



Complexité

*"Nos études ont montré que la probabilité
qu'un programme corrigé fonctionne comme avant la correction
est seulement de cinquante pour cent."*

Bev Littlewood & Lorenzo Strigini



Efficacité des algorithmes

2


- Un bon programme tend à être le plus efficace possible
- Plusieurs critères d'efficacité
 - Temps d'exécution,
 - Quantité d'espace mémoire utilisée
 - Quantité d'informations transférée
 - ...



Analyse d'un algorithme


3

- Mesure la quantité de ressources utilisées
- Cependant
 - Temps d'exécution dépend de l'ordinateur
 - Espace mémoire peut dépendre du compilateur
- \Rightarrow Trouver des fonctions indépendantes de ces points
- Fonction de la taille du problème: $f(n)$




L'analyse pessimiste

- Les performances des algo ne sont pas toujours identiques
 - dépendent des données qui leur sont passées
 - 3 cas peuvent se présenter
 - le meilleur
 - le pire
 - entre les deux
 - Exemple: recherche linéaire
 - meilleur: l'élément recherché est le 1er
 - pire: l'élément recherché est le dernier (ou n'existe pas)
 - cas moyen: entre le 1er et le dernier



L'analyse pessimiste


- Pourquoi l'analyse pessimiste ?
Pourquoi analyse-t-on, le plus souvent, les algorithmes dans les pires situations ?
 - de nombreux algorithmes fonctionnent, la plupart du temps, dans la situation la plus mauvaise pour eux
 - Exemple, le pire cas d'une recherche est lorsqu'on ne trouve pas l'élément cherché (souvent avec manipulation de BD)



L'analyse pessimiste

- Pourquoi l'analyse pessimiste ?
 - souvent, fonctionnement dans la pire situation
 - le cas optimal apporte peu d'informations
 - beaucoup d'algorithmes font exactement la même chose dans une telle situation


7



L'analyse pessimiste

- Pourquoi l'analyse pessimiste ?
 - souvent, fonctionnement dans la pire situation
 - le cas optimal apporte peu d'informations
 - déterminer la performance dans le cas moyen est difficile
 - Qu'est exactement le cas moyen pour notre algorithme ?


8



L'analyse pessimiste

- Pourquoi l'analyse pessimiste ?
 - souvent, fonctionnement dans la pire situation
 - le cas optimal apporte peu d'informations
 - performance dans le cas moyen difficile à évaluer
 - analyse pessimiste donne une borne supérieure de la performance
 - garantit que l'algorithme ne fera jamais moins bien que ce que nous calculons
 - les autres cas se dérouleront au moins aussi bien

9



Complexité maximale

- Quelle précision pour la complexité ?
- Lors d'une étude, nous pouvons arriver à la conclusion que la complexité maximale du programme est $T_{\max}(n)=3n^2+10n+10$.
(on a ainsi le nombre exact d'instructions élémentaires utiles pour le programme)
- Une telle précision est inutile, l'ordre de grandeur est suffisant.



Complexité maximale

10

- Pourquoi l'ordre de grandeur est-il suffisant ?
 - Exemple: complexité max $T_{\max}=3n^2+10n+10$
 - Si la taille des données est $n=10$, on a
 - tps d'exécution de $3n^2$: $(3*10^2)/(3*10^2+10*10+10)=73.2\%$
 - tps d'exécution de $10n$: $(10*10)/(3*10^2+10*10+10)=24.4\%$
 - tps d'exécution de 10 : $(10)/(3*10^2+10*10+10)=2.4\%$
 - Si la taille des données est $n=100$, on a
 - tps d'exécution de $3n^2$: $(3*100^2)/(3*100^2+10*100+10)=96.7\%$
 - tps d'exécution de $10n$: $(10*100)/(3*100^2+10*100+10)=3.2\%$
 - tps d'exécution de 10 : $(10)/(3*100^2+10*100+10)<0.1\%$
 - Le terme important est **$3n^2$**



Le grand O

11

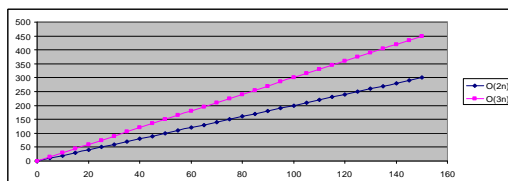
- Sachant que n est le paramètre donnant la taille des données, nous voulons déterminer une fonction simple $f(n)$ qui, à une constante près, borne la complexité de l'algorithme.
- Une complexité est dite "en grand O de $f(n)$ "
- Exemple: $T_{\max}(n)=3n^2+10n+10$
est en **$O(3n^2)$** \equiv **$O(n^2)$**



Règles simples de la notation O

12

- Termes constants de la forme $O(1)$
 - $O(c)=O(1)$
- Constantes multiplicatives omises
 - $O(c.n)=c.O(n)=O(n)$



13

Règles simples de la notation O

- Termes constants de la forme $O(1)$
 - $O(c)=O(1)$
- Constantes multiplicatives omises
 - $O(c \cdot n)=O(n)$
- Addition
 - $O(n) + O(m) = \max\{O(n), O(m)\}$

14

Règles simples de la notation O

- Termes constants de la forme $O(1)$
 - $O(c)=O(1)$
- Constantes multiplicatives omises
 - $O(c \cdot n)=c \cdot O(n)=O(n)$
- Addition réalisée en prenant le maximum
 - $O(n) + O(m) = \max\{O(n), O(m)\}$
- Multiplication reste inchangée (souvent réécrite de façon plus compacte)
 - $O(n)O(m)=O(nm)$

15

Grandes classes de complexité

Grand O	Classe
$O(1)$	constante
$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \log n)$	$n \log n$
$O(n^2)$	quadratique
$O(n^3)$	cubique
...	...
$O(2^n)$	exponentiel en 2^n
$O(3^n)$	exponentiel en 3^n
$O(n^n)$	exponentiel en n^n

polynomial

exponentiel

5



Calcul d'une complexité

16

- Utilise la structure du programme: l' "arbre syntaxique" du programme
- 6 règles simples



Règle 1: Unité

17

- Définir l'**unité** utilisée
- Exemple
 - Temps d'exécution $O(1)$
 - expression simple ($i \leftarrow i+1$; $a \leftarrow \text{tab}[i]$; ...)
 - instructions lecture / écriture
 - retour de fonction



Règle 2: séquence

18

- La complexité d'une séquence de plusieurs instructions est le maximum des complexités de chaque instruction

Traitement	instruction 1	instruction 2	séquence
$O()$	$O(f_1(n))$	$O(f_2(n))$	$O(\max\{f_1(n), f_2(n)\})$
Exemple	$O(n^2)$	$O(n)$	$O(n^2)$

19

Règle 3: si alors sinon

- La complexité d'un *si condition alors instruct1 sinon instruct2 fin si* est le maximum des complexités de condition, instruct1 et instruct2

Traitement	condition	instruct1	instruct2	si
Exemple	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$

20

Règle 4: Tantque...

- La complexité d'un *Tantque condition faire instructions Fintq* est la complexité du nombre d'itérations * le maximum des complexités de condition et instructions

Traitement	Nb itérat.	condition	instructions	Tantque
Exemple	$O(n)$	$O(1)$	$O(n^2)$	$O(n^3)$

Répéter...Jusqu'à similaire au Tantque

21

Règle 5: Pour...

Pour i allant de a à b faire instructions; finpour

traduit en

```

i ← a;
Tantque i ≤ b faire
  instructions;
  i ← i + 1;
Fintantque
  
```



Règle 6: fonction

- L'appel à une fonction de complexité $O(n)$ est en $O(n')$ où n' est la taille des paramètres effectifs
- Exemple: $x \leftarrow \text{minimum}(V)$
 - minimum en $O(n)$ où n est la taille du vecteur V
 - V de taille m
 - Assignment: en $O(m)$



Arbre syntaxique (1/2)

```
Fonction Minimum(val V[],n:entier):entier
Var i,Min:entier
Début
Min ← V[0];
Pour i allant de 1 à n-1 faire
  Si (V[i]<Min) alors
    Min ← V[i];
  Finsi
Finpour
Retourner(Min);
Fin
```

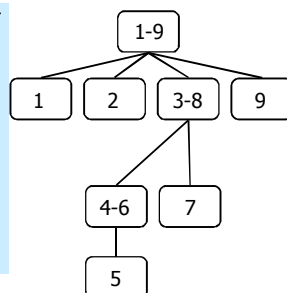


```
Fonction Minimum(val V[],n:entier):entier
Var i,Min:entier
Début
(1) Min ← V[0];
(2) i ← 1;
(3) Tantque i<n faire
(4)   Si (V[i]<Min) alors
(5)     Min ← V[i];
(6)   Finsi
(7)   i ← i + 1;
(8) Fintantque
(9) Retourner(Min);
Fin
```



Arbre syntaxique (2/2)

```
Fonction Minimum(val V[],n:entier):entier
Var i,Min:entier
Début
(1) Min ← V[0];
(2) i ← 1;
(3) Tantque i<n faire
(4)   Si (V[i]<Min) alors
(5)     Min ← V[i];
(6)   Finsi
(7)   i ← i + 1;
(8) Fintantque
(9) Retourner(Min);
Fin
```



25

Fonction Minimum...

```

Fonction Minimum(val V[],n:entier):entier
Var i,Min:entier
Début
(1) Min ← V[0];
(2) i ← 1;
(3) Tantque i < n faire
(4)   Si (V[i] < Min) alors
(5)     Min ← V[i];
(6)   Finsi
(7)   i ← i + 1;
(8) Fintantque
(9) Retourner(Min);
Fin

```

26

Remarques


- 2 algorithmes peuvent avoir la même complexité mais l'un peut être plus rapide que l'autre

Pour i allant de 1 à n faire x ← x + 1; Finpour	Pour i allant de 1 à 100n faire x ← x + 1; Finpour
---	--

27


Remarques

- pour un problème, on peut parler de complexité en temps et de complexité en espace mémoire:
 - Un problème peut être de complexité
 - $O(n^2)$ en temps
 - $O(n)$ en espace mémoire
 - Cela signifie qu'il existe 2 algorithmes A et B pour résoudre le problème (peut-être différents) tels que
 - A a un temps de $O(n^2)$
 - B utilise un espace mémoire en $O(n)$



28

Exercices




29

Calculs de complexité (1)

Procédure p1(val t[]:entier, n:entier)
var i:entier;
Début
 pour i allant de 0 à n-1 faire
 t[i] ← 0;
 finpour
Fin

- t[i] ← 0 ⇒ O(1)
- on parcourt 1 fois le tableau de taille n
⇒ n*O(1)
⇒ O(n)



30

Calculs de complexité (2)

Procédure p2(val t[]:entier, k:entier)
var i:entier;
Début
 pour i allant de 0 à k-1 faire
 t[i] ← 0;
 finpour
 pour i allant de 0 à k-1 faire
 t[i] ← i+1;
 finpour
Fin

- t[i] ← 0, i+1 et t[i] ← i+1
⇒ O(1)
- on parcourt une 1ère fois le tableau de taille k
⇒ k*O(1)
⇒ O(k)
- on parcourt une 2ème fois le tableau de taille k
⇒ k*O(1)
⇒ O(k)
- ⇒ O(k)+O(k) = O(k)



Calculs de complexité (3)

```
Procédure p1(val t[]:entier, n:entier)
var i:entier;
Début
  pour i allant de 0 à n-1 faire
    t[i] ← 0;
  finpour
Fin
```

Quelle est la complexité de p2?

- $p1 \Rightarrow O(n)$
- Dans p2, on appelle 1 fois la procédure p1, qui a une complexité dépendant de la taille des données. Le tableau envoyé est de taille k, $\Rightarrow O(k)$

```
Procédure p2(val t[]:entier, k:entier)
var i:entier;
Début
  p1(t,k);
Fin
```



Calculs de complexité (4)

```
k ← 0;
pour i allant de 1 à n faire
  pour j allant de 1 à i faire
    k ← k+i*j;
  finpour
finpour
```

- $k \leftarrow 0$ et $k \leftarrow k+i*j \Rightarrow O(1)$
- pour j...: i itérations à chaque fois
- pour i...: n itérations
- $\Rightarrow 1+2+3+\dots+n$ itérations
 $= n(n+1)/2$ itérations
- \Rightarrow le "pour" global: $O(n^2)$
- Algo complet: $O(n^2)$



Calculs de complexité (5)

```
k ← 0;
i ← 1;
tantque i < n faire
  pour j allant de 1 à n faire
    k ← k*2;
  finpour
  i ← i*2;
fintantque
```

- $k \leftarrow 0, i \leftarrow 0, k \leftarrow k*2$ et $i \leftarrow i*2 \Rightarrow O(1)$
- pour j...: n itérations à chaque fois
- tantque...:
 - i varie de la façon suivante: 1,2,4,8,...,2^p
 - arrêt: i=n quand $p = \log_2(n)$
 - \Rightarrow tantque...: $\log_2(n)$ itérations
- $\Rightarrow n \cdot \log_2(n)$ fois l'affectation $k \leftarrow k*2$
- \Rightarrow tantque...: $O(n \cdot \log_2(n))$
- Algo complet: $O(n \cdot \log_2(n))$

Calculs de complexité (6) (4+5)

```

k ← 0;
pour i allant de 1 à n faire
  pour j allant de 1 à i faire
    k ← k+i*j;
  finpour
finpour
i ← 1;
tantque i < n faire
  pour j allant de 1 à n faire
    k ← k*2;
  finpour
  i ← i*2;
fintantque
  
```

Complexity analysis:

- The first loop (for i) has complexity $O(n^2)$.
- The second loop (while i < n) has complexity $O(n \cdot \log_2(n))$.
- The overall complexity is $?$.

Calculs de complexité (6) (4+5)

```

k ← 0;
pour i allant de 1 à n faire
  pour j allant de 1 à i faire
    k ← k+i*j;
  finpour
finpour
i ← 1;
tantque i < n faire
  pour j allant de 1 à n faire
    k ← k*2;
  finpour
  i ← i*2;
fintantque
  
```

Complexity analysis:

- The first loop (for i) has complexity $O(n^2)$.
- The second loop (while i < n) has complexity $O(n \cdot \log_2(n))$.
- The overall complexity is $O(n^2)$.

Calculs de complexité (7)

```

i ← 1;
tantque i < n faire
  j ← 1;
  tantque j < n faire
    si(i==j) alors
      pour k allant de 1 à n faire
        Ecrire("test");
      finpour
    finsi
    j ← j*2;
  fintantque
  i ← i*2;
fintantque
  
```

- tantque i...: $\log_2(n)$ itérations
- tantque j...: $\log_2(n)$ itérations
- pour k...: $O(n)$
 ⇒ si...: $O(n)$
- le "si" est effectué $\log_2^2(n)$ fois
- mais ne sera valide que si $i=j$ donc $\log_2(n)$ fois
- ⇒ algorithme: $O(n \cdot \log_2(n))$

37

Calculs de complexité (8)

```

Fonction f1(val t[],n,position:entier):entier
Début
  SI position<n-1 ALORS
    retourner(t[position]+f1(t,n,position+1));
  SINON
    retourner(t[position]);
Fin

var tab[n],res,i:entier;
Début
  res <- 0;
  POUR i allant de 0 à n-1 FAIRE
    tab[i] <- i+1;
  FINPOUR
  POUR i allant de 0 à n-1 FAIRE
    res <- res + f1(tab,n,i);
  FINPOUR
  Ecrire("res="+res);
Fin

```

Complexité?

- Complexité de f1()
 - chaque passage dans f1: $O(1)$
 - appels récursifs: dépend du nombre d'appels
- Algorithme
 - 1ère boucle (initialisation): $O(n)$
 - 2ème boucle: n itérations
 - A chaque itération i, la fonction f1() est appelée, et exécute (n-i) appels récursifs
 - $\Rightarrow 1+2+\dots+n$ appels de f1() exécutés
 - \Rightarrow complexité: $O(n^2)$

38

Calculs de complexité (9)

```

Fonction Min(val t[],n,position:entier):entier
Var min:entier;
Début
  min <- t[position];
  POUR i allant de position+1 à n-1 FAIRE
    SI t[i]<min ALORS min <- t[i]; FINSI
  FINPOUR
  Retourner(min);
Fin

var tab[n],i,j:entier;
Début
  initTab(tab,n); //place des entiers dans tab
  POUR i allant de 0 à n-1 FAIRE
    POUR j allant de 0 à n-1 FAIRE
      SI t[j]=Min(t,n,i) et j>i ALORS
        inverser(t, i, j);
      FINSI
    FINPOUR
  FINPOUR
Fin

```

Résultat et complexité?

- Complexité de Min()
 - chaque passage: $O(n)$
(une boucle de taille n-position)
- Algorithme
 - initTab(): $O(n)$
 - inverser(): $O(1)$
 - boucles
 - on fait n^2 fois un appel à Min
 - En détail:
 - n fois un appel avec une boucle de taille n
 - n fois un appel avec une boucle de taille n-1
 -
 - n fois un appel avec une boucle de taille 1
 - $\Rightarrow n(1+2+\dots+n)$ boucles
 - \Rightarrow complexité: $O(n^3)$

39

Calculs de complexité (9)

```

Fonction Min(val t[],n,position:entier):entier
Var min:entier;
Début
  min <- t[position];
  POUR i allant de position+1 à n-1 FAIRE
    SI t[i]<min ALORS min <- t[i]; FINSI
  FINPOUR
  Retourner(min);
Fin

var tab[n],i,j:entier;
Début
  initTab(tab,n); //place des entiers dans tab
  POUR i allant de 0 à n-1 FAIRE
    POUR j allant de 0 à n-1 FAIRE
      SI t[j]=Min(t,n,i) et j>i ALORS
        inverser(t, i, j);
      FINSI
    FINPOUR
  FINPOUR
Fin

```

Peut-on améliorer le résultat (avec peu de changements)?

```

var tab[n],i,j,tmp:entier;
Début
  initTab(tab,n); //place des entiers dans tab
  POUR i allant de 0 à n-1 FAIRE
    tmp <- Min(t,n,i);
    POUR j allant de 0 à n-1 FAIRE
      SI t[j]=tmp et j>i ALORS
        inverser(t, i, j);
      FINSI
    FINPOUR
  FINPOUR
Fin

```

Nouvelle complexité? $O(n^2)$



Calculs de complexité (suite)

Ecrivez un programme permettant de remplir la diagonale d'une matrice carrée de taille n avec des 1. Calculez le nombre d'opérations élémentaires à réaliser. Donnez alors la complexité en temps et en espace.

Ecrivez un programme permettant de remplir la partie triangulaire supérieure d'une matrice carrée de taille n avec des 1. Calculez le nombre d'opérations élémentaires à réaliser. Donnez alors la complexité en temps et en espace.



Calculs de complexité (suite)

On dispose des deux fonctions suivantes avec $0 < q \leq 1$, $n > 0$:

Fonction Fct1 (val q: réel, n: entier): réel Début Si (n = 0) alors retourner (1) Sinon retourner (1+q*Fct1(q, n-1)); fin Fin	Fonction Fct2 (val q: réel, n: entier): réel Var res: réel; i: entier; Début Res ← 0; Pour i allant de 0 à n faire Res ← res*q; Fin pour Retourner ((1-res)/(1-q)); fin
---	--

- 1) Que font ces fonctions ?
- 2) Calculez le nombre d'opérations à exécuter pour chacune en fonction de n ;
- 3) Calculez leur complexité en temps.
- 4) Laquelle utiliseriez vous ?
