IFT339
Structures de données

Thème 2 : Complexité algorithmique

Aïda Ouangraoua
Département d'informatique

UNIVERSITÉ DE
SHERBROOKE

Contexte

- Plusieurs algorithmes possibles pour une même opération
- Comparaison et choix de l'algorithme en fonction de deux critères : temps de calcul et espace de stockage (ressources à optimiser)

Contexte

- Deux types de comparaisons possibles

  - **Expérimentale** : implémenter les algorithmes sur la même machine avec les mêmes choix de programmation, et exécuter sur le même jeu de données.
    Limites : nécessité d'implémenter tous les algorithmes et tester sur des données de tailles raisonnables ; résultats de comparaison valables uniquement sur des données similaires aux données testées

  - **Théorique** : évaluer les complexités théoriques des algorithmes avant de choisir lequel implémenter

Définition

- **Complexité en temps** dans le pire des cas : expression du **nombre maximum d'instructions élémentaires** effectuées lors d'une exécution de l'algorithme en fonction de la taille n de l'entrée ➔ borne supérieure du nombre d'instructions requises pour une exécution en fonction de n.

- Exemples d'instruction élémentaire
  - Appel de fonctions
  - Affectation
  - Opérateurs arithmétiques sur des types primitifs scalaires
  - Retour de fonction

Définition

- **Complexité en temps** dans le pire des cas : expression du **nombre maximum d'instructions élémentaires** effectuées lors d'une exécution de l'algorithme en fonction de la taille n de l'entrée ➔ borne supérieure du nombre d'instructions requises pour une exécution en fonction de n.

Définition

- **Complexité en temps** dans le pire des cas : expression du **nombre maximum d'instructions élémentaires** effectuées lors d'une exécution de l'algorithme en fonction de la taille n de l'entrée ➔ borne supérieure du nombre d'instructions requises pour une exécution en fonction de n.

- Exemple

Chaine_bin : Chaine_dec → Chaine_bin
Entree : x

Pour i de n-1 à 0, faire:
    $b_i \leftarrow x / 2^i$
    $x \leftarrow x - b_i * 2^i$
res.chaine ← $b_{n-1}b_{n-2} \ldots b_1b_0$
Sortie : res

n X
(1
+1)
+1 (cas1: compte 1)

T(n) = 2n +1

Définition

- **Complexité en espace** dans le pire des cas : expression de l'**espace de stockage maximum requis** pour les variables locales lors d'une exécution de l'algorithme en fonction de la taille de l'entrée ➔ borne supérieure de la taille de l'espace requis en fonction de n (taille de l'entrée).

Définition

- **Complexité en espace** dans le pire des cas : expression de l'**espace de stockage maximum requis** pour les variables locales lors d'une exécution de l'algorithme en fonction de la taille de l'entrée ➔ borne supérieure de la taille de l'espace requis en fonction de n (taille de l'entrée).

- Exemple

Chaine_bin : Chaine_dec → Chaine_bin
Entree : x

Pour i de n-1 à 0, faire:
    $b_i \leftarrow x / 2^i$
    $x \leftarrow x - b_i * 2^i$
res.chaine ← $b_{n-1}b_{n-2} \ldots b_1b_0$
Sortie : res

n X
(1)

+n (cas2: compte n)

E(n) = 2n

Définition

- **Complexité en moyenne** (en temps ou en espace): **moyenne de la complexité pour les différentes tailles d'entrée** possibles, pondérées suivant la distribution des entrées.

Définition

- **Complexité en moyenne** (en temps ou en espace): **moyenne de la complexité pour les différentes tailles d'entrée** possibles, pondérées suivant la distribution des entrées.

- Exemple:

Taille possible	Complexité	Distribution
1	1	1/2
2	2	1/4
3	3	1/8
4	4	1/16
5	5	1/32
6	6	1/64
7	7	1/128
…	…	…
m	m	$1/2^m$

Définition

- **Complexité en moyenne** (en temps ou en espace): **moyenne de la complexité pour les différentes tailles d'entrée** possibles, pondérées suivant la distribution des entrées.

- Exemple:

Taille possible	Complexité	Distribution
1	1	1/2
2	2	1/4
3	3	1/8
4	4	1/16
5	5	1/32
6	6	1/64
7	7	1/128
…	…	…
m	m	$1/2^m$

S = ½ + 2/4 + 3/8 + 4/16 + ….

Définition

- **Complexité en moyenne** (en temps ou en espace): **moyenne de la complexité pour les différentes tailles d'entrée** possibles, pondérées suivant la distribution des entrées.

- Exemple:

Taille possible	Complexité	Distribution
1	1	1/2
2	2	1/4
3	3	1/8
4	4	1/16
5	5	1/32
6	6	1/64
7	7	1/128
…	…	…
m	m	$1/2^m$

S = ½ + 2/4 + 3/8 + 4/16 + ….

S-½ = 2/4 + 3/8 + 4/16 + 5/32 + …

Définition

- **Complexité en moyenne** (en temps ou en espace): **moyenne de la complexité pour les différentes tailles d'entrée** possibles, pondérées suivant la distribution des entrées.

- Exemple:

Taille possible	Complexité	Distribution
1	1	1/2
2	2	1/4
3	3	1/8
4	4	1/16
5	5	1/32
6	6	1/64
7	7	1/128
…	…	…
m	m	$1/2^m$

S = ½ + 2/4 + 3/8 + 4/16 + ….
S-½ = 2/4 + 3/8 + 4/16 + 5/32 + …
(S-½)/2 = 2/8 + 3/16 + 4/32 + …

Définition

- **Complexité en moyenne** (en temps ou en espace): **moyenne de la complexité pour les différentes tailles d'entrée** possibles, pondérées suivant la distribution des entrées.

- Exemple:

Taille possible	Complexité	Distribution
1	1	1/2
2	2	1/4
3	3	1/8
4	4	1/16
5	5	1/32
6	6	1/64
7	7	1/128
…	…	…
m	m	$1/2^m$

S = ½ + 2/4 + 3/8 + 4/16 + ….
S-½ = 2/4 + 3/8 + 4/16 + 5/32 + …
(S-½)/2 = 2/8 + 3/16 + 4/32 + …
(S-½)-(S-½)/2 = 2/4 +1/8 +1/16+ 1/32 +…
(S-½)-(S-½)/2 = 2/4 + ¼ = 3/4

Définition

- **Complexité en moyenne** (en temps ou en espace): **moyenne de la complexité pour les différentes tailles d'entrée** possibles, pondérées suivant la distribution des entrées.

- Exemple:

Taille possible	Complexité	Distribution
1	1	1/2
2	2	1/4
3	3	1/8
4	4	1/16
5	5	1/32
6	6	1/64
7	7	1/128
…	…	…
m	m	$1/2^m$

S = ½ + 2/4 + 3/8 + 4/16 + ….
S-½ = 2/4 + 3/8 + 4/16 + 5/32 + …
(S-½)/2 = 2/8 + 3/16 + 4/32 + …
(S-½)-(S-½)/2 = 2/4 +1/8 +1/16+ 1/32 +…
(S-½)-(S-½)/2 = 2/4 + ¼ = 3/4
S-½ = 6/4
S = 8/4

Définition

- **Complexité en moyenne** (en temps ou en espace): **moyenne de la complexité pour les différentes tailles d'entrée** possibles, pondérées suivant la distribution des entrées.

- Exemple:

Taille possible	Complexité	Distribution
1	1	1/2
2	2	1/4
3	3	1/8
4	4	1/16
5	5	1/32
6	6	1/64
7	7	1/128
…	…	…
m	m	$1/2^m$

S = ½ + 2/4 + 3/8 + 4/16 + ….
S-½ = 2/4 + 3/8 + 4/16 + 5/32 + …
(S-½)/2 = 2/8 + 3/16 + 4/32 + …
(S-½)-(S-½)/2 = 2/4 +1/8 +1/16+ 1/32 +…
(S-½)-(S-½)/2 = 2/4 + ¼ = 3/4
S-½ = 6/4
S = 8/4

Définition

- **Complexité en moyenne** (en temps ou en espace): **moyenne de la complexité pour les différentes tailles d'entrée** possibles, pondérées suivant la distribution des entrées.

- Exemple:

Taille possible	Complexité	Distribution
1	1	1/2
2	2	1/4
3	3	1/8
4	4	1/16
5	5	1/32
6	6	1/64
7	7	1/128
…	…	…
m	m	$1/2^m$

S = ½ + 2/4 + 3/8 + 4/16 + ….
S-½ = 2/4 + 3/8 + 4/16 + 5/32 + …
(S-½)/2 = 2/8 + 3/16 + 4/32 + …
(S-½)-(S-½)/2 = 2/4 +1/8 +1/16+ 1/32 +…
(S-½)-(S-½)/2 = 2/4 + ¼ = 3/4
S-½ = 6/4
S = 8/4 =2

Pire cas : T(n) = n ; Moyenne : T(n) = 2

Exercice 1

Calculer le nombre maximum d'instructions élémentaires nécessaires pour chaque opération du type Entier, en fonction de la taille de l'entrée n (nombre de bits de la représentation).

complement : Ø → Chaine_bin
Entree : Ø
res ← Chaine_bin(0)
$b_{n-1}b_{n-2} \ldots b_1b_0$ ← __.chaine
Pour i de n-1 à 0, faire:
    c_i ← 1- b_i
res.chaine ← $c_{n-1}c_{n-2} \ldots c_1c_0$
Sortie : res

Exercice 1

Calculer le nombre maximum d'instructions élémentaires nécessaires pour chaque opération du type Entier, en fonction de la taille de l'entrée n (nombre de bits de la représentation).

decaler : Entier → Chaine_bin
Entree : k
res ← Chaine_bin(0)
$b_{n-1}b_{n-2} \dots b_1b_0$ ← __.chaine
Pour i de n-1+k à 0+k, faire:
    $c_i \leftarrow b_{i-k}$
Pour i de k-1 à 0, faire:
    $c_i \leftarrow 0$
res.chaine = $c_{n-1+k}c_{n-2} \dots c_1c_0$
Sortie : res

Exercice 1

Calculer le nombre maximum d'instructions élémentaires nécessaires pour chaque opération du type Entier, en fonction de la taille de l'entrée n (nombre de bits de la représentation).

+ : Chaine_bin → Chaine_bin
Entree : m
res ← Chaine_bin(0)
$b_{n-1}b_{n-2} \ldots b_1b_0$ ← __.chaine
$c_{n-1}c_{n-2} \ldots c_1c_0$ ← m.chaine
retenue = 0
Pour i de 0 à n-1, faire:
    somme ← $b_i + c_i$ + retenue
    d_i ← somme % 2
    retenue ← 1 si somme ≥ 2, 0 sinon
res.chaine = $d_{n-1}d_{n-2} \ldots d_1d_0$
Sortie: res

Exercice 1

Calculer le nombre maximum d'instructions élémentaires nécessaires pour chaque opération du type Entier, en fonction de la taille de l'entrée n (nombre de bits de la représentation).

- unaire : Ø → Chaine_bin
Entree : Ø
res ← __.complement() + Chaine_bin(1)
Sortie: res

Exercice 1

Calculer le nombre maximum d'instructions élémentaires nécessaires pour chaque opération du type Entier, en fonction de la taille de l'entrée n (nombre de bits de la représentation).

- : Chaine_bin → Chaine_bin
Entree : m
res ← __ + -(m)
Sortie: res

Exercice 1

Calculer le nombre maximum d'instructions élémentaires nécessaires pour chaque opération du type Entier, en fonction de la taille de l'entrée n (nombre de bits de la représentation).

***** : Chaine_bin → Chaine_bin
Entree : m

Donner un algorithme
Indice: utiliser les opérations : decaler et +

Sortie: res

Comparaison

- Critère de comparaison le plus fréquent: complexité en temps dans le pire des cas

- Comparaison des complexités et choix de l'algorithme pour des tailles de données très grandes.

Comparaison

- Critère de comparaison le plus fréquent: complexité en temps dans le pire des cas

- Comparaison des complexités et choix de l'algorithme pour des tailles de données très grandes.

- Exemple : deux algorithmes pour une même opération de complexités en temps : f(n) = n ! et g(n) = 2^n

n	**1**	**2**	**3**	**4**	**5**	**…**	**10**
f(n)	1	2	6	24	120	…	3628800
g(n)	2	4	8	16	32	…	1024

  - Le second algorithme est choisi (g(n) = 2^n) car g(n) croît moins vite que f(n) pour des grandes valeurs de n (tendant vers l'infini).
  - Pour des petites valeurs de n, la différence entre les deux fonctions est insignifiante.

Exemple de comparaison

- Algorithmes pour trouver la valeur maximum dans un tableau d'entiers

```
Algo A : Tableau de n entiers → Entier
    Entree : tab
    max ← tab[0]
    Pour i ← 1 à n-1
        Si max < tab[i]
            max ← tab[i]
    Sortie : max
```

```
Algo B : Tableau de n entiers → Entier
    Entree : tab
    Pour i ← 0 à n-1
        est_max ← Vrai
        Pour j ← 0 à n-1
            Si tab[i] < tab[j]
                est_max ← Faux
        Si est_max == Vrai
            max ← tab[i]
            Arret
    Sortie : max
```

Pour une instruction de complexité f(n) se trouvant dans une boucle qui se répète k fois, on compte k x f(n) instructions élémentaires.

Exemple de comparaison

- Algorithmes pour trouver la valeur maximum dans un tableau d'entiers

```
Algo A : Tableau de n entiers → Entier
 1    Entree : tab
(n-1) max ← tab[0]
      Pour i ← 1 à n-1
 x 1
  x 1   Si max < tab[i]
                max ← tab[i]
      Sortie : max
```

a(n) = 2n -1

```
Algo B : Tableau de n entiers → Entier
      Entree : tab
      Pour i ← 0 à n-1
            est_max ← Vrai
            Pour j ← 0 à n-1
                  Si tab[i] < tab[j]
                        est_max ← Faux
            Si est_max == Vrai
                  max ← tab[i]
                  Arret
      Sortie : max
```

Pour une instruction de complexité f(n) se trouvant dans une boucle qui se répète k fois, on compte k x f(n) instructions élémentaires.

Exemple de comparaison

- Algorithmes pour trouver la valeur maximum dans un tableau d'entiers

```
Algo A : Tableau de n entiers → Entier
1     Entree : tab
(n-1) max ← tab[0]
      Pour i ← 1 à n-1
x 1     Si max < tab[i]
x 1         max ← tab[i]
      Sortie : max
```

a(n) = 2n -1

```
Algo B : Tableau de n entiers → Entier
n     Entree : tab
x 1   Pour i ← 0 à n-1
x n     est_max ← Vrai
x 1     Pour j ← 0 à n-1
x 1       Si tab[i] < tab[j]
x 1           est_max ← Faux
x 1   Si est_max == Vrai
x 1       max ← tab[i]
          Arret
      Sortie : max
```

b(n) = $2n^2 + 5n$

Pour une instruction de complexité f(n) se trouvant dans une boucle qui se répète k fois, on compte k x f(n) instructions élémentaires.

Exemple de comparaison

- Algorithmes pour trouver la valeur maximum dans un tableau d'entiers

```
Algo A : Tableau de n entiers → Entier
 1    Entree : tab
(n-1) max ← tab[0]
      Pour i ← 1 à n-1
 x 1
  x 1   Si max < tab[i]
                max ← tab[i]
      Sortie : max
```

a(n) = 2n -1

Algo A est choisi car a(n) croît moins vite que b(n)

```
Algo B : Tableau de n entiers → Entier
 n    Entree : tab
 x 1  Pour i ← 0 à n-1
 x n      est_max ← Vrai
          Pour j ← 0 à n-1
      x 1
      x 1     Si tab[i] < tab[j]
 x 1                  est_max ← Faux
 x 1      Si est_max == Vrai
 x 1           max ← tab[i]
               Arret
      Sortie : max
```

$b(n) = 2n^2 + 5n$

Pour une instruction de complexité f(n) se trouvant dans une boucle qui se répète k fois, on compte k x f(n) instructions élémentaires.

Exemple de comparaison

- Algorithmes pour rechercher la position d'une valeur donnée dans un tableau d'entiers trié

```
Algo A : Entier x Tableau de n
entiers              → Index (Entier)
      Entree : x, tab
n     Pour i ← 0 à n-1
  x 1     Si tab[i] == x
  x 1           position ← i
      Sortie : position
```

a(n) = 2n

```
Algo B : Entier x Tableau de n
entiers              → Index (Entier)
      Entree : x, tab
      pas_trouve ← Vrai
      debut ← 0
      fin ← n-1
      milieu  ← (debut+fin)/2
      Tant que pas_trouve == Vrai
            Si tab[milieu] == x
                  position ← milieu
                  pas_trouve ← Faux
            Sinon Si x < tab[milieu]
                  fin ← milieu -1
            Sinon
                  debut ← milieu +1
            milieu  ← (debut+fin)/2
```

Exemple de comparaison

- Algorithmes pour rechercher la position d'une valeur donnée dans un tableau d'entiers trié

```
Algo A : Entier x Tableau de n
entiers              → Index (Entier)
      Entree : x, tab
n     Pour i ← 0 à n-1
  x 1     Si tab[i] == x
  x 1           position ← i
      Sortie : position
```

a(n) = 2n

```
Algo B : Entier x Tableau de n
entiers              → Index (Entier)
      Entree : x, tab
      pas_trouve ← Vrai
      debut ← 0
      fin ← n-1
      milieu  ← (debut+fin)/2
      Tant que pas_trouve == Vrai
            Si tab[milieu] == x
                  position ← milieu
                  pas_trouve ← Faux
            Sinon Si x < tab[milieu]
                  fin ← milieu -1
            Sinon
                  debut ← milieu +1
            milieu  ← (debut+fin)/2
```

Exemple de comparaison

- Algorithmes pour rechercher la position d'une valeur donnée dans un tableau d'entiers trié

```
Algo A : Entier x Tableau de n entiers → Index (Entier)
        Entree : x, tab
n       Pour i ← 0 à n-1
x 1       Si tab[i] == x
x 1           position ← i
        Sortie : position
```

a(n) = 2n

b(n) = 8log(n) +4

```
Algo B : Entier x Tableau de n entiers → Index (Entier)
        Entree : x, tab
1       pas_trouve ← Vrai
1       debut ← 0
1       fin ← n-1
1       milieu ← (debut+fin)/2
log(n)  Tant que pas_trouve == Vrai
log(n) x 1    Si tab[milieu] == x
log(n) x 1        position ← milieu
log(n) x 1        pas_trouve ← Faux
log(n) x 1    Sinon Si x < tab[milieu]
log(n) x 1        fin ← milieu -1
              Sinon
log(n) x 1        debut ← milieu +1
log(n) x 1    milieu ← (debut+fin)/2
```

Exemple de comparaison

- Algorithmes pour rechercher la position d'une valeur donnée dans un tableau d'entiers trié

```
Algo A : Entier x Tableau de n entiers → Index (Entier)
        Entree : x, tab
n       Pour i ← 0 à n-1
n x 1       Si tab[i] == x
n x 1           position ← i
 1      Sortie : position
```

a(n) = 3n +1

b(n) = 8log(n) +4

Algo B est choisi car b(n) croît moins vite que a(n)

```
Algo B : Entier x Tableau de n entiers → Index (Entier)
        Entree : x, tab
 1      pas_trouve ← Vrai
 1      debut ← 0
 1      fin ← n-1
 1      milieu ← (debut+fin)/2
log(n)  Tant que pas_trouve == Vrai
 x 1        Si tab[milieu] == x
 x 1            position ← milieu
 x 1            pas_trouve ← Faux
 x 1        Sinon Si x < tab[milieu]
 x 1            fin ← milieu -1
            Sinon
 x 1            debut ← milieu +1
 x 1        milieu ← (debut+fin)/2
```

Complexité log(n) ?

```
Algo B : Entier x Tableau de n entiers  → Index (Entier)
        Entree : x, tab
 1      pas_trouve ← Vrai
 1      debut ← 0
 1      fin ← n-1
 1      milieu  ← (debut+fin)/2
log(n)  Tant que pas_trouve == Vrai
 x 1        Si tab[milieu] == x
 x 1            position ← milieu
 x 1            pas_trouve ← Faux
 x 1        Sinon Si x < tab[milieu]
 x 1            fin ← milieu -1
            Sinon
 x 1            debut ← milieu +1
 x 1        milieu  ← (debut+fin)/2
```

Pour une boucle dont la taille n'est pas un multiple de n, on choisit une variable dont on peut évaluer la suite des valeurs au cours de l'exécution de la boucle.
Taille de la boucle = taille de la suite des valeurs.

Complexité log(n) ?

- La taille de la boucle Tant que est divisée par 2 à chaque itération jusqu'à une taille égale à 1, donc:

  $\underbrace{n,\quad n/2,\quad n/4,\quad n/8,\ \ldots\ ,\ 1}_{k \text{ valeurs}}$

  soit:

  $n/2^0,\ n/2^1,\ n/2^2,\ n/2^3,\ \ldots\ ,\ n/2^k$

Pour une boucle dont la taille n'est pas un multiple de n, on choisit une variable dont on peut évaluer la suite des valeurs au cours de l'exécution de la boucle.
Taille de la boucle = taille de la suite des valeurs.

```
Algo B : Entier x Tableau de n entiers  → Index (Entier)
        Entree : x, tab
 1      pas_trouve ← Vrai
 1      debut ← 0
 1      fin ← n-1
 1      milieu  ← (debut+fin)/2
log(n)  Tant que pas_trouve == Vrai
 x 1        Si tab[milieu] == x
 x 1            position ← milieu
 x 1            pas_trouve ← Faux
 x 1        Sinon Si x < tab[milieu]
 x 1            fin ← milieu -1
            Sinon
 x 1            debut ← milieu +1
 x 1        milieu  ← (debut+fin)/2
```

Complexité log(n) ?

- La taille de la boucle Tant que est divisée par 2 à chaque itération jusqu'à une taille égale à 1, donc:

  $\underbrace{n,\quad n/2,\quad n/4,\quad n/8,\ \ldots\ ,\ 1}_{k \text{ valeurs}}$

  soit:

  $n/2^0, n/2^1, n/2^2, n/2^3, \ldots, n/2^k$

  donc: $n/2^k = 1 \rightarrow 2^k = n$

➔ $k = \log(n)$

➔ Taille de la boucle = log(n)

Pour une boucle dont la taille n'est pas un multiple de n, on choisit une variable dont on peut évaluer la suite des valeurs au cours de l'exécution de la boucle.
Taille de la boucle = taille de la suite des valeurs.

```
Algo B : Entier x Tableau de n entiers  → Index (Entier)
        Entree : x, tab
 1      pas_trouve ← Vrai
 1      debut ← 0
 1      fin ← n-1
 1      milieu ← (debut+fin)/2
log(n)  Tant que pas_trouve == Vrai
 x 1        Si tab[milieu] == x
 x 1            position ← milieu
 x 1            pas_trouve ← Faux
 x 1        Sinon Si x < tab[milieu]
 x 1            fin ← milieu -1
            Sinon
 x 1            debut ← milieu +1
 x 1        milieu ← (debut+fin)/2
```

Comparaison : Notation Grand O

- Pour comparer la croissance de deux fonctions au voisinage de l'infini

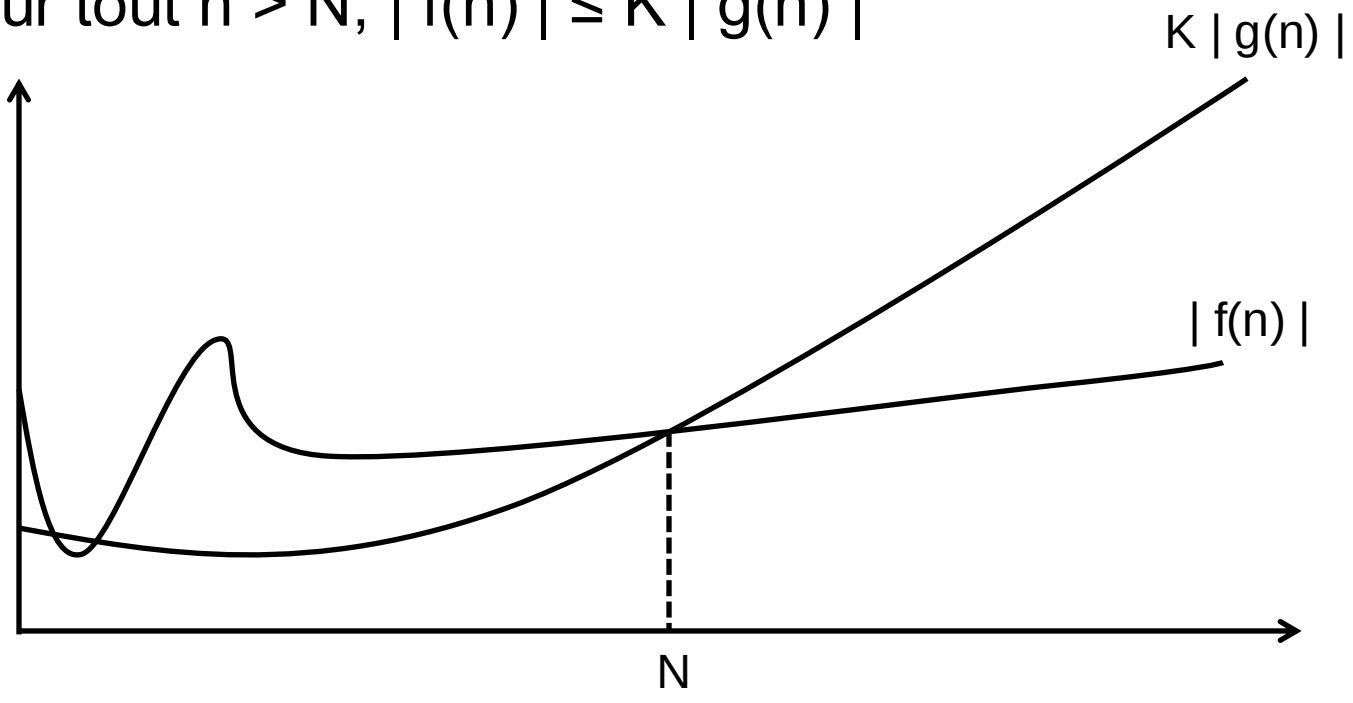
- f(n) = O(g(n)) si g croît au moins aussi vite que f

Comparaison : Notation Grand O

- Pour comparer la croissance de deux fonctions au voisinage de l'infini

- f(n) = O(g(n)) si g croît au moins aussi vite que f

- f(n) = O(g(n)) si il existe deux constantes K et N telles que pour tout n > N, | f(n) | ≤ K | g(n) |



Comparaison : Notation Grand O

- Pour comparer la croissance de deux fonctions au voisinage de l'infini

- f(n) = O(g(n)) si g croît au moins aussi vite que f

- f(n) = O(g(n)) si il existe deux constantes K et N telles que pour tout n > N, $| f(n) | \leq K | g(n) |$

- Exemple:

  $n+1 = O(n)$: pour tout $n > 1$, $n+1 \leq 2 . n$

  $n = O(n+1)$: pour tout $n > 0$, $n \leq 1 . (n+1)$

  $n^2+2n+4 = O(n^2)$: pour tout $n > 1$, $n^2+2n+4 \leq 7 . n^2$

  $n^2 = O(n^2+2n+4)$: pour tout $n > 0$, $n^2 \leq 1 . (n^2+2n+4)$

Exercice 2

- Démontrez les relations O suivantes:

$2^{10} \times n = O(n+2)$
$n^2 + 2n = O(n^2)$
$n^2 + 10n + 1 = O(n^2+6n)$
$n^4 = O(n^2+n^4)$
$n^4+5n^4+n+1 = O(n^4)$

Ordres de complexité

- Le Grand O définit une relation de semi-ordre (réflexive et transitive) sur l'ensemble des fonctions, à partir de laquelle une relation d'équivalence est définie.

- Deux fonctions f(n) et g(n) sont dans la même classe d'équivalence si f(n) = O(g(n)) et g(n) = O(f(n))

  Exemple : n et n+1 ; n^2 et n^2+2n+4 ;

  Pour n très grand, la différence entre f(n) et g(n) est négligeable

Ordres de complexité

- Les fonctions sont donc regroupées par ordres de complexité similaires (classes d'équivalence) et un représentant est choisi pour chaque classe

  Exemple : n, n+1, 3n, 5n + 10… appartiennent à la classe représentée par f(n) = n.

Ordres de complexité

- Ordres de complexité usuels, ordonnés du plus petit au plus grand:
  - $O(1)$: constant
  - $O(\log(n))$: logarithmique
  - $O(n)$: linéaire
  - $O(n\log(n))$: sub-quadratique
  - $O(n^2)$: quadratique
  - $O(n^3)$: cubique
  - $O(n^C)$: polynomial (exemple: n^4, n^5, …)
  - $O(C^n)$: exponentiel (exemple 2^n, 3^n, …)
  - $O(n!)$: factoriel

Ordres de complexité

- Ordres de complexité usuels, ordonnés du plus petit au plus grand:

  O(1) : constant
  O(log(n)) : logarithmique
  O(n) : linéaire
  O(nlog(n)) : sub-quadratique
  $O(n^2)$: quadratique
  $O(n^3)$: cubique
  $O(n^C)$: polynomial (exemple: n^4, n^5, …)
  $O(C^n)$: exponentiel (exemple 2^n, 3^n, …)
  O(n!) : factoriel

- En pratique pour la complexité dans le pire des cas:

  Instruction élémentaire ➔ O(1)
  Boucle « pour » ➔ O(n)
  k boucles « pour » imbriquées ➔ n^k

Combinaison de complexités

- Un fonction f peut être une combinaison de fonctions plus simples. On peut exprimer la complexité de f en fonction des complexités de ces fonctions plus simples.

- Si f1(n) = O(g1(n)) et f2(n) = O(g2(n))

  - Produit : f(n) = f1(n).f2(n) ➔ f(n) = O(g1(n).g2(n))

Combinaison de complexités

- Un fonction f peut être une combinaison de fonctions plus simples. On peut exprimer la complexité de f en fonction des complexités de ces fonctions plus simples.

- Si f1(n) = O(g1(n)) et f2(n) = O(g2(n))

  - Produit : f(n) = f1(n).f2(n) ➔ f(n) = O(g1(n).g2(n))

  - Somme : f(n) = f1(n) + f2(n) ➔ f(n) = O($\max_O$(g1(n),g2(n))) (maximum en termes de Grand O)

Combinaison de complexités

- Un fonction f peut être une combinaison de fonctions plus simples. On peut exprimer la complexité de f en fonction des complexités de ces fonctions plus simples.

- Si f1(n) = O(g1(n)) et f2(n) = O(g2(n))

  - Produit : f(n) = f1(n).f2(n) ➔ f(n) = O(g1(n).g2(n))

  - Somme : f(n) = f1(n) + f2(n) ➔ f(n) = O(max_O(g1(n),g2(n))) (maximum en termes de Grand O)

  - Multiplication par const : f(n) = K . f1(n) ➔ f(n) = O(g1(n))

Combinaison de complexités

- Un fonction f peut être une combinaison de fonctions plus simples. On peut exprimer la complexité de f en fonction des complexités de ces fonctions plus simples.

- Si f1(n) = O(g1(n)) et f2(n) = O(g2(n))

  - Produit : f(n) = f1(n).f2(n) ➔ f(n) = O(g1(n).g2(n))

  - Somme : f(n) = f1(n) + f2(n) ➔ f(n) = O(max_O(g1(n),g2(n))) (maximum en termes de Grand O)

  - Multiplication par const : f(n) = K . f1(n) ➔ f(n) = O(g1(n))

- Exemple:
  - $4n^3 + 2n^2 + 20 = O(n^3)$
  - $n^3 + 5\ n\ \log(n)^2 = O(n^3)$
  - $n + n\ \log(2^n) = O(n^2)$
  - $2^n + 10! = O(2^n)$