Compléments de Programmation

Benoit Donnet
Année Académique 2023 - 2024

Agenda

- Chapitre 1: Raisonnement Mathématique
- Chapitre 2: Construction de Programme
- Chapitre 3: Introduction à la Complexité
- Chapitre 4: Récursivité
- Chapitre 5: Types Abstraits de Données
- Chapitre 6: Listes
- Chapitre 7: Piles
- Chapitre 8: Files
- Chapitre 9: Elimination de la Récursivité

Agenda

- Chapitre 3: Introduction à la Complexité
  - Principe
  - Quantification des Instructions
  - Notation de Landau
  - Exemples

Agenda

- Chapitre 3: Introduction à la Complexité
  - Principe
    - ✓ Idée
    - ✓ Définition
    - ✓ Fonctionnement
  - Quantification des Instructions
  - Notation de Landau
  - Exemples

Idée

- Soient
  - $\mathscr{P}$, un problème
  - $\mathscr{M}$, une méthode pour résoudre le problème $\mathscr{P}$
- Un **<u>programme</u>** est la description de $\mathscr{M}$ dans un langage de programmation

Idée (2)

- On veut
  - évaluer l'efficacité de la méthode $\mathscr{M}$
  - comparer $\mathscr{M}$ avec une autre méthode $\mathscr{M}$ '
- Et ce, indépendamment de l'environnement
  - machine, système, compilateur, ...

Définition

- Complexité des programmes
  - Définition: étude formelle de la *quantité de ressources* nécessaire pour l'exécution d'un programme
    - ✓ **complexité spatiale**: utilisation mémoire que va nécessiter le programme
    - ✓ **complexité temporelle**: nombre d'opérations élémentaires effectuées par un programme
- On va s'intéresser (uniquement) à la complexité temporelle

Définition (2)

- La complexité est donc une évaluation du nombre d'opérations élémentaires en fonction
  - de la taille des données
  - de la nature des données
- Notations
  - n, la taille des données
  - $T(n)$, fonction représentant le nombre d'opérations élémentaires
- Configurations caractéristiques
  - meilleur cas
    - ✓ cfr. INFO0902
  - **pire des cas**
  - cas moyen
    - ✓ cfr. INFO2050

Fonctionnement

- 2 étapes pour déterminer la complexité d'un segment de code
  1. déterminer le nombre d'instructions élémentaires
     - ✓ fonction $T(n)$
  2. borner $T(n)$ pour représenter le pire des cas
     - ✓ notation de Landau
     - ✓ $O(\cdot)$

Agenda

- Chapitre 3: Introduction à la Complexité
  - Principe
  - Quantification des Instructions
    - ✓ Règles
  - Notation de Landau
  - Exemples

Règles

- Comment appliquer la complexité théorique à des programmes pour avoir une idée de leur efficacité?
- Solution
  - inventaire des instructions exécutées par le programme

Règles (2)

- Règles pour quantifier les instructions
  1. instruction de base
     - ✓ écriture à l'écran
     - ✓ lecture au clavier
     - ✓ accès à une variable
     - ✓ *T(1)*

Règles (3)

- Règles pour quantifier les instructions (cont.)
  2. séquence d'instructions
     - ✓ somme de la fonction $T(\cdot)$ de chacune des instructions

```
Traitement1;

Traitement2;
```

$$\left.\begin{array}{l} \texttt{Traitement1;} \quad T_1(\cdot) \\ \\ \texttt{Traitement2;} \quad T_2(\cdot) \end{array}\right\} T(\cdot) = T_1(\cdot) + T_2(\cdot)$$

Règles (4)

- Règles pour quantifier les instructions (cont.)
  3. structure conditionnelle `if`
     - ✓ le maximum des fonctions $T(\cdot)$ de chaque branche

```
if(expression)
   Traitement1;
else
   Traitement2;
```

$$\left.\begin{array}{l} \texttt{Traitement1;} \quad T_1(\cdot) \\ \\ \texttt{Traitement2;} \quad T_2(\cdot) \end{array}\right\} T(\cdot) = \max(T_1(\cdot), T_2(\cdot))$$

Règles (5)

- Règles pour quantifier les instructions (cont.)
  4. structure conditionnelle `switch`
     - ✓ le maximum des fonctions $T(\cdot)$ de chaque branche

```
switch(variable){
   case x1: Traitement1;        T1(·)
   case x2: Traitement2;        T2(·)
   ...
   case xi: Traitementi;        Ti(·)
   ...
   case xk: Traitementk;        Tk(·)
   default: Traitement;         Tk+1(·)
}
```

$$T(\cdot) = \max\big(T_1(\cdot), \ldots, T_k(\cdot), T_{k+1}(\cdot)\big)$$

Règles (6)

- Règles pour quantifier les instructions (cont.)
  5. structure itérative
     - ✓ fonction T(·) du Corps de Boucle × le nombre de tours

```
while(expression)
   Traitement;              Ti(·)
```

$$T(\cdot) = \sum_{i=1}^{k} T_i(\cdot)$$

Règles (7)

- Règles pour quantifier les instructions (cont.)
  6. programme complet
     - ✓ séquence d'instructions
     - ✓ cfr. Règle 2

Agenda

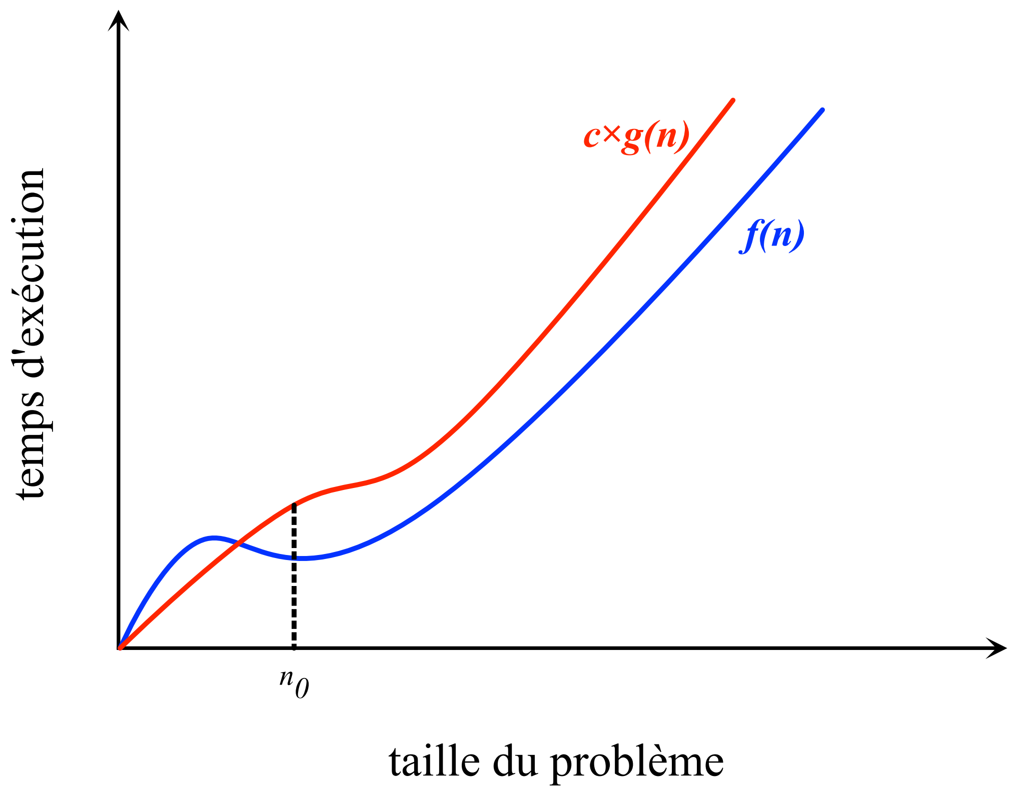
- Chapitre 3: Introduction à la Complexité
  - Principe
  - Quantification des Instructions
  - Notation de Landau
    - ✓ Notation Asymptotique
    - ✓ Propriétés
    - ✓ Exemples
    - ✓ Classification
  - Exemples

Notation Asymptotique

- Comment évaluer la complexité temporelle?
  - évaluer le nombre d'instructions élémentaires exécutées par le programme
    - ✓ fonction $T(\cdot)$
    - ✓ temps d'exécution individuel supposé borné
  - borne supérieure de $T(\cdot)$ avec la notation "O"
    - ✓ *big O*
    - ✓ **notation de Landau**
- Notation asymptotique
  - soient f et g, deux fonctions $\mathbb{N} \rightarrow \mathbb{R}^+$

$$f \in O(g) \iff \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ : \forall n > n_0 : f(n) \leq c \times g(n)$$

Notation Asymptotique (2)



Notation Asymptotique (3)

- *n* c'est quoi?
- La complexité s'exprime toujours en fonction de la taille des données
  - en fonction du problème
  - et ses variables

Propriétés

- Propriétés de la notation "O"
  - multiplication par une constante
    - ✓ si $f(n) \in O(g(n))$,
      alors $\forall\ k \in \mathbb{N}_0$ on a $k \times f(n) \in O(g(n))$
  - addition (1)
    - ✓ si $f(n), e(n) \in O(g(n))$,
      alors $e(n) + f(n) \in O(g(n))$
  - addition (2)
    - ✓ si $e(n) \in O(g(n))$ et $f(n) \in O(h(n))$
      alors $e(n) + f(n) \in O(g(n) + h(n))$

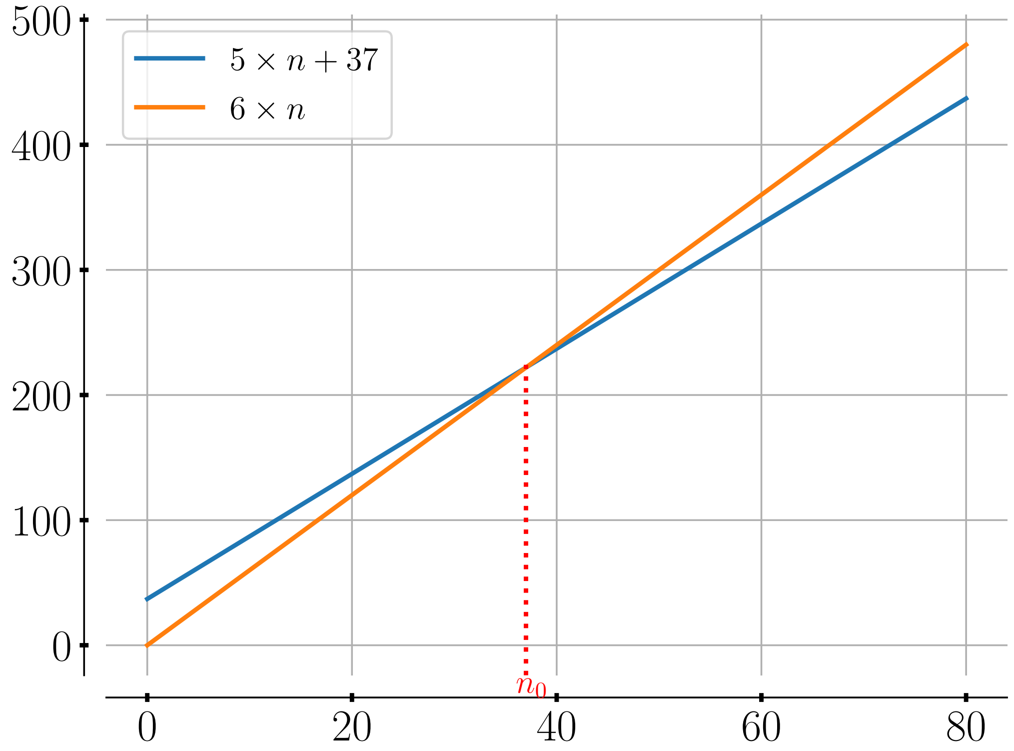
Propriétés (2)

- Propriétés de la notation "O" (suite)
  - produit
    - ✓ <u>si</u> $e(n) \in O(g(n))$ et $f(n) \in O(h(n))$
      <u>alors</u> $e(n) \times f(n) \in O(g(n) \times h(n))$
  - transitivité
    - ✓ <u>si</u> $f(n) \in O(g(n))$ et $g(n) \in O(h(n))$
      <u>alors</u> $f(n) \in O(h(n))$

Exemples

- $T(n) = 5 \times n + 37$
- Par quoi borner $T(n)$?
  - $O(n)$
- Preuve?
  - but: trouver une constante $c \in \mathbb{R}^+$ et un seuil n_0 à partir duquel $T(n_0) \leq c \times n_0$
  - on remarque que $5 \times n + 37 \leq 6 \times n$ si $n \geq 37$
    - ✓ $5 \times 37 + 37 \leq 6 \times 37$
    - ✓ $5 \times 38 + 37 \leq 6 \times 38$
    - ✓ ...
  - on déduit que $c = 6$ fonctionne à partir du seuil $n_0 = 37$
- Remarque
  - on ne demande pas d'optimisation (i.e., le plus petit c et n_0 qui fonctionnent), juste donner des valeurs
  - $c = 10$ et $n_0 = 8$ sont donc aussi acceptables

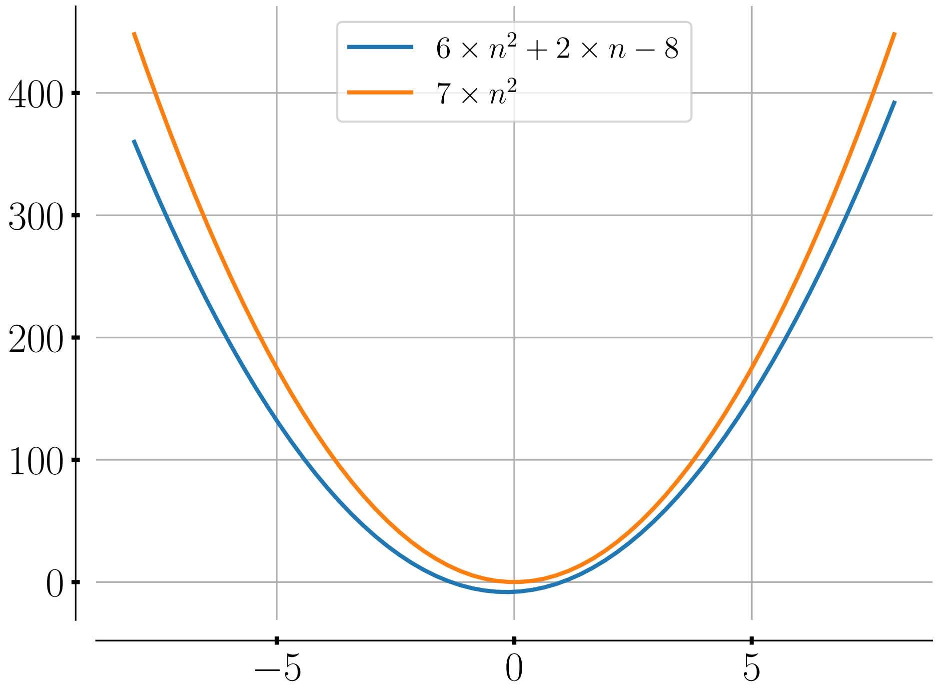
Exemples (2)



Exemples (3)

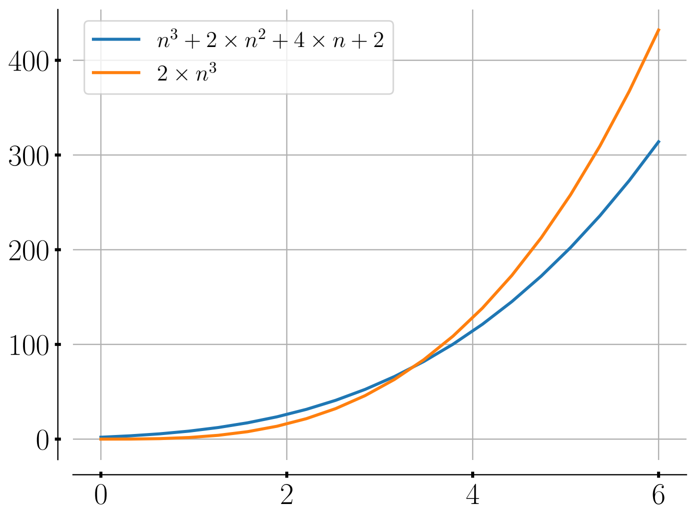
- $T(n) = 6 \times n^2 + 2 \times n - 8$
- Par quoi borner $T(n)$?
  - $O(n^2)$
- Preuve
  - cherchons d'abord une constante c
    - ✓ $c = 6 \Rightarrow$ ne fonctionne pas
    - ✓ essayons avec $c = 7$
  - trouvons un seuil n_0 à partir duquel
    - ✓ $6 \times n^2 + 2 \times n - 8 \leq 7 \times n^2, \forall\ n > n_0$
    - ✓ un simple calcul de racines nous donne
      - $n_1 = -4/3$
      - $n_2 = 1$
  - $c = 7$ et $n_0 = 1$ nous donnent le résultat voulu

Exemples (4)



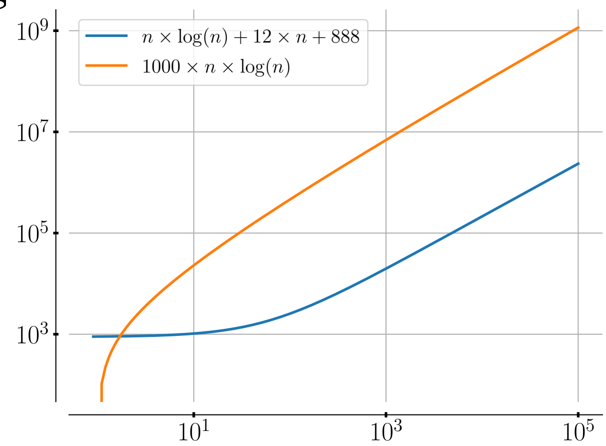
Exemples (5)

- $T(n) = n^3 + 2 \times n^2 + 4 \times n + 2$
- Par quoi borner $T(n)$?
  - $O(n^3)$
- Preuve?
  - Si $n \geq 3$
  - Alors $T(n) \leq 2 \times n^3$



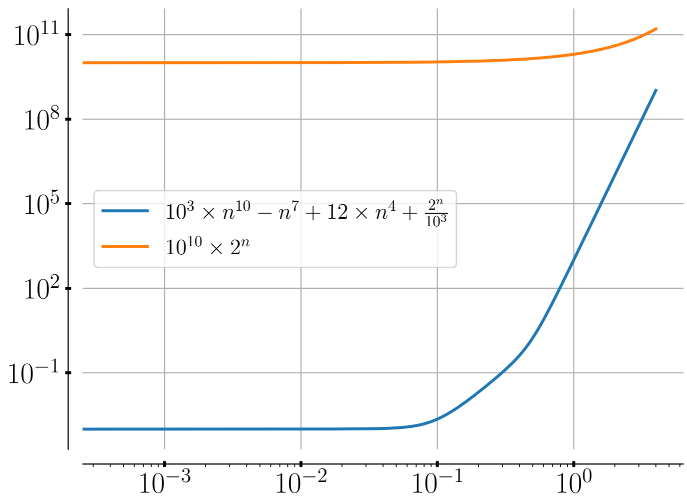
Exemples (2)

- $T(n) = n \times \log n + 12 \times n + 888$
- Par quoi borner $T(n)$?
  - $O(n \times \log n)$
- Preuve?
  - <u>Si</u> $n \geq 1$
  - <u>Alors</u> $T(n) \leq 1000 \times n \times \log n$



Exemples (3)

- $T(n) = 10^3 \times n^{10} - n^7 + 12 \times n^4 + 2^n/10^3$
- Par quoi borner $T(n)$?
  - $O(2^n)$
- Preuve?
  - <u>Si</u> $n \geq 1$
  - <u>Alors</u> $T(n) \leq 10^{10} \times 2^n$

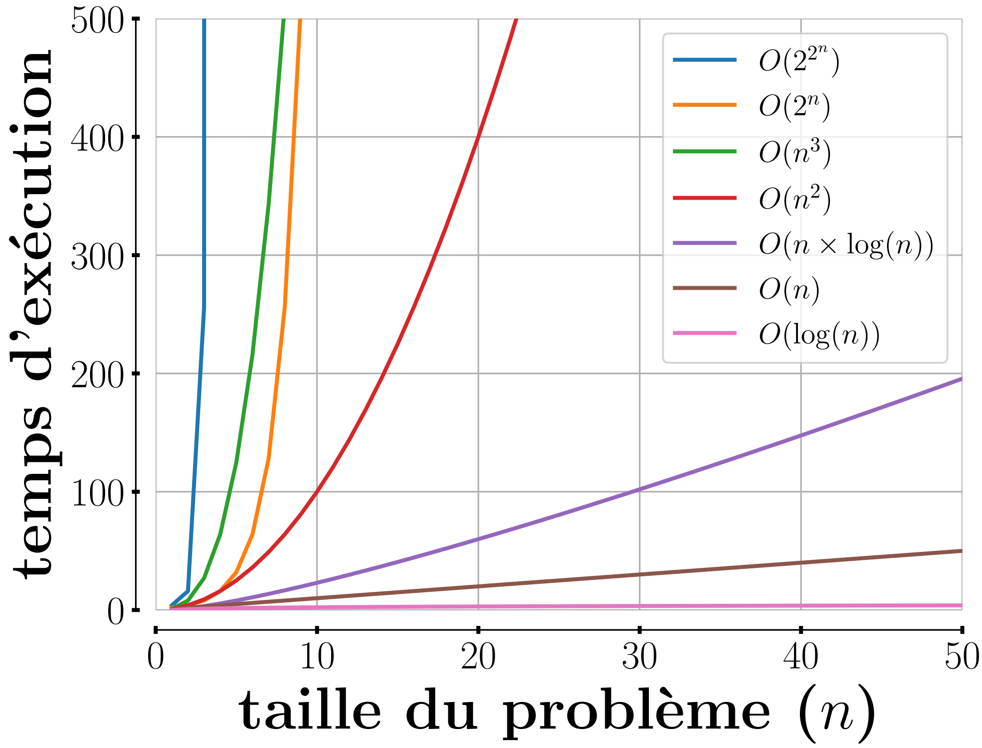


Classification

- Classification de la complexité

Notation	Type de Complexité	Exemple
$O(1)$	complexité constante	accès variable
$O(log(n))$	complexité logarithmique	dichotomie
$O(n)$	complexité linéaire	triangulation delaunay
$O(n \times log(n))$	complexité linéarithmique	tri rapide
$O(n^2)$	complexité quadratique	parcours tableau 2D
$O(n^3)$	complexité cubique	parcours tableau 3D
$O(e^n)$	complexité exponentielle	facteurs premiers
$O(n!)$	complexité factiorelle	voyageur de commerce
$O(2^{2^\wedge n})$	complexité doublement exponentielle	arithmétique de Presburger

Classification (2)



Classification (3)

- Evaluation du temps de calcul en fonction de la complexité

		Complexité			
		log(n)	**n**	n^2	2^n
flops	10^6	0,013 msec	1 sec	278 heures	10.000 ans
	10^9	0,013 µsec	1 msec	15 min	10 ans
	10^{12}	0,013 nsec	1 µsec	1sec	1 semaine

Agenda

- Chapitre 3: Introduction à la Complexité
  - Principe
  - Quantification des Instructions
  - Notation de Landau
  - Exemples
    - ✓ Permutation de 2 variables
    - ✓ Somme des *n* Premières Valeurs
    - ✓ Factorielle
    - ✓ Renversement de Chiffres
    - ✓ Nombres Parfaits (version 1)

Permutation Variables

- Exemple 1
  - permutation de deux variables

`tmp = x;`	T(A)
`x = y;`	T(B)
`y = tmp;`	T(C)

Permutation Variables (2)

- Par application de la règle 2
  - $T = T(A) + T(B) + T(C)$
- Par application de la règle 1
  - $T = 1 + 1 + 1$
  - $= 3$
- Par quoi borner T?
  - $O(1)$
  - **<u>complexité constante</u>**

Somme

- Exemple 2
  - somme des *n* premières valeurs

```
#include <stdio.h>

int main(){
   unsigned int i = 1, n, somme = 0;
   scanf("%u", &n);

   while(i<=n){
      somme += i;
      i++;
   }//fin while - i

   printf("Somme: %u\n", somme);
}//fin programme
```

T(A)

T(B)

T(C)

Somme (2)

- Par application des règles 2 & 6
  - $T(n) = T(A) + T(B) + T(C)$
- Par application de la règle 1
  - $T(n) = 1 + T(B) + 1$
- Quid de $T(B)$?
  - application de la règle 5

Somme (3)

- Evaluation de *T(B)*

```
while(i<=n){
    somme += i;
    i++;
}//fin while - i
```

T(B')
T(B'')
T(B)

- Quid de *T(B')* et *T(B'')*?
  - application de la règle 1
  - $T(B') = 1$
  - $T(B'') = 1$

Somme (4)

- Quid de *T(B)*?
  - application de la règle 5

$$T(B) = \sum_{i=1}^{n} \left(T(B') + T(B'')\right)$$
$$= \sum_{i=1}^{n} (1+1)$$
$$= 2 \times n$$

Somme (5)

- Il vient donc
  - $T(n) = T(A) + T(B) + T(C)$
    $= 1 + 2 \times n + 1$
    $= 2 \times n + 2$
- Par quoi borner $T(n)$?
  - $O(n)$
  - **complexité linéaire**

Factorielle

- Exemple 3
  - factorille de n

```
#include <stdio.h>

int main(){
  unsigned int i=1, fact=1, n;
  scanf("%u", &n);

  while(i<=n){
    fact *= i;
    i++;
  }//fin while - i

  printf("factorielle: %u\n", fact);
}//fin programme
```

T(A)

T(B)

T(C)

Factorielle (2)

- Par application des règles 2 & 6
  - $T(n) = T(A) + T(B) + T(C)$
- Par application de la règle 1
  - $T(n) = 1 + T(B) + 1$
- Quid de $T(B)$?
  - même raisonnement que pour la somme
  - $T(B) = 2 \times n$
- Il vient
  - $T(n) = 1 + 2 \times n + 1$
    $= 2 \times n + 2$
- Par quoi borner $T(n)$?
  - $O(n)$
  - **complexité linéaire**

Renversement

- Exemple 4
  - renverser les chiffres des nombres en base 10 entre 1 et *fin*
- Fonctionnement
  - $35276 \rightarrow 67253$
  - $19 \rightarrow 91$
  - $3 \rightarrow 3$
  - $0 \rightarrow 0$
- Raisonnement
  - cfr. Chap. 3

Renversement (2)

```
#include <stdio.h>

int main(){
    unsigned int i = 1, fin, n, r = 0;
    scanf("%u", &fin);

    while(i<=fin){
        r = 0;
        n = i;

        while(n > 0){
            r = 10*r + n%10;
            n /= 10;
        }//fin while - n

        printf("%u %u\n", i, r);
        i++;
    }//fin while - i
}//fin programme
```

T(A)

T(B')

T(B'') T(B)

T(B''')

Renversement (3)

- Par application des règles 2 & 6
  - $T(fin) = T(A) + T(B)$
- Par application des règles 1 & 5

$$T(fin) = T(A) + T(B)$$
$$= 1 + \sum_{i=1}^{fin} \left(T(B') + T(B'') + T(B''')\right)$$
$$= 1 + \sum_{i=1}^{fin} \left(1 + T(B'') + 1\right)$$

- Quid de $T(B'')$?

Renversement (4)

- Evaluation de *T(B'')*
  - déterminer le nombre de tours de la boucle

```
while(n>0){
  r = 10*r + n%10;
  n = n/10;
}//fin while - n
```

```
évaluation gardien 0: n = n (i.e., n = n/10⁰)
évaluation gardien 1: n = n/10 (i.e., n = (n/10⁰)/10)
évaluation gardien 2: n = n/10² (i.e., n = (n/10¹)/10)
...
évaluation gardien k: n = n/10ᵏ (i.e., n = (n/10ᵏ⁻¹)/10)
```


Renversement (5)

- Evaluation de *T(B'')* (cont.)?
- quand est-ce que la boucle s'arrête?
  - $n > 0$
  - $n/10^k > 0$ doit être satisfait
- Estimer la valeur de k?

$$\frac{n}{10^k} > 0$$

$$\frac{n}{10^k} \geq 1$$

$$n \geq 10^k$$

$$\log_{10} n \geq k$$

Renversement (6)

- Evaluation de *T(B'')* (cont.)
- Dans le pire des cas, on effectue *T(B'')* pour $n = fin$
- Donc
  - $T(B'') = \log_{10}(fin)$
- Il vient donc
  - $T(fin) = 1 + fin \times log_{10}(fin)$
  - $= fin \times log_{10}(fin) + 1$
- Par quoi borner $T(fin)$?
  - $O(fin \times log_{10}\ (fin))$
  - **complexité linéarithmique**

Nombres Parfaits

```
#include <stdio.h>
int main(){
  unsigned int nMax, n, som, div;

  printf("Entrez une valeur pour nMax: ");
  scanf("%u", &nMax);

  for(n=1; n<nMax; n++){
    som = 0;

    for(div=1; div<n; div++){
      if(!(n % div))
        som += div;
    }//fin for - div
    if(som==n)
      printf(“%u\n”, n);
  }//fin for - n
}//fin programme
```

Nombres Parfaits (2)

```
#include <stdio.h>
int main(){
  unsigned int nMax, n=1, som, div;
  printf("Entrez une valeur pour nMax: ");
  scanf("%u", &nMax);

  while(n<nMax){
    som = 0;
    div = 1;
    while(div<n){
      if(!(n % div))
        som += div;
      div++;
    }//fin for - div
    if(som==n)
      printf("%u\n", n);
    n++;
  }//fin for - n
}//fin programme
```

T(A)

T(B')

T(B'')

T(B)

T(B''')

Nombres Parfaits (3)

- Par application des règles 2 & 6
  - $T(nMax) = T(A) + T(B)$
- Par application des règles 1 & 5

$$\begin{aligned} T(nMax) &= T(A) + T(B) \\ &= 1 + \sum_{n=1}^{nMax-1} \left(T(B') + T(B'') + T(B''')\right) \\ &= 1 + \sum_{n=1}^{nMax-1} \left(1 + T(B'') + 1\right) \end{aligned}$$

- Quid de $T(B'')$?

Nombres Parfaits (4)

- Evaluation *T(B'')*
  - application de la règle 5

```
while(div<n){
  if(!(n % div))
    som += div;
  div++;
}//fin for - div
```

$T_1(B'')$

$T_2(B'')$

- Application de la règle 3
  - $T_1(B'') = 1$
- Application de règle 1
  - $T_2(B'') = 1$

Nombres Parfaits (5)

- Il vient donc

$$T(B'') = \sum_{div=1}^{n-1} \left(T_1(B'') + T_2(B'')\right)$$
$$= \sum_{div=1}^{n-1} (1+1)$$
$$= \sum_{div=1}^{n-1} 2$$
$$= 2 \times n - 2$$

- Dans le pire des cas, *T(B'')* est exécuté *nMax*-1 fois
- On a donc
  - $T(B'') = 2 \times (nMax - 1)$

Nombres Parfaits (6)

- On a donc, pour *T(B)*

$$
\begin{aligned}
T(B) &= \sum_{n=1}^{nMax-1} \left(T(B') + T(B'') + T(B''')\right) \\
&= \sum_{n=1}^{nMax-1} \left(2 + T(B'')\right) \\
&= \sum_{n=1}^{nMax-1} \left(2 + 2 \times (nMax - 1)\right) \\
&= nMax - 1 \times \left(2 \times (nMax - 1) + 2\right) \\
&= 2 \times nMax^2 - 2 \times nMax
\end{aligned}
$$

Nombres Parfaits (7)

- Il vient donc
  - $T(nMax) = T(A) + T(B)$
    $= 1 + 2 \times nMax^2 - 2 \times nMax$
    $= 2 \times nMax^2 - 2 \times nMax + 1$
- Par quoi borner *T(nMax)*?
  - $O(nMax^2)$
  - **complexité quadratique**