

COMPLEXITE ALGORITHMIQUE

1. INTRODUCTION

Exercice

Ecrire en python une fonction qui prend en argument une chaîne de caractères et détermine si le caractère 'a' est présent dans la chaîne (on retourne soit **True** soit **False**).

- Analysons plusieurs solutions :

<u>Première solution</u>	<u>Deuxième solution</u>
<pre>def contient1(chaine): k=0 N=len(chaine) trouve=False while(trouve == False and k < N): if chaine[k]=='a': trouve = True k=k+1 return trouve</pre>	<pre>def contienta2(chaine): for i in range(len(chaine)): if chaine[i]=='a': return True return False</pre>
<u>Troisième solution</u>	<u>Quatrième solution</u>
<pre>def contienta3(chaine): n=chaine.count('a') return bool(n)</pre>	<pre>def contienta4 (chaine): return ('a' in chaine)</pre>

- Quelques questions:

1. que remarquez-vous concernant cet exercice?
2. Le Coûte le plus court ! Est-il meilleur?
3. Comment peut-on désigner le meilleur Coûte parmi ces quatre solutions?

2. DÉFINITION DE LA COMPLEXITÉ D'UN ALGORITHME

- La complexité d'un problème mathématique **P** est une mesure de la quantité de ressources nécessaires à la résolution du problème **P**.
- Cette mesure est basée sur une estimation du nombre d'opérations de base effectuées par l'algorithme en fonction de la taille des données en entrée de l'algorithme.
- Généralement, pour le même problème **P**, on peut trouver plusieurs algorithmes (**Alg₁**; **Alg₂**;...; **Alg_n**).
- L'objectif de la complexité est d'évaluer le coût d'exécution de chaque algorithme afin de choisir le meilleur.

Problème:

Comment évaluer le coût d'exécution d'un algorithme donné?

3. TYPES DE COMPLEXITÉ

En fonction des ressources utilisées pour évaluer la complexité d'un algorithme, on sera amené à distinguer deux types de complexité : **complexité temporelle** et **complexité spatiale**.

Complexité temporelle (en temps)

La **complexité temporelle** d'un algorithme est le nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques) effectuées par un algorithme.

Complexité spatiale (en espace)

La complexité en mémoire donne le nombre d'emplacements mémoires occupés lors de l'exécution d'un programme.

Remarque : Dans ce cours, nous nous intéressons uniquement à la **complexité temporelle**.

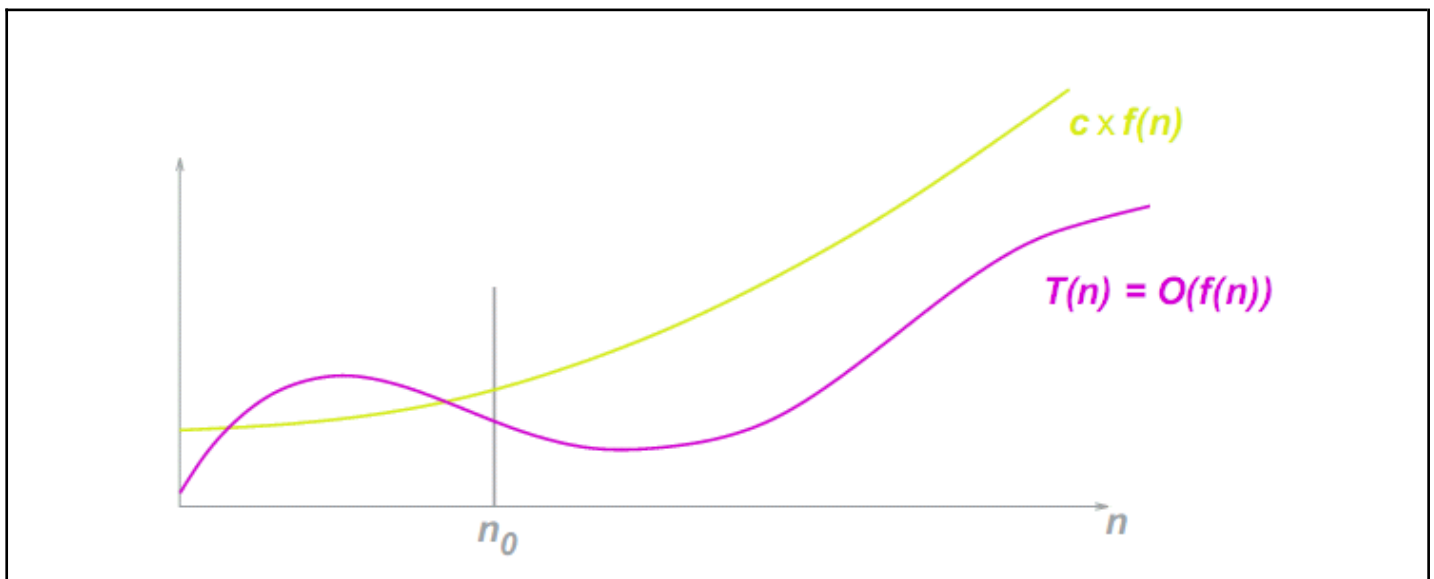
4. LA NOTATION O : (Complexité asymptotique)

Définition

Soit $T(n)$ une fonction qui désigne le temps de calcul d'un algorithme A.

On dit que $T(n)$ est en grand O de $f(n)$: $T(n) = O(f(n))$ si et seulement si:

$$\exists n_0 \in \mathbb{N}^*, \exists c \in \mathbb{R}, \forall n \geq n_0 : T(n) \leq c * f(n).$$



La notation $O(f(n))$ est aussi appelée **Notation de Landau : (Complexité asymptotique)** (i.e. quand $n \rightarrow \infty$)

Exemples

Exemple1:

Prouvons que la fonction $T(n)=3n+6$ est en $O(n)$ (on écrit aussi $3n+6 \in O(n)$)

But : trouver une constante $c \in \mathbb{R}$ et un seuil $n_0 \in \mathbb{N}^*$, à partir du quel $T(n) \leq c.n$

On remarque $3n+6 \leq 9n$ si $n \geq 2$

$$3*2+6 \leq 9*2$$

$$3*3+6 \leq 9*3$$

$$3*4+6 \leq 9*4$$

$$3*5+6 \leq 9*5$$

$\vdots$

On déduit donc que $c=9$ fonctionne à partir d'un seuil $n_0=2$

Remarque:

On ne demande pas d'optimisation (le plus petit c ou n_0 qui fonctionne), juste de donner des valeurs qui fonctionnent;

$c=10$ et $n_0=5$ est donc aussi acceptable;

$$3*5+6 \leq 10*2$$

$$3*6+6 \leq 10*3$$

$$3*7+6 \leq 10*4$$

Exemple2:

Prouvons que la fonction : $T(n) = n^2 + 3n$ est en $O(n^2)$

But : trouver une constante $c \in \mathbb{R}$ et un seuil $n_0 \in \mathbb{N}^*$, à partir duquel $T(n) \leq c.n$

Cherchons d'abord la constante c ; $c=1$ ne peut pas marcher, essayons donc $c=2$;

On doit alors trouver un seuil $n_0 \in \mathbb{N}^*$, à partir duquel : $n^2 + 3n \leq 2n^2$

On remarque $n^2 + 3n \leq 2n^2$ si $n \geq 3$

On déduit donc que $c=2$ fonctionne à partir d'un seuil $n_0=3$

5. PROPRIÉTÉS DE LA NOTATION O :

✓ $\forall k \in \mathbb{R}$, on a toujours $k \in O(1)$. (P1)

✓ $\forall k \in \mathbb{R}$, on a toujours $k.f(x) \in O(f(x))$ (P2)

Cette propriété implique la **réflexivité** de la notation grand O : $f(x) \in O(f(x))$.

✓ Si $f_1(x) \in O(g(x))$ et $f_2(x) \in O(g(x))$, alors $(f_1 + f_2)(x) \in O(g(x))$ (P3)

✓ Si $f_1(x) \in O(g_1(x))$ et $f_2(x) \in O(g_2(x))$, alors $(f_1 + f_2)(x) \in O(\max(g_1(x), g_2(x)))$ (P4)

✓ Si $f_1(x) \in O(g_1(x))$ et $f_2(x) \in O(g_2(x))$, alors $(f_1 \cdot f_2)(x) \in O(g_1(x) \cdot g_2(x))$ (P5)

✓ Si $p > 0$, alors $\log_2 x \in O(x^p)$ (P6)

✓ Si $b > 1$, alors $\log_b x \in O(\log_2 x)$ (Tous les logarithmes, $\forall$ la base, croissent au même rythme, à un facteur près) (P7)

✓ (Transitivité) : Si $f(x) \in O(g(x))$ et $g(x) \in O(h(x))$, alors $f(x) \in O(h(x))$ (P8)

✓ Si $f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ (où les a_i sont tous des réels),

Alors $f(x) \in O(x^n)$ (P9)

Note :

Quand $f(x) \in O(g(x))$ et $g(x) \in O(f(x))$, on dit que $f(x)$ et $g(x)$ sont du même ordre (de la **même classe**) et on écrit : $f(x) \in \Theta(g(x))$

6. CLASSES DE COMPLEXITE LES PLUS USUELLES

On voit souvent apparaître les complexités suivantes:

Complexité	Nom courant	Description
$O(1)$	Constante	Le temps d'exécution ne dépend pas des données traitées, ce qui est assez rare!
$O(\log(n))$	Logarithmique	augmentation très faible du temps d'exécution quand le paramètre croît.
$O(n)$	Linéaire	augmentation linéaire du temps d'exécution quand le paramètre croît (si le paramètre double, le temps double).
$O(n \log(n))$	Quasi-linéaire	augmentation un peu supérieure à $O(n)$
$O(n^2)$	Quadratique	quand le paramètre double, le temps d'exécution est multiplié par 4. Exemple: algorithmes avec deux boucles imbriquées.

$O(n^k)$	Polynômiale	ici, n^k est le terme de plus haut degré d'un polynôme en n ; il n'est pas rare de voir des complexités en $O(n^3)$ ou $O(n^4)$.
$O(k^n)$	Exponentielle	quand le paramètre double, le temps d'exécution est élevé à la puissance k avec $k > 1$.
$O(n!)$	Factorielle	asymptotiquement équivalente à n^n

7. COMPARAISON DE TEMPS D'EXECUTION

- Prenons un ordinateur, dont la vitesse du microprocesseur est **1GHz** (c.à.d. il exécute un milliard d'opérations par seconde), et imaginons des algorithmes qui effectuent un traitement donné dans un temps **$T(N)$** .
- Nous donnons ci-dessous un tableau des ordres de grandeur des temps d'exécution que l'on peut espérer pour N éléments suivant la complexité de l'algorithme.

Complexité	Temps d'exécution en fonction du nombre d'opérations					
N	5	10	15	20	100	1000
$\log N$	$3.10^{-9}s$	$4.10^{-9}s$	$5.10^{-9}s$	$7.10^{-9}s$	$10^{-8}s$	$3.10^{-9}s$
$2N$	$10^{-8}s$	$2.10^{-8}s$	$3.10^{-8}s$	$4.10^{-8}s$	$2.10^{-7}s$	$2.10^{-6}s$
$N \log N$	$1, 2.10^{-8}s$	$3.10^{-8}s$	$6.10^{-8}s$	$10^{-7}s$	$7.10^{-7}s$	$10^{-5}s$
N^2	$2, 5.10^{-8}s$	$10^{-7}s$	$2, 3.10^{-7}s$	$4.10^{-7}s$	$10^{-5}s$	$10^{-3}s$
N^5	$3.10^{-6}s$	$10^{-4}s$	$7, 2.10^{-4}s$	$3.10^{-3}s$	$10s$	11jours
2^N	$3, 2.10^{-8}s$	$10^{-6}s$	$3, 3.10^{-5}s$	$10^{-3}s$	4.10^{13}ans	3.10^{284}ans
$N!$	$1, 2.10^{-7}s$	$4.10^{-3}s$	23minutes	77ans	$3.10^{141}s$	$10^{500}s$
N^N	$3.10^{-6}s$	$10s$	13ans	3.10^9ans	3.10^{183}ans	$10^{300}s$

8. COMMENT MESURER LA COMPLEXITE D'UN ALGORITHME

a. Le coût des instructions élémentaires

✓ Opération élémentaire.

On appelle opération de base, ou opération élémentaire, toute:

- Affectation;
- Test de comparaison: $==$; $<$; $<=$; $>=$; $!=$
- Opération de lecture (**input**) et d'écriture (**print**);
- Opération arithmétique : $+$; $-$; $*$; $/$; $\%$

Le coût d'une opération élémentaire est égal à 1.

Exemple1: Que vaut le coût de l'algorithme A

```
somme = n +1           #instr1
somme = somme * n       #instr2
somme = somme/2         #instr3
```

$$\begin{aligned}\text{Coût}(A) &= \text{Coût}(\text{instr1}) + \text{Coût}(\text{instr2}) + \text{Coût}(\text{instr3}) \\ &= 2 + 2 + 2 \\ &= 6\end{aligned}$$

b. Le coût des instructions composées

✓ Opération composée

On appelle opération composée, toute instruction contenant:

- L'exécution d'une instruction conditionnelle :

```
Si (test) alors
    Q1
Sinon
    Q2
Finsi
```

Le nombre d'opérations est: $\text{Coût}(P) = \text{Coût}(\text{test}) + \max(\text{Coût}(Q1), \text{Coût}(Q2))$

- **L'exécution d'une boucle** : est égal à la multiplication de nombre de répétitions par la somme du coût de chaque instruction x_i du corps de la boucle;

$$\text{Coût (bouclefor)} = \sum \text{Coût}(x_i)$$

$$\text{Coût(boucle while)} = \sum (\text{Coût}(comparaison) + \text{Coût}(x_i))$$

- **L'appel d'une fonction** : Lorsqu'une fonction ou une procédure est appelée, le coût de cette fonction ou procédure est le nombre total d'opérations élémentaires engendrées par l'appel de cette fonction.

Exemple2: Que vaut le coût de l'algorithme B

```
if i%2==0:
    n=i//2
else:
    i=i+1
    n=i//2
```

$$\begin{aligned} \text{Coût (B)} &= \text{Coût}(i\%2==0) + \max(\text{Coût}(n = i//2), \text{Coût}(i = i + 1, n = i//2)) \\ &= 2 + \max(2, 4) \\ &= 6 \end{aligned}$$

Exemple3: Que vaut le coût de l'algorithme C

```
i=1
while i <= n :
    somme = somme + i
    i = i+1
```

$$\begin{aligned} \text{Coût(C)} &= 1 + \sum (\text{Coût}(i \leq n) + \text{Coût}(somme = somme + i) + \text{Coût}(i = i + 1)) \\ &= 1 + n + 2n + 2n \\ &= 1 + 5n \end{aligned}$$

c. Principe pour calculer la complexité d'un algorithme

La mesure de complexité d'un algorithme consiste à évaluer le temps d'exécution de cet algorithme.

Dans l'évaluation de ce temps d'exécution (**coût**), on sera amené à suivre les étapes suivantes:

- Déterminer les opérations élémentaires (**OE**) à prendre en considération : coût $T(OE)$.
- Calculer le nombre d'instructions effectuées par les opérations composées (**OC**) : coût $T(OC)$.
- Préciser les instructions à négliger.
- Le coût de l'algorithme $T(Alg)$ est la somme des deux coûts $T(OE)$ et $T(OC)$: $T(Alg) = T(OE) + T(OC)$

d. La notation O : motivation

- Les calculs à effectuer pour évaluer le temps d'exécution d'un algorithme peuvent parfois être longs et pénibles;
- De plus, le degré de précision qu'ils requièrent est souvent inutile;
- On aura donc recours à une **approximation** de ce temps de calcul, représenté par la notation $O(\cdot)$;

e. Règles de calcul : simplifications

- On calcule le temps d'exécution comme avant, mais on effectue les simplifications suivantes:
 - On oublie les constantes multiplicatives (elles valent 1);
 - On annule les constantes additives;
 - On ne retient que les termes dominants;
- Exemple (simplifications)

Soit un algorithme effectuant $T(n) = 4n^3 - 5n^2 + 2n + 3$ opérations;

- On remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$
- On annule les constantes additives : $n^3 - n^2 + n + 0$
- On garde le terme de plus haut degré : $n^3 + 0$

Et on a donc : $T(n) = O(n^3)$.

- **Justification des simplifications**
- Les processeurs actuels effectuent plusieurs milliards d'opérations à la seconde;
 1. qu'une affectation requière **2** ou **4** unités de temps ne change donc pas grand-chose ; d'où le remplacement des constantes par des **1** pour les multiplications.
 2. un nombre constant d'instructions est donc aussi négligeable par rapport à la croissance de la taille des données ; d'où l'annulation des constantes additives.
 3. pour de grandes valeurs de **n**, le terme de plus haut degré l'emportera ; d'où l'annulation des termes inférieurs
- On préfère donc avoir une **idée** du temps d'exécution de l'algorithme plutôt qu'une expression plus précise mais inutilement compliquée;

f. Exemples de calcul de complexité

Exemple 1	Exemple 2
• La fonction suivante permet de retourner le quotient et le reste de la division d'un nombre entier a par b. <pre>def division (a ,b): q=a//b r=a%b return (q,r)</pre> • Exemple <code>>>> x , y = 10 , 3</code> <code>>>> division(x , y)</code> <code>3 , 1</code> • Le nombre d'opérations est: 5 • Temps de calcul est constant • Complexité: O(1)	La fonction suivante permet de retourner la somme des éléments d'une liste: <pre>def Somme (L) : s=0 for i in rang(len(L)) : s=s+L[i] return s</pre> Exemple <code>>>> L =[1, 2, 8, 4]</code> <code>>>> Somme (L)</code> <code>15</code> • Le paramètre de complexité est la taille de la liste d'entrée L. • en fixant i on exécute 2 opérations: (addition et affectation) • Nombre de fois d'exécution de ces 2 opérations est: len(T) • Le nombre total d'opérations est : 4 * len(L) + 3 • Complexité: O(len(L))= O(n)

Exemple 3 : remplir un tableau	Exemple 4 : remplir une matrice
<pre>def RemplirTab (T): for i in range(len(T)): print("T[",i, "]=",end=' ') T[i]=int(input())</pre> • Le paramètre de complexité est la taille du tableau d'entrée T. • en fixant i on exécute 4 opérations: (print, input, int et affectation) • Nombre de fois d'exécution de ces 4 opérations est: len(T) • Le nombre total d'opérations est : 4 * len(T) • Complexité: O(len(T))= O(n)	<pre>def RemplirMat (M) : n=len(M) p=len(M[0]) for i in range(n): for j in range(p): print("T[",i, "][",j, "]=") T[i][j]=int(input())</pre> • Coût pour saisir une valeur est : 4 • Le nombre d'itérations de la boucle sur j pour i fixé égal à : p • Le nombre total pour lire toutes les valeurs pour i fixé égal à 4p • Le nombre total d'opérations est : 2 + 4p.n • Complexité : O(n.p)

Exemple 5 : Produit matriciel

```
def prodMatrice(A,B):
    n=len(A) #nombre de lignes de A
    m=len(A[0]) #nombre de colonne de A
    p=len(B[0]) #nombre de colonnes de B
    C=[p*[0] for i in range(n)]
    for i in range(0,n):
        for j in range (0,p):
            s=0;
            for k in range (0,m):
                s = s + A[i][k]*B[k][j]
            C[i][j]=s
    return C
```

- On suppose que **A** et **B** sont deux matrices carrées d'ordre $n = m = p$
- Coût pour calculer une valeur de **s** est : **3** (produit, addition et affectation)
- Le nombre d'itérations de la boucle sur **k** pour **j** fixé égal à $m = n$
- Le nombre d'itérations de la boucle sur **j** pour **i** fixé égal à $p = n$
- Le nombre total d'opérations est : $T(n)=4n + n(n(2+n(3)))$
- Complexité: $O(n^3)$

Exemple6 : Tri par sélection

```
def TriSelection(T):
    n=len(T)
    for i in range(n-1):
        Posmin=i
        for j in range(i+1,n):
            if T[j] < T[Posmin]:
                Posmin=j
        #Permutation
        T[i] , T[Posmin] = T[Posmin] , T[i]
```

- **Coût des échanges:**
 - Le tri par sélection sur **n** nombres fait **n-1** échanges, ce qui fait : **3(n-1)** affectations
- **Coût des recherches de minimum:**
 - On recherche le minimum parmi **n** éléments : au plus **3(n-1)** opérations. (c'est le nombre d'itérations de la boucle sur **j** pour **i** fixé égal à **n-1**)
 - On recherche le minimum parmi **n-1** éléments : au plus **3(n-2)** opérations. (c'est le nombre d'itérations de la boucle sur **j** pour **i** fixé égal à **n-2**)
 - On recherche en suite le minimum parmi **n-2** éléments: **3(n-3)** tests.
 - ...
- Le nombre total de tests est : $3(1+2+3+ \dots + (n-2) + (n-1)) = 3(n-1)n/2$
- Le nombre total d'opérations est : $3(n-1) + 3((n-1)n)/2$
- Donc, la complexité est **quadratique** : $O(n^2)$

Exemple 7 : recherches séquentielle

```
def Recherche_Seq(T,x):  
    i=0  
    n=len(T)  
    for i in range(n):  
        if T[i]==x :  
            return True  
    return False
```

- Le nombre d'opérations avant for est : **3**
- La boucle for contient **2** opérations: (test et return True)
- Le nombre de fois d'exécution de ces **2** instructions dans le pire de cas est **n**
- Le nombre d'opérations après la boucle for est : **1**(return False)
- Le nombre d'opérations total est : **3+2n+1**
- Complexité : **O(n)**

Exemple 8 : Recherche dichotomique

```
def RechDichotomique(T,x):  
    g,d=0,len(T)-1  
    while g<=d:  
        m=(g+d)//2  
        if T[m]==x: return True  
        if T[m]<x:  
            g=m+1  
        else:  
            d=m-1  
    return False
```

- Soit **k** le nombre de passages dans la boucle **while**.
- on divise le nombre d'éléments restants par 2 jusqu'à ce qu'il n'en reste qu'un (k divisions)

$$(((n/2)/2)/2)/..../2=1$$

- Donc $n/2^k = 1$ et ainsi $k = \log_2 n$
- Complexité: **O(log₂(n))**

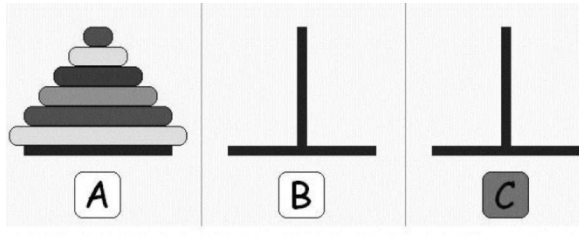
Exemple9 : Recherche d'un mot dans une chaîne de caractères

Le principe est le même que pour la recherche d'un caractère dans une chaîne mais ici il est nécessaire d'ajouter une boucle "while" pour tester tous les caractères du mot.

```
def searchWord (mot, texte):  
    n = len(texte)  
    m = len(mot)  
    if m > n : return False  
    for i in range(1+ n-m):  
        j=0  
        while j < len(mot) and mot[j] == texte[i+j]:  
            j=j+1  
        if j == len(mot): return i  
    return None
```

- Le programme va chercher le mot à toutes les places possibles et va tester tous les caractères du mot
- Le nombre d'opérations total est de l'ordre de $m(1 + n - m)$ où m est la longueur du mot et n la longueur du texte
- En particulier le maximum est atteint pour $m = n/2$ et on obtient $(n^2 + 2n)/4$
- Donc la complexité de l'algorithme est $O(n^2)$

Exemple.10 : les tours de Hanoi



```
def hanoi ( n , A='A' , B='B' , C='C'):  
    if n > 0:  
        hanoi(n-1, A, C, B)  
        print("Déplacer le disque %d de la tige %s vers la tige %s." % (n, A, C))  
        hanoi(n-1, B, A, C)
```

- L'évaluation de la complexité de cet algorithme est assez simple. Nous allons chercher à savoir combien de déplacements de disques sont nécessaires pour arriver à nos fins. Très simplement, la fonction calculant ceci peut être définie de la manière suivante (on peut se baser sur la fonction ci-dessus) :
- $T(0) = 0$; $T(1) = 1$; $T(n) = 2 * T(n-1) + 1$. On peut démontrer par récurrence que ceci est égal à $T(n) = 2^n - 1$:
- On a $T(n+1) = 2 * T(n) + 1$. En supposant que $T(n) = 2^n - 1$, on a $T(n+1) = 2 * (2^n - 1) + 1$. On développe ensuite en $T(n+1) = 2 * 2^n - 2 + 1$ ce qui nous donne au final $T(n+1) = 2^{n+1} - 1$. On a ainsi démontré que si c'est vrai pour n , alors c'est vrai pour $(n+1)$. Comme on a $T(1) = 2 * 0 + 1 = 2^1 - 1 = 1$, c'est vrai pour tous les n supérieurs à 1. La complexité exacte de l'algorithme présenté ci-dessus est donc en $2^n - 1$ pour n disques à déplacer de la tour
- On trouve $T(n) = 2^n - 1$
- Donc la complexité est exponentielle $O(2^n)$

9. DIFFÉRENTES NUANCES DE COMPLEXITÉ

- Il est fréquent que la **complexité en temps** soit améliorée au prix d'une augmentation de la **complexité en espace**, et vice-versa.
- La complexité dépend notamment:
 - De la puissance de la machine sur la quelle l'algorithme est exécuté,
 - Du langage et interpréteur (ou compilateur) utilisé pour coder l'algorithme,
 - Pour des données de même taille, un algorithme n'effectue pas nécessairement le même nombre d'opérations élémentaires.
- Pour ce la, on distingue 3 types de complexités:
 - 1- **Complexité au pire des cas.**
 - 2- **Complexité dans le meilleur des cas.**
 - 3- **Complexité en moyenne des cas.**

9-1 : Complexité au pire des cas

La **complexité au pire** est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée à n .

$$T_{\max}(n) = \max_{d \in D_n} \{C(d)\}$$

- Avec D_n l'ensemble des données de taille n
- $C(d)$ est le coût d'exécution de l'algorithme sur la donnée d de taille n .

Exemple : Recherche d'un élément dans un tableau

```
def find(T,x):  
    for e in T:  
        if e==x:  
            return True  
    return False
```

On note n la longueur du tableau $T(n=\text{len}(T))$.

Dans le pire des cas : quand l'élément recherché x est le dernier (dans la case $n-1$) ou est absent.

Donc la complexité dans le pire est $C_{\max}(n) = n$

9-2 : Complexité dans le meilleur des cas

La **complexité au meilleur** est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée. C'est une borne inférieure de la complexité de l'algorithme sur un jeu de données de taille n .

$$T_{\min}(n) = \min_{d \in D_n} \{C(d)\}$$

Exemple: Recherche d'un élément dans un tableau

On considère le même exemple précédent.

Dans le meilleur des cas : quand l'élément recherché x se trouve en **1ère** position.

Donc la complexité dans le meilleur des cas est $C_{\min}(n) = 1$

9-3 : Complexité en moyenne des cas

La **complexité en moyenne** est la moyenne des complexités de l'algorithme sur des jeux de données de taille n . Cette complexité en moyenne reflète le **comportement "général"** de l'algorithme si les cas extrêmes sont rares ou si la complexité varie peu en fonction des données.

$$T_{\text{moy}}(n) = \left\{ \sum Pr(d) \cdot C(d), d \in D_n \right\}$$

Où $Pr(d)$ est la probabilité d'avoir la donnée d en entrée de l'algorithme.

Exemple : Recherche d'un élément dans un tableau

En moyenne, dans l'hypothèse où la probabilité de trouver x dans le tableau est q et si x est présent, il peut se trouver dans n'importe quelle case avec la même probabilité, à savoir $1/n$

Alors la probabilité que x se trouve dans la i ème case est : $q * 1/n$, d'où:

$$C_{\text{moy}}(n) = q * (1/n + 2/n + 3/n + \dots + n/n) + (1-q) * n = q(n+1)/2 + n(1-q)$$

Si $q=1$, la complexité en moyenne des cas sera $C_{\text{moy}}(n) = (n+1)/2$

10. COMPLEXITÉ ET RÉCURSIVITÉ

Il existe diverses techniques pour la résolution des équations de récurrence.

- méthode des fonctions génératrices et décomposition des fractions rationnelles
- Transformée en Z
- ...

Résolution des récurrences.

❖ $C(n) = C(n-1) + b$

- solution : $C(n) = c(0) + b \cdot n = O(n)$
- exemples : factorielle, recherche séquentielle récursive dans un tableau

❖ $C(n) = a \cdot C(n-1) + b, a \neq 1$

- solution : $C(n) = a^n \cdot (C(0) - b/(1-a)) + b/(1-a) = O(a^n)$
- exemples : répétition a fois d'un traitement sur le résultat de l'appel récursif

❖ $C(n) = C(n-1) + a \cdot n + b$

- solution : $C(n) = c(0) + a \cdot n \cdot (n+1)/2 + n \cdot b = O(n^2)$.
- exemples : traitement en coût linéaire avant l'appel récursif, tri à bulle

❖ $C(n) = C(n/2) + b$

- solution : $C(n) = C(1) + b \cdot \log_2(n) = O(\log(n))$
- exemples : élimination de la moitié des éléments en temps constant avant l'appel récursif, recherche dichotomique récursive

❖ $C(n) = a \cdot C(n/2) + b, a \neq 1$

- solution : $C(n) = n^{\log_2(a)} \cdot (c(1) - b/(1-a)) + b/(1-a) = O(n^{\log_2(a)})$
- exemples : répétition a fois d'un traitement sur le résultat de l'appel récursif dichotomique

❖ $C(n) = C(n/2) + n$

- solution : $C(n) = O(n)$
- exemples : traitement linéaire avant l'appel récursif dichotomique

❖ $C(n) = 2 \cdot C(n/2) + a \cdot n + b$

- solution : $C(n) = O(n \cdot \log(n))$
- exemples : traitement linéaire avant double appel récursif dichotomique, tri fusion