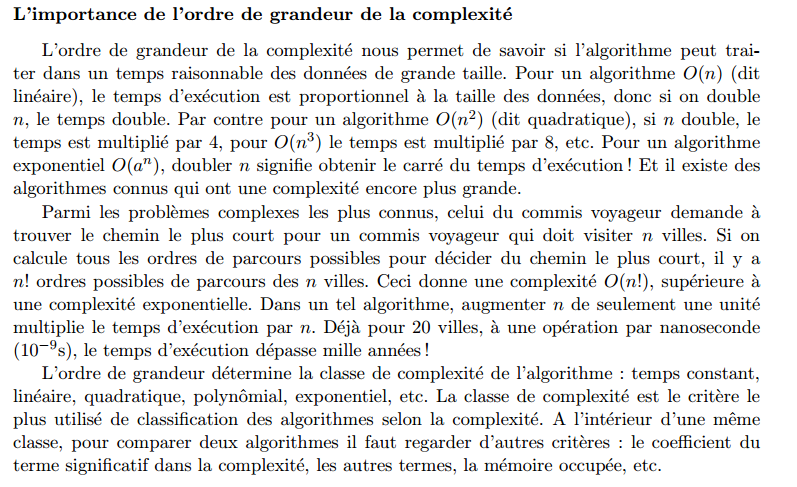












**Types de problems**

* Tris d'éléments d'une liste
* Recherche d'un élément
* Calcul sur des chaînes (caractères, nombres, bits, ...)‏
* Problèmes de graphes
* Problèmes combinatoires
* Problèmes géométriques
* Problèmes numériques
* Algorithmes exacts / d'approximation

**Analyse d'un algorithme**

* Complexité: mesure de son efficacité
  + Taille mémoire nécessaire à son exécution
  + Temps d'exécution nécessaire
    - dans le meilleur des cas
    - dans le pire des cas
    - en moyenne

Exemple: recherche d'un élément dans une liste?

Exemple: recherche du plus grand élément d'une liste?

**Efficacité d'un algorithme**

* Temps d'exécution
  + fonction de la taille des données en entrée

choix du bon paramètre

* + - taille d'une liste, degré d'un polynôme
    - taille d'une matrice?
    - nombre de noeuds, profondeur, largeur d'un graphe?
    - nombre de mots d'un fichier ou nombre de caractères?
  + fonction du nombre de fois où une opération de base est répétée dans l'algorithme

**Efficacité d'un algorithme**

* Notations asymptotiques:
  + O(*g(n)*)
  + Ω(*g(n)*)

θ(*g(n)*)

**Conception d'un algorithme**

* Stratégie de résolution d'un problème
* Approche itérative
  + répéter jusqu'à obtention du résultat souhaité
* Approche récursive
  + diviser pour régner
* **Définition 1:** la complexité temporelle d’un algorithme est le temps mis par ce dernier pour transformer les données du problème considéré en un ensemble de résultats.
* **Définition 2**: la complexité spatiale d’un algorithme est l’espace utilisé par ce dernier pour transformer les données du problème considéré en un ensemble de résultats.
* **Facteurs affectant le temps d’exécution**:
* 1. machine,
* 2. langage,
* 3. programmeur,
* 4. compilateur,
* 5. algorithme et structure de données.
* Le temps d’exécution dépend de la longueur de l’entrée.
* Ce temps est une fonction **T**(*n*) où *n* est la longueur des données d’entrée.

**La méthode empirique**

Problème

Ces résultats dépendent

* la machine utilisée;
* jeu d’instructions utilisées
* l’habileté du programmeur
* jeu de données générées
* compilateur choisi
* l’environnement dans lequel est exécuté les deux algorithmes (partagés ou non)
* .... etc.

**Méthode mathématique**

Notation grand-O

**Définition:** Soit **T**(*n*) une fonction non négative. **T**(*n*) est en O(*f*(*n*)) s’il existe deux constante positives *c* et *n*0 telles que:

**T**(*n*) <= *cf*(*n*) pour tout *n* >= *n*0.

**Utilité**: Le temps d’exécution est borné

**Signification**: Pour toutes les grandes entrées (i.e., *n* >= *n*0), on est assuré que l’algorithme ne prend pas plus de *cf*(*n*) étapes.

Þ Borne supérieure.

Grand-O: Exemples

**Exemple 1**: Initialiser un tableau d’entiers

*for (int i=0; i<n; i++) Tab[i]=0;*

Il y a n itérations

Chaque itération nécessite un temps <= c,

où c est une constante (accès au tableau + une affectation).

Le temps est donc T(n) <= cn

Donc T(n) = O(n)

**Exemple 2:** **T**(*n*) = *c*1*n*2 + *c2n* .

*c*1n2 + *c*2*n* <= *c*1*n*2 + *c*2*n*2 = (*c*1 + *c*2)*n*2

pour tout *n* >= 1.

**T**(*n*) <= *cn*2 où *c* = *c*1 + *c*2 et *n*0 = 1.

Donc, **T**(*n*) est en O(*n*2).

**Exemple 3:** T(*n*) = *c*. On écrit T(n) = O(1).

Grand-Omega, La notation Theta

Règles de simplification 2

Si

*f*(*n*) = O(*kg*(*n*))

où *k* > 0 est une constante,

alors

*f*(*n*) = O(*g*(*n*)).

Si

*f*1(*n*) = O(*g*1(*n*))

et

*f*2(*n*) = O(*g*2(*n*)),

alors

(*f*1 + *f*2)(*n*) = O(max(*g*1(*n*), *g*2(*n*)))

* **Classes de complexité**

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité :

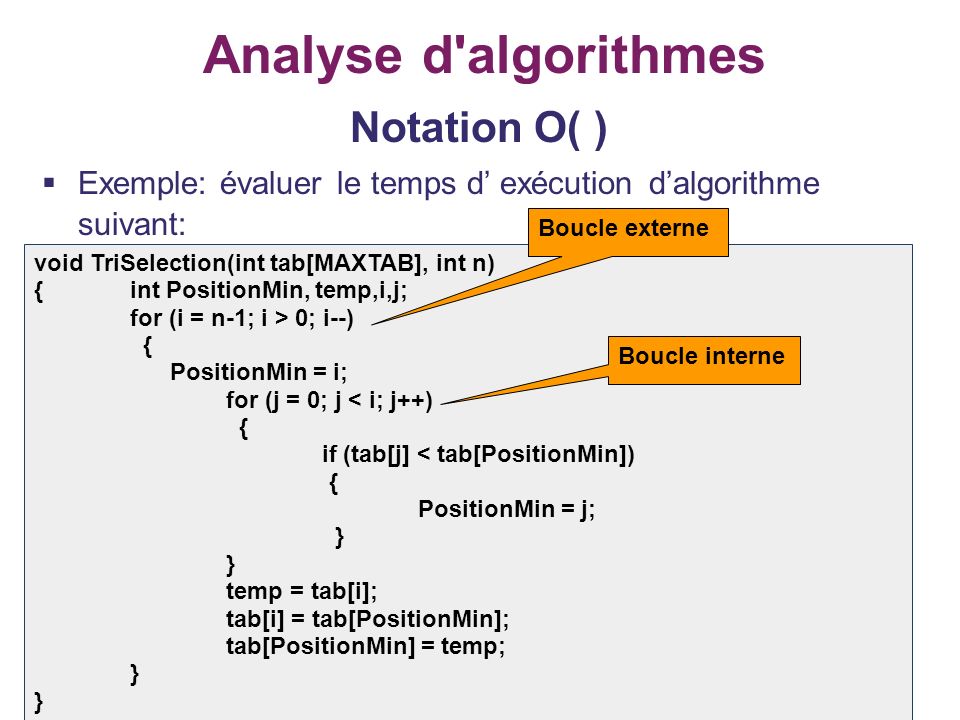
– Les algorithmes ***sub-linéaires*** dont la complexité est en général en *O(logn).*

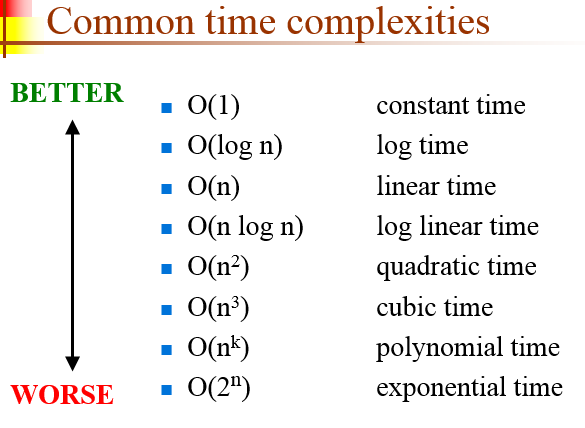
– Les algorithmes ***linéaires*** en complexité *O(n)* et ceux en complexité en *O(nlogn)* sont considérés comme rapides.

– Les algorithmes ***polynomiaux*** en *O*(*nk*) pour *k* > 3 sont considérés comme lents, sans parler des algorithmes ***exponentiels*** (dont la complexité est supérieure à tout polynôme en *n*) que l’on s’accorde à dire impraticables dès que la taille des données est supérieure à quelques dizaines d’unités.

Indécidabilité

* Les limites des ordinateurs telles qu’on les comprend au début du XXe siècle.
* Turing (et d’autres) formalisent la notion de calcul automatique et d’algorithme et démontrent qu’il existe des *langages* *indécidables* c’est-à-dire des problèmes de calcul qui ne peuvent être résolus par aucun algorithme.





i = 0;

while (i < MAX && this\_array[i] != target)

i = i + 1;

if (i <MAX)

printf ( “Yes, target is there \n” );

else

printf( “No, target isn’t there \n” );

no. of operations: 1 (best case)

n (worst case)

n/2 (average case)

**What is a problem?**

* Definition
  + A mapping/relation between a set of input instances (domain) and an output set (range)
* Problem Specification
  + Specify what a typical input instance is
  + Specify what the output should be in terms of the input instance
* Example: Sorting
  + **Input**: A sequence of N numbers a1…an
  + **Output**: the permutation (reordering) of the input sequence such that a1 ≤ a2 ≤ … ≤ an .

Types of Problems

**Search**: find X in the input satisfying property Y

**Structuring**: Transform input X to satisfy property Y

**Construction**: Build X satisfying Y

**Optimization**: Find the best X satisfying property Y

**Decision**: Does X satisfy Y?

**Adaptive**: Maintain property Y over time.

**Two desired properties of algorithms**

* Correctness
  + Always provides correct output when presented with legal input
* Efficiency
  + What does efficiency mean?

Algorithm Analysis Overview

* RAM model of computation
* Concept of input size
* Three complexity measures
  + Best-case, average-case, worst-case
* Asymptotic analysis
  + Asymptotic notation

Example Problems

1. What does it mean if:

f(n) **≠** O(g(n)) *and* g(n) **≠** O(f(n)) ???

2. Is 2n+1 = O(2n) ?

Is 22n = O(2n) ?

3. Does f(n) = O(f(n)) ?

4. If f(n) = O(g(n)) and g(n) = O(h(n)),

can we say f(n) = O(h(n)) ?

**java vcteur**

import java.util.Scanner;

/\* Declaration et creation du tableau \*/

int[] tab;

tab = new int[taille];

/\* Pour chaque case, on demande a l'utilisateur de saisir la valeur \*/

for(int i = 0; i < taille; i++) {

tab[i] = in.nextInt();

}

for(int i = 0; i < taille; i++) {

tab[i] = aleatoire(min, max);

}

/\* 4. Sous-programme (ou fonction) retournant la valeur maximum d'un tableau \*/

for(int i = 1; i < tab.length; i++) {

/\* Si la valeur de la case lue est plus grande que la valeur maximum courante \*/

if(tab[i] > max)

/\* On met a jour la valeur maximum \*/

max = tab[i];

}

/\* 5. Sous-programme (ou fonction) retournant la \*position\* de la valeur maximum d'un tableau \*/

/\* 6. Sous-programme (ou fonction) retournant la \*position\* de la valeur minimum d'un tableau \*/

/\* 8. Sous-programme retournant la moyenne d'un tableau d'entiers \*/

/\* 9. Sous-programme recherchant la presence d'un entier dans un tableau \*/

/\* 10. Sous-programme inversant les elements d'un tableau sans utiliser de tableau intermediaire \*/

/\* 12. Sous-programme recherchant une valeur entiere dans un tableau \*suppose trie\* \*/

boolean trouve = false;

int i = 0;

/\* Le tableau etant trie, on ne recherche que dans les cases dont la valeur

\* est plus petite que l'entier recherche \*/

while(i < tab.length && tab[i] <= recherche && trouve == false) {

if(tab[i] == recherche) {

trouve = true;

}

else {

i++;

}

}

public class Dichotomie

{

// Fonction de recherche dichotomique dans un tableau tri'e.

// Retourne un indexe i tel que t[i] == val si val est dans t,

// retourne un indexe i tel que t[i] <= val <= t[i+1] si possible,

// retourne -1 sinon.

static int recherche\_rec (String[] t, String val, int g, int d)

{

if (g >= d)

{

if (val.compareTo (t[g]) == 0) return g ;

if (g > 0 && val.compareTo (t[g-1]) == 0) return g - 1 ;

return -1 ;

}

// Ici, d > g

int m = (d + g) / 2 ;

if (val.compareTo (t[m]) <= 0) return recherche\_rec (t, val, g, m) ;

return recherche\_rec (t, val, m+1, d) ;

}

static int recherche (String[] t, String val)

{

return recherche\_rec (t, val, 0, t.length - 1) ;

}

public static void main (String args[])

{

if (args.length != 1)

System.out.println ("Usage: java Dichotomie nom\_poisson") ;

else {

int i = recherche (Machines.poissons, args[0]) ;

System.out.println ("\nslogin " + Machines.departements[i] + "\n") ;

}

}

}

**// dichotomique search**

static int dichoSearch(String table[], String cle) {

int deb,fin,milieu;

deb = 0;

fin = table.length-1;

do {

milieu = (deb + fin) / 2;

int comp = table[milieu].compareTo(cle);

if (comp == 0) return milieu;

if (comp > 0) fin = milieu - 1;

else deb = milieu + 1;

} while (deb <= fin);

return -1;

}

**I.2. Implémentation d’un algorithme**

L’implémentation d’un algorithme consiste à le traduire dans un langage de programmation et ceci dans le but de l’exécuter sur ordinateur. L’implémentation nécessite le choix de la représentation des données ou la **structure de données**. La résolution d’un problème peut être souvent effectuée par plusieurs algorithmes et pour chaque algorithme il existe plusieurs structures de données. L’algorithme ainsi que la structure de donnée associée à son implémentation est caractérisé par :

* Sa **simplicité** : un algorithme simple est facile à comprendre, à implémenter et généralement à prouver.
* Son **temps d’exécution** (complexité en temps ou complexité temporelle) : On préfère un algorithme qui s’exécute en une seconde sur un ordinateur de bureau à un autre qui prend une heure sur la même machine. De même, on rejette tout algorithme dont le temps d’exécution dépasse un siècle !!
* Son **requis mémoire** (complexité en mémoire ou complexité spatiale) : cette mémoire dépend à la fois de l’algorithme et des structures des données choisies. Le requis mémoire ne dois jamais dépasser une certaine limite qui dépend de la machine utilisée. Ceci a pour conséquence de mettre une limite sur la taille des problèmes pouvant être résolus.
* **COMPLEXITE DES ALGORITHMES**

L’étude de la complexité des algorithmes a pour objectif l’estimation du coût d’un algorithme (assorti d’une structure de donnée). Cette mesure permet donc la comparaison de deux algorithmes sans avoir à les programmer.

Si l’on prend en compte pour l’estimation de la complexité les ressources de la machine telles que la fréquence d’horloge, le nombre de processeurs, le temps d’accès disque etc., on se rend compte immédiatement de la complication voir l’impossibilité d’une telle tâche. Pour cela, on se contente souvent d’estimer la **relation entre la taille des données et le temps d’exécution**, et ceci **indépendamment de l’architecture utilisée**.

Il s’agit d’un modèle simplifié qui tient compte des ressources technologiques ainsi que leurs coûts associés.

Dans ce modèle, on appelle **opérations élémentaires** les opérations suivantes :

* un accès mémoire pour lire ou écrire la valeur d’une variable ou d’une case d’un tableau ;
* une opération arithmétique entre entiers ou réels telle que l’addition, soustraction, multiplication, division ou calcul du reste d’une division entière ;
* une comparaison entre deux entiers ou réels.

**Exemple :**

L’instruction « c ← a + b ; » nécessite les quatre opérations élémentaires suivantes :

1. un accès mémoire pour la lecture de la valeur de a,
2. un accès mémoire pour la lecture de la valeur de b,
3. une addition de a et b,
4. un accès mémoire pour l’écriture de la nouvelle valeur de c.

*On définit ainsi la complexité (temporelle) d’un algorithme comme étant la mesure du nombre d’opérations élémentaires qu’il effectue sur un jeu de données. La complexité est exprimée comme une fonction de la taille du jeu de données.*

**II.1. Mesure de la complexité**

Les trois notations précédentes peuvent être récapitulées par le schéma suivant :

Ө

Borne **S**upérieure et **I**nférieure

**L**arge

O

Borne **S**upérieure **L**arge

Ω

Borne **I**nférieure rge

**- Notation O (grand o) :**  Lorsque la fonction f est bornée **uniquement supérieurement** par la fonction g on utilise la notation **O**. **O(g(n))** désigne donc l’ensemble des fonctions positives de la variable n, pour lesquelles il existe une constante c et un entier n0, satisfaisant la relation :

**0 ≤ f(n) ≤ c g(n) ∀ n≥n0**

La relation f(n) =O(g(n)) indique que la fonction f est bornée supérieurement par la fonction g pour des valeurs suffisamment grandes de l’argument n. Donc f(n) =Θ(g(n)) implique que f(n) =O(g(n)) (par abus de notation). Formellement, on peut écrire Θ(g(n)) ⊆ O(g(n)).

**Exemple** : 100n2 + 10n = **O**(n2) mais on peut aussi écrire **100n = O(n2) !** Car ceci est équivalent à dire que 100n est asymptotiquement bornée supérieurement par n2.

**II.2. Mise en œuvre du calcul de la complexité**

**Exemple d’illustration :** Considérons l’algorithme de tri par insertion discuté précédemment permettant de trier un tableau A de taille N :

**void tri\_inserer (int A[50] , int N) // coût Nombre de fois**

**{ // ==========================**

**int j, temp, i;**

**for (j=2 ; j<=N ; j++) // coût C1 ........ N**

**{**

**temp = A[j]; // coût C2 ........ N-1**

**i = j-1; // coût C3 ........ N-1**

**while (i>0 && A[i]>temp) // coût C4 ........ 2+3+...+N**

**{**

**A[i+1]=A[i]; // coût C5 ........ 1+2+...N-1**

**i=i-1; // coût C6 ........ 1+2+...N-1**

**}**

**A[i+1]=temp; // coût C7 ........ N-1**

**}**

**}**