Ecole Normale Supérieure de l'Enseignement Technique de Mohammedia
Université Hassan II de Casablanca

ENSET

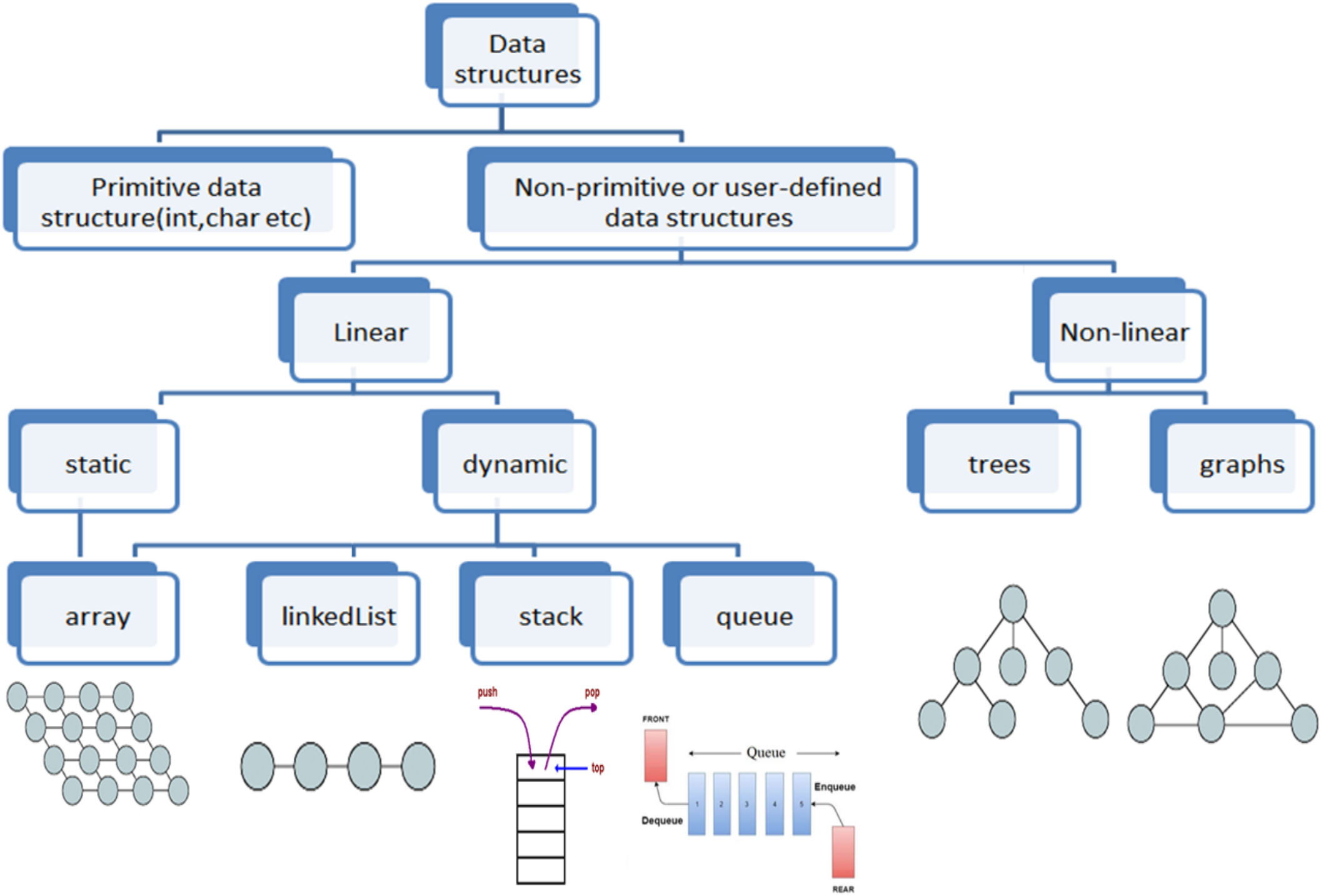
المدرسة العليا لأساتذة التعليم التقني
المحمدية
جامعة الحسن الثاني بالدار البيضاء

Département Mathématiques & Informatique

Cours structures de données & programmation fonctionnelle

Cours TD TP

Exemples de structures de données



Chapitre I :

Analyse de la complexité des algorithmes

Objectifs :

☞ **Etudier la complexité des algorithmes : Mesurer les ressources (temps, mémoire) utilisées par un programme, ou nécessaires à la résolution d'un problème**

- **Evaluation théorique et calcul asymptotique**
- **Mesure expérimentale**

Sommaire

Chap I : Analyse de la complexité des algorithmes

- Introduction - Exemple de problème
- Méthodes de calcul
  - Méthode théorique
  - Méthode expérimentale
- Complexités représentatives – cas Meilleur, Cas Pire, Moyenne
- Notation de Landau – Analyse asymptotique
- Classes de complexité
- Applications

I. Introduction

1. Position du problème

Question

soient A1 et A2 deux algorithmes pour le même problème.

Comment faire le choix de l'une des Solutions A1 et A2?

Réponse

Mesurer le temps T1(A1) et le temps T2(A2) pour les mêmes **données** et pour **différentes configurations** et choisir le plus rapide.

Question

Comment Mesurer le temps d'un algorithme A ?

Réponse

Procéder selon une **méthode de mesure**

Remarques :

On distingue aussi la notion de **complexité spatiale**. Elle correspond à la capacité mémoire exigée par une solution algorithmique.

Cette complexité est moins importante que la **complexité temporelle**. La suite du cours sera consacrée à cette dernière. En effet la complexité temporel reflète indirectement la complexité spatiale

2. Méthodes de mesure de la complexité Temporelle

Pour évaluer la complexité temporelle d'un algorithme A on distingue deux démarches possibles :

- **Méthode expérimentale (Empirique)**
- **Méthode Théorique (Mathématique)**

II. Méthode expérimentale

1. Principe

Cette méthode consiste à :

- Programmer l'algorithme dans un langage (choix du langage ?)
- Exécuter le programme pour des tailles de données différentes
- évaluer le temps pour chaque exécution.

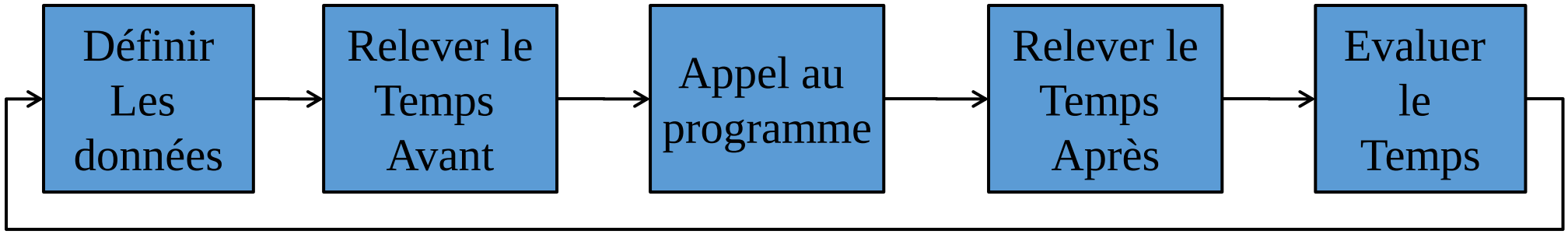
2. Inconvénients

cette méthode très limitée pour les raisons suivantes :

- Nécessite de programmer l'algorithme
- La mesure dépend de la **machine utilisée**, de **l'environnement d'exécution,** de **l'environnement de programmation** choisi et du **jeu de données** utiliser pour faire la mesure.

3. Démarche

Utiliser les fonctions de relevé du temps système avant et après le programme à mesurer.



4. Outils de travail

Outils C / C++

void gettime(struct time *ptemp) // déclarée dans dos.h

struct time{ unsigned char ti_min,ti_hour,ti_hand,ti_sec;};

Ou bien

time_t time (time_t * timer);// envois le temps courant (time.h)

clock_t clock() //renvoi la durée d'exécution d'un prog (time.h)

CLK_TCK |CLOCKS_PER_SEC//le nombre de tops d'horloge par seconde

java

- Classe **Date** appartenant au package java.util

```
Date d1= new Date();
//appel au programme
…
Date d2=new Date();
d2.getTime()-d1.getTime())/1000.0
```

//getTime() renvoi le temps en millisecondes depuis 1/1/70 à 00:00:00

- Classe **Calendar** appartenant au package java.util

```
Calendar c1= Calendar.getInstance();
//appel au programme
…
 Calendar c2= Calendar.getInstance();
c2.getTimeInMillis()-c1.getTimeInMillis())/1000.0
```

//getTimeInMillis() renvoi le temps en millisecondes depuis 1/1/70

5. Exercices

Exercice1

Mesurer le temps nécessaire pour calculer X^N pour les deux schémas de calculs suivants :

- Schéma 1 :

  $P_0 = 1$;

  $P_i = P_{i-1} * x$

- Schéma2 :

  $P_0 = 1$;

  $P_i = P_i/2 * P_i/2$ si i est pair;

  $P_i = P_i/2 * P_i/2 * x$ si non

Exercice2

Mesurer le temps nécessaire pour calculer le produit de deux matrice carrées de taille N

Correction de l'exercice1

```
import java.util.Calendar;
public class MesurePuissance{
    public static double puis2(double x, long n){
        if(n==0)return 1;
        double pdemi=puis2(x,n/2);
        if(n%2==0)return pdemi*pdemi;
        return pdemi*pdemi*x;
    }

    public static double puis1(double x, long n){
        double p=1;     long i;
        for(i=1;i<=n;i++) p*=x;
        return p;
    }

public static String toString(double d){
        final long nbSecJour=24*3600L;
        String s =new String();
            if(d>=nbSecJour){
              s=(long)d/nbSecJour + "Jours ";
              d=d-((long)d/nbSecJour)*nbSecJour;
            }
            if(d>=3600){
              s=s+(long)d/3600 +" Heures ";
              d=d-((long)d/3600)*3600;}
            if(d>=60){
              s=s+(long)d/60 +" Minutes ";
              d=d-((long)d/60)*60;}
          d=(long)(d*1000);d=d/1000; s=s+d+" Secondes" ;
          return s;
    }

    public static void main(String[] args){
        double x=1.0;  long n=100000000000L;
        Calendar c=Calendar.getInstance();
        long t1=c.getTimeInMillis();
        puis1(x,n);
        c=Calendar.getInstance();
        long t2=c.getTimeInMillis();
        System.out.println("le temps de calcul de la
puissance pour x="+x +" et n="+n +" : ");
        System.out.println("puis1 : " +toString((t2-
t1)/1000.0));
        c=Calendar.getInstance();t1=c.getTimeInMillis();
        puis2(x,n);
        c=Calendar.getInstance();t2=c.getTimeInMillis();
        System.out.println("puis2 : " +toString((t2-
t1)/1000.0));
    }
}
```

Temps obtenus pour X^N pour N=100000000000.0

```
le temps de calcul de la puissance pour x=1.0 et n=100000000000 :
puis1 : 7 Minutes 31.994 Secondes
puis2 : 0.0 Secondes
```

Correction de l'exercice2

```
import java.util.Random;
public class ProdMat {
  public static void initMat(float[][] m){
     Random rnd=new Random();
     for(int i=0;i<m.length;i++)
        for(int j=0;j<m[0].length;j++)
           m[i][j]=rnd.nextFloat()*100;
  }

  public static String toString(float[][] m) {
     String s="";
     for(int i=0;i<m.length;i++){
        s+=""+m[i][0];
        for(int j=1;j<m[0].length;j++)
           s+="  ,  "+m[i][j];
        s+="\n";
     }
     return s;
  }

  public static float[][] prodMat(float[][] m1,float[][]
                                  m2)throws Exception{
     if(m1[0].length!=m2.length) throw new
           Exception("produit matriciel impossible");
     float[][] r=new float[m1.length][m2[0].length];
     for(int i=0;i<m1.length;i++)
        for(int j =0;j<m2[0].length;j++)
           for(int k=0;k<m1[0].length;k++)
              r[i][j]+=m1[i][k]*m2[k][j];
     return r;
  }
}
```

Temps obtenus

Pour le produit d'une matrice [3000][10] par une matrice [10][1000]

```
le temps de calcul du produit MAtricie :
0.165 Secondes
BUILD SUCCESSFUL (total time: 25 seconds)
```

6. Application

Ecrire une application programmée en C/C++ pour comparer pour des tailles différentes les méthodes de tri suivantes :

- Tri bulle
- Tri par insertion
- Tri par sélection
- Tri par fusion
- Tri rapide
- (Tri par tas)

L'application doit représenter les résultats des mesures sous forme graphique.

III. Méthode théorique

1. Principe

Consiste à évaluer la complexité d'un algorithme par le **décompte** des **instructions de base** de l'algorithme et non pas par sa mesure en unité de temps (par exemple les secondes).

Cette méthode consiste à :

- Déterminer une grandeur entière **N** qui quantifie la **taille des données de l'algorithme,**
- Déterminer les **opérations fondamentales** qui constituent les **opérations déterminantes (à compter)** pour l'algorithme,
- Faire le **décompte** des opérations fondamentales en fonction de N en suivant des **règles de décomptage**.

Le résultat constitue alors la **complexité théorique notée T(N).**

1. Principe - Exemples

- déterminer une grandeur entière **N** qui quantifie la **taille des données de l'algorithme**

Algorithme	Taille N

- Déterminer les **opérations fondamentales** qui constituent des **opérations déterminantes** pour l'algorithme

Algorithme	Oper. Fondamentale(s)

2. Règles de calcul de T(N)

Pour déterminer T(N) on s'appuie sur les règles suivantes :

- **Rg1 : Opération élémentaire**

  T(opération fondamentale) = constant

- **Rg2 : Séquence d'opérations**

  $\boldsymbol{T(seq) = \sum_{oper \in Seq} T(oper)}$

  la complexité d'une suite est la somme des complexités de ses actions.

- **Rg3 : conditionnelle et sélective**

$$T(si\ C\ Alors\ A\ Sinon\ B) = T(C) + Max(T(A), T(B))$$

$$T(switch(S)\{case\ V_1 : A_1; case\ V_2 : A_2; \dots, case\ Vn : An; default : An_{+1}\}) = T(S) + \sum_{i=1..n} T(Vi) + Max_{i=1..n}(T(Ai))$$

- **Rg4 : Boucle for(E1,E2,E3)B;**

Soit $T_i(B)$ la complexité du bloc B de la boucle pour le $i^{ième}$ passage, la complexité de la boucle est donnée par :

$$T_{boucle} = T(E1) + \sum_{i=1}^{n} (T_i(B) + T(E2) + T(E3)) + T(E2)$$

Où n est le nombre de passages de la boucle

- **Rg5 : Fonction/procédure**

leur complexité est déterminée par **celle de leur corps**.

Dans le cas de la **récursivité**, le temps de calcul est exprimé comme une **relation de récurrence** (**à trouver et à résoudre**) de la forme :

$$\begin{cases} T(n) = f(T(k)), k < n \; et \; n > n_0 \\ T(n\ ) = C_{n_0}, n \leq n_0 \end{cases}$$

Où n et k sont les tailles pour les appels récursifs successifs. N<=n0 étant les tailles pour l'arrêt des appels récursifs.

Exemple : Calcul de la Puissance

Solution 1 :

```
double puis1(double x, long n){
    double p=1; long i;
    for(i=1;i<=n;i++)
        p*=x;
    return p;
}
```

Solution 2 :

```
double puis2(double x, long n){
    if(n==0)return 1;
    double pdemi=puis2(x,n/2);
    if(n%2==0)return pdemi*pdemi;
    return pdemi*pdemi*x;
}
```

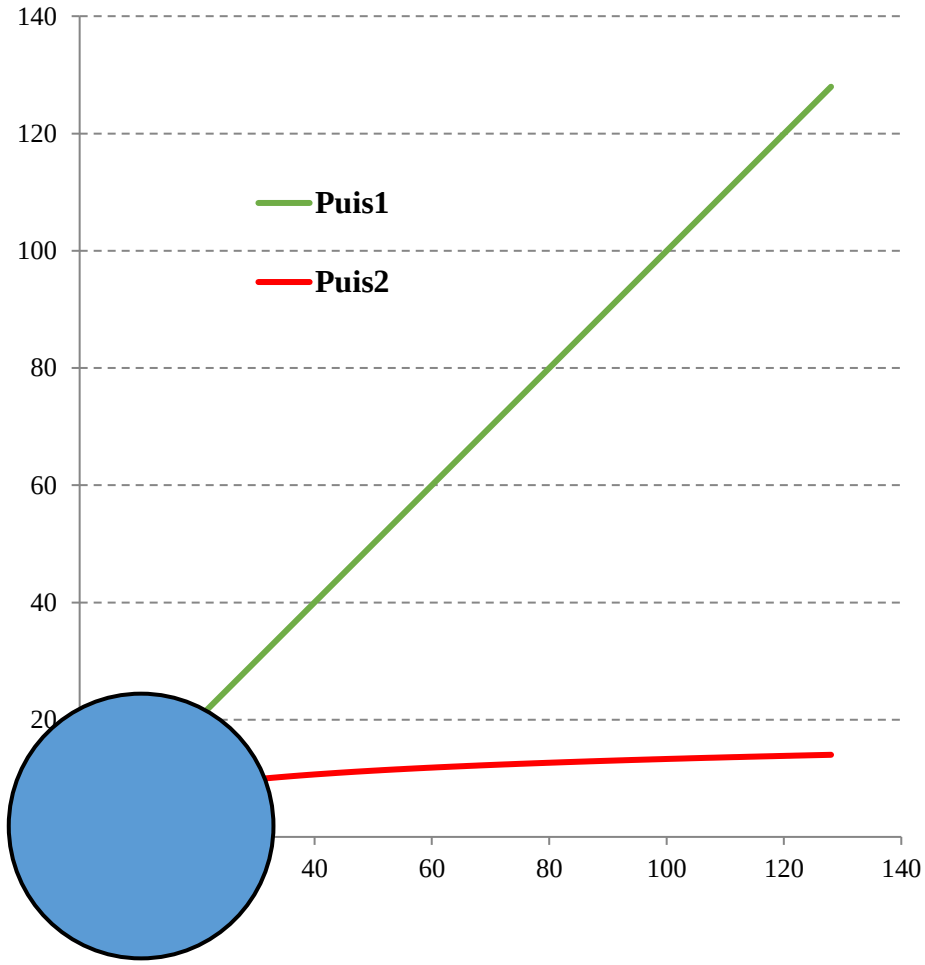
- Taille du problème ➔ Exposant n,
- Opération fondamentale ➔ *

$$T_1(n) = \sum_{i=1}^{n} 1 = n$$

$$\begin{cases} T_2(n) = T_2(n/2) + 2 \quad \forall\, n > 0 \\ T_2(0) = 0 \end{cases}$$

- **Résolution de l'équation de récurrence :**

$$\begin{cases} T_2(n) = T_2(n/2^1) + 2 \\ T_2(n/2^1) = T_2(n/2^2) + 2 \\ \ldots \\ T_2(n/2^p) = T_2(0) + 2 \\ T_2(0) = 0 \end{cases}$$



Où p est tel que $\mathbf{2^p = n \rightarrow p = \log_2 n}$

$$T_2(n) = \sum_{i=0}^{p} 2 = 2(p+1) = 2 + 2\log_2 n$$

Exercices

Exercice 1

1. Rappeler l'algorithme relatif au jeu de Hanoï qui consiste à déplacer N disques placés dans une tige par ordre décroissant du rayon du bas en haut, vers une 2ième tige en ayant la possibilité d'utiliser une seule tige intermédiaire. On ne pourra déplacer qu'une seul disque à la fois. On n'a pas le droit de poser un disque plus grand sur un disque plus petit. On suppose que l'opération de déplacement d'un disque est primitive et est déjà définie.
2. Evaluer sa complexité

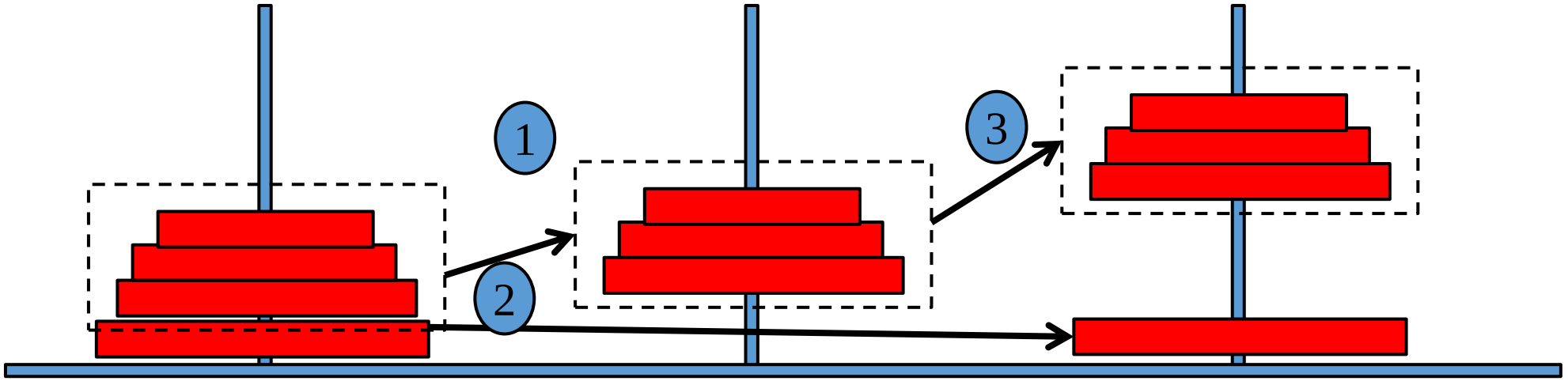
Exercice 2

1. Donner les solutions itérative et récursive qui calculent le nième terme de la suite de Fibonacci.
2. Evaluer leurs complexités.

Correction :

Exercice 1

Principe :



Solution

```
void Deplacer(int NbreDis, int NumTigSrc, int NumTigDest){
    if(NbreDis==1) DeplacerDisque(NumTigSrc,NumTigDesti);
    else{
            Deplacer(NbreDis-1,NumTigSrc,6-NumTigSrc-NumTigDest);
            DeplacerDisque(NumTigSrc,NumTigDest);
            Deplacer(NbreDis-1,6-NumTigSrc-NumTigDest,NumTigDesti);
    }
}
```

- Taille du problème ➔ Nombre de disques NbreDis noté N,
- Opération fondamentale ➔ DeplacerDisque(Tsrc,Tcibl)
- Le traitement est récursif ➔ Trouver une équation de récurrence :
  - Soit T(N) la complexité du traitement Deplacer(int N, …),

    ➔ T(N-1) est alors la complexité pour Deplacer(int N-1,…)

    ➔ d'où la relation : $\begin{cases} T(n) = 2*T(n-1)+1 \ \forall n>1 \\ T(1)=1 \end{cases}$

- **Résolution de l'équation de récurrence :**

$$\begin{cases} T(n) = 2*T(n-1)+1 \\ T(n-1) = 2*T(n-2)+1 \\ \dots \\ T(2) = 2*T(1)+1 \\ T(1)=1 \end{cases}$$

➔ Par substitution on obtient :

$$\begin{aligned} T(n) &= 2(2*T(n-2)+1)+1 \\ &= 2^2*T(n-2)+2^1+2^0 \\ &= 2^2*(2*T(n-3)+1)+2^1+2^0 \\ &= 2^3*T(n-3)+2^2+2^1+2^0 \\ &= 2^{n-1}*T(1)+2^{n-2}+\dots+2^1+2^0 \\ &= 2^{n-1}+2^{n-2}+\dots+2^1+2^0 \end{aligned}$$

➔ $T(n) = 2^n - 1$

3. Complexités représentatives

T(n) ne peut être déterminée parfaitement. En effet, pour le même algorithme, elle peut **changer en fonction de la configuration des données entrées.** Par conséquent, il est alors nécessaire de distinguer les trois **situations représentatives suivantes à trouver :**

- Complexité dans le **meilleur cas**
- Complexité dans le **prie cas**
- Complexité dans le **cas moyen**

Exemple : Recherche séquentielle dans un tableau➔ Deux cas limites :

- ❑ Valeur recherchée au début du tableau ➔ **Tmin(N)=1**
- ❑ Valeur recherchée n'est pas dans le tableau ➔ **Tmax(N)=N**

Un calcul statistique d'une **complexité moyenne Tmoy(N)** peut s'avérer aussi représentatif.

3.1 Définitions

Soient D_N et $T_A(d,N)$ respectivement l'ensemble des entrées de taille N et la complexité temporelle de l'algorithme A pour une entrée $d \in D_N$.

Complexité dans le meilleur cas

Notée $T_{min}(N)$ est définie par : $T_{\min}(N) = Min\{T_A(d,N), d \in D_N\}$

Complexité dans le pire cas

Notée $T_{max}(N)$ est définie par : $T_{\max}(N) = Max\{T_A(d,N), d \in D_N\}$

Complexité dans cas moyen

Notée $T_{moy}(N)$ est définie par : $T_{moy}(N) = \sum_{d \in D_N} p(d) T_A(d,N)$

où p(d) est la **probabilité associée** à d

Si toutes les configurations sont **équiprobables**, la complexité cas moyen est simplifiée par : $T_{moy}(N) = \frac{\sum_{d \in D_N} T_A(d,N)}{|D_N|}$

$|D_N|$: Cardinal de D_N = Nombre de configurations possibles

Remarques

- $T_{\min}(N) \leq T_{moy}(N) \leq T_{\max}(N)$

- Si l'algorithme ne dépend que de la taille N, les trois complexités représentatives sont confondues.

3.2. Exemples

- Recherche séquentielle

```
public static int rechSeq(int[] T, int x){
    int i ;
    for(i=0;i<=T.length;i++)
        if(T[i]= = x) return i;
    return -1;
}
```

Meilleur cas : x au début de T

$T_{\min}(N) = 1$

Pire cas : x n'appartient pas à T

$T_{\max}(N) = N$

- Recherche dichotomique

```
public static int rechDich(int[] T, int x){
    int i , g=0,d=T.length,m;
    while(g<=d){
        m=(g+d)/2;
        if(T[m]= =x) return m;
        if(T[m]>x) d=m-1;
        g=m+1;}
    return -1;
}
```

$T_{\min}(N) = 1$

$T_{\max}(N) = \log_2 N$

Cas moyen : 1 exemple de calcul

- On se place dans les deux cas limites suivants : **$x \in T$ et $x \notin T$**.
- soit **q la probabilité** associée au cas $x \notin T$, la probabilité associée au cas $x \in T$ est alors donnée par **1-q**.
- le cas $x \in T$ peut être divisé en N cas suivants : x=T[i] , i=0 .. N-1
- Si on considère que ces N cas sont équiprobables càd que la probabilité associée à un cas est : (1-q)/N. On peut alors écrire :

$$T_{moy}(N) = qN + \frac{1-q}{N}\sum_{i=0}^{N-1} i + 1$$

$$= qN + \frac{1-q}{N}\sum_{i=1}^{N} i$$

$$= qN + \frac{1-q}{N}\frac{N(N+1)}{2}$$

$$= qN + \frac{(1-q)(N+1)}{2}$$

$$T_{moy}(N) = q\log_2 N + \frac{1-q}{N}\sum_{i=1}^{\log_2 N} i$$

$$= q\log_2 N + \frac{1-q}{N}\frac{\log_2 N(1+\log_2 N)}{2}$$

IV. Complexité asymptotique

1. Principe

Le calcul exacte de T(N) peut être **simplifié** par une fonction f(N) qui a le **même comportement** que T(N) quand **N augmente infiniment** ($n \to \infty$). La fonction f(N) est appelée **Complexité asymptotique**

2. Notation grand-O

Soit T(n) une fonction non négative. On dit que T(N) est un O(f(N)) s'il existe deux constantes positives C et N0 telle que :

$T(N) \leq C\ f(N)$ pour tout $N > N0$.

On dit que T(N) est **dominée asymptotiquement par f(N)**

Exemples

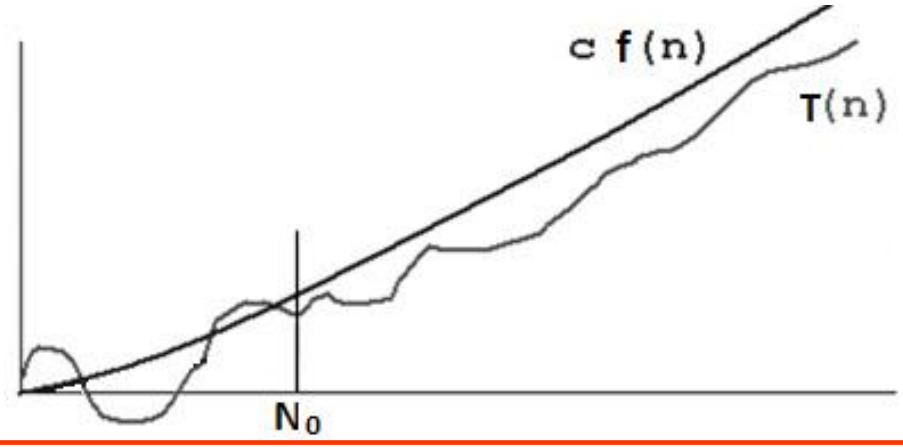
- Si $T(N) = 3N^2 + 2$ alors $T(N) = O(N^2)$.
- Initialiser un tableau d'entiers :

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

Si on compte toutes les actions :

T(N)=2+3*N = O(N)

Interprétation géométrique



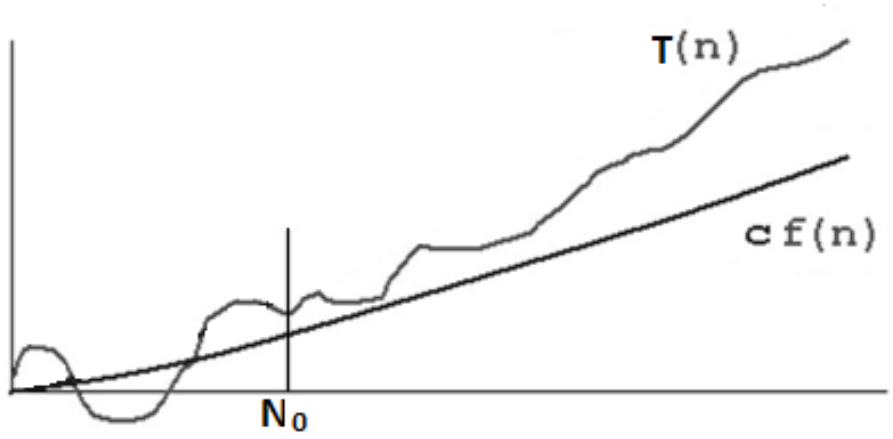
3. Notation Grand-Oméga

Soit T(n) une fonction non négative. On dit que T(N) est un Ω(f(N)) s'il existe deux constantes positives C et N0 telle que :

$T(N) \geq C\ f(N)$ pour tout $N > N0$.

On dit que T(N) **domine asymptotiquement** f(N)

Interprétation géométrique



Exemples

$T(n) = c_1n^2 + c_2n$.

$c_1n^2 + c_2n \geq c_1n^2$ pour tout $n > 1$.

$T(n) \geq cn^2$ pour $c = c_1$ et $n_0 = 1$. Ainsi, $T(n) = \Omega(n^2)$

4. Notation thêta

Lorsque le grand-O et le grand-oméga d'une fonction coïncident, on utilise alors la notation **grand-thêta**. Le temps d'exécution d'un algorithme est dans **Θ(f(N))** s'il est à la fois dans **O(f(N))** et dans **Ω(f(N)).**

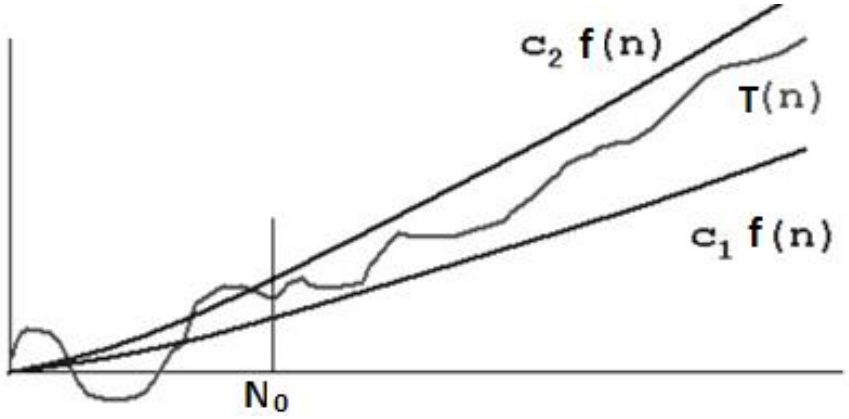
Autrement :

si T(n) est une fonction non négative. On dit que T(N) est un **Θ** (f(N)) s'il existe trois constantes positives C1, C2 et N0 telle que :

C1 f(N) ≤ T(N) ≤ C2 f(N) pour tout N > N0.

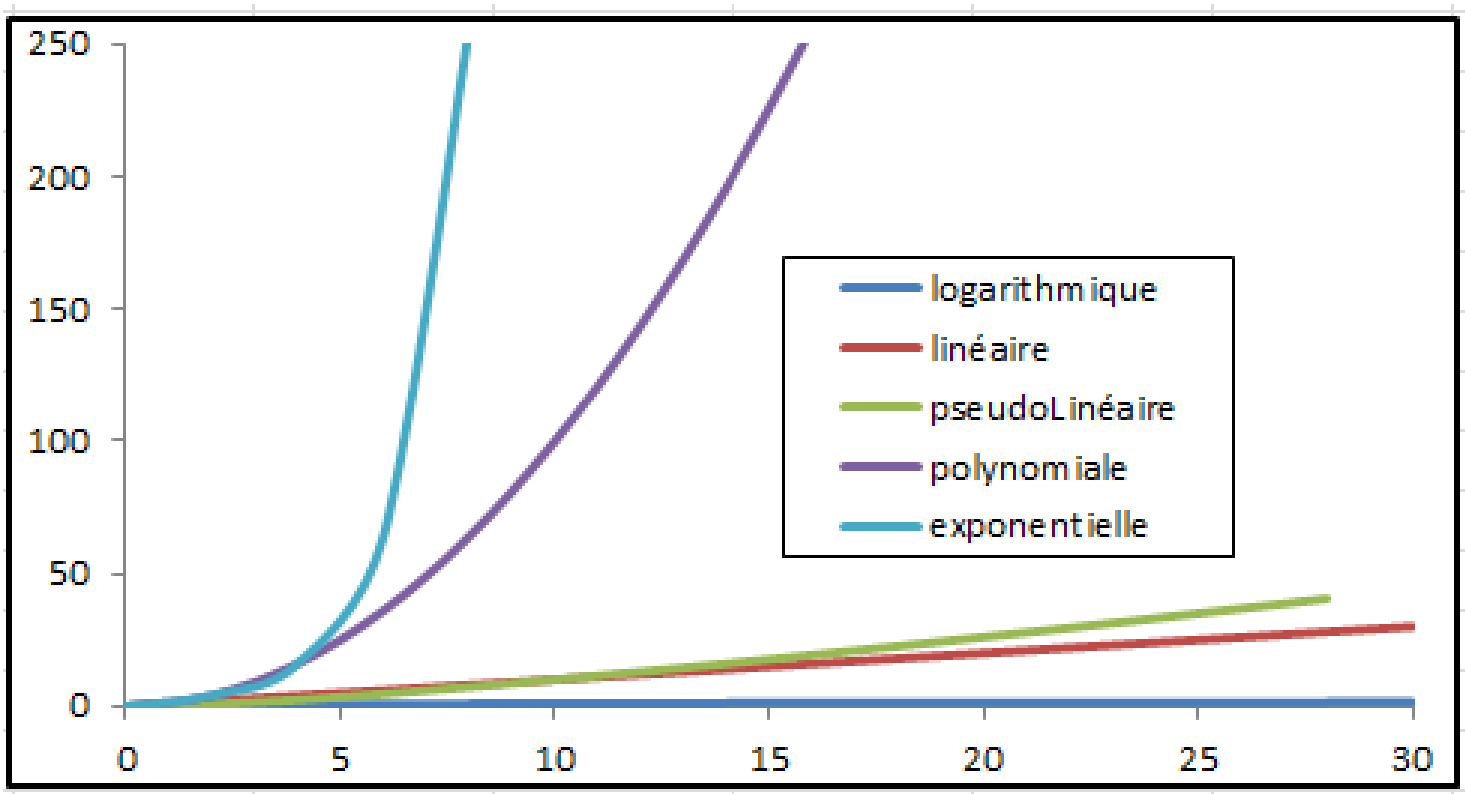
On dit que T et f sont **en même ordre.**

Interprétation géométrique



6. Classes de complexité

Complexité	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	4 s
$n = 30$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	18 min	10^{25} ans
$n = 50$	< 1 s	< 1 s	< 1 s	< 1 s	11 min	36 ans	∞
$n = 100$	< 1 s	< 1 s	< 1 s	$1s$	12,9 ans	10^{17} ans	∞
$n = 1000$	< 1 s	< 1 s	1s	18 min	∞	∞	∞
$n = 10000$	< 1 s	< 1 s	2 min	12 jours	∞	∞	∞
$n = 100000$	< 1 s	2 s	3 heures	32 ans	∞	∞	∞
$n = 1000000$	1s	20s	12 jours	31,710 ans	∞	∞	∞



Exercices

Exercice 3 : Supprimer un élément d'une liste

On considère une liste simplement chaînée d'entiers

1. Écrire des fonctions qui suppriment :
   - La 1ère occurrence de l'élément donné en paramètre dans la liste ;
   - Toutes les occurrences de l'élément donné en paramètre dans la liste.

   A chaque fois on renverra la liste initiale si l'élément à supprimer n'apparaît pas dans la liste.
2. Evaluer la complexité représentative pour les deux solutions

Exercice 4 : Afficher une liste simplement chaînée dans l'ordre inverse

1. Écrire une fonction qui afficher dans l'ordre inverse les éléments entiers d'une liste simplement chaînée.
2. Evaluer sa complexité.

Exercice 5 : Tri par fusion

L'algorithme de tri par fusion fait partie des algorithmes basés sur le principe « Diviser pour régner ». Il consiste à :

- diviser la tableau à trier en deux moitiés
- Trier chacune des deux moitiés selon le même principe tant que sa taille est supérieure à 1
- Fusionner les deux moitiés triées dans un tableau trié.

1. Écrire une fonction qui divise en tableau en deux moitiés
2. Ecrire une fonction qui fusionne deux tableau triés en un tableau trié
3. Ecrire une fonction tri par fusion
4. Evaluer les complexités représentatives des trois fonctions.

Exercice 6 : Analyse d'itérations emboitées

On considère les codes suivants

1. Code 1

```
for(i = 1; i<= n; i++)
        for( j = 1 ; j<= n ; j++) x += a;
```

2. Code 2

```
for(i = 1; i<= n; i++)
        for( j = 1 ; j<= i ; j++) x += a;
```

3. Code 3

```
for(i = 1; i<= n; i*=2)
        for( j = 1 ; j<= i ; j++) x += a;
```

4. Code 4

```
for(i = 1; i<= n; i++)
        for( j = 1 ; j<= i ; j++)
                for( k = j ; k>= 1 ; k--)  x += a;
```

Evaluer la complexité pour chacun des codes ci-dessus.

Exercice 7 : Recherche du maximum dans un tableau

On considère un tableau T d'entiers de taille n

1. Écrire une fonction itérative qui recherche et retourne l'entier maximal de T.
2. Donner une forme récursive de cette fonction
3. Donner la complexité théorique des deux fonctions en prenant comme opération fondamentale les opérations de comparaisons
4. Quelle est la classe de complexité de ces deux fonctions
5. Donner une approximation du temps de recherche du maximum que fera un ordinateur équipé d'un processeur 2GHz pour trouver le résultat dans un méga tableau de 2^{63} entiers. On supposera que le processeur est utilisé à son entière capacité pour ce traitement. On fera pour simplifier les calculs l'approximation suivante : $2^{10} \approx 10^3$
6. Refaire les questions de 1 à 5 pour le cas où T est pré-trié.

Exercice 8 : Recherche dans un tableau (Séquentielle)

On étudie un algorithme de recherche séquentielle dans un tableau T d'entiers de taille n. On se place dans le cas où il n'y a pas d'hypothèse sur le fait que le tableau est ordonné ni sur la présence de l'élément cherché dans le tableau.

1. Spécifier et écrire un tel algorithme,
2. Donner une forme récursive de cette fonction
3. Déterminer les cas favorables et défavorables et les complexités pire et meilleur en nombres de comparaisons correspondants.
4. On suppose que le cas défavorable est probable à 30% et que tous les autres cas restant sont équiprobables, donner une complexité moyenne.

Exercice 9 : Recherche dans un tableau trié (binaire)

On étudie un algorithme de recherche binaire dans un tableau T ordonné d'entiers de taille n. On se place dans le cas où il n'y a pas d'hypothèse sur la présence de l'élément cherché dans le tableau.

1. Spécifier et écrire un tel algorithme,
2. Donner une forme récursive de cette fonction
3. Déterminer les cas favorables et défavorables et les complexités pire et meilleur en nombres de comparaisons correspondants.
4. On suppose que le cas défavorable est probable à 20% et que tous les autres cas restant sont équiprobables, donner une complexité moyenne.