# Développement efficace (R3.02)
# Quelques mots de complexité

Marin Bougeret
LIRMM, IUT/Université de Montpellier


# Outline

# Idée 1 : toute opération élémentaire coûte 1

Idée 1 : compter le nombre d'opérations (pas le temps), et considérer que toutes les opération élém. ont le MEME coût.

Ainsi, les calculs sont indépendants de la machine !

On considère que toutes les opérations élémentaires coûtent 1 :

- opération arithm/logiques (+,-,..,et,ou..)
- déclaration, affectations
- lecture d'une case d'un tableau (t[i])
- comparaison de deux types de base
- retour d'une valeur (return)

Attention : cela ne s'applique pas aux opérations suivantes :

- appel d'une fonction (x = f(n)) : compter les op. de f(n)
- entrée/sorties

# Idée 1 : toute opération élémentaire coûte 1

Idée 1 : compter le nombre d'opérations (pas le temps), et considérer que toutes les opération élém. ont le MEME coût.

Ainsi, les calculs sont indépendants de la machine !

On considère que toutes les opérations élémentaires coûtent 1 :

- opération arithm/logiques (+,-,..,et,ou..)
- déclaration, affectations
- lecture d'une case d'un tableau (t[i])
- comparaison de deux types de base
- retour d'une valeur (return)

Attention : cela ne s'applique pas aux opérations suivantes :

- appel d'une fonction (x = f(n)) : compter les op. de f(n)
- entrée/sorties

# Idée 1 : toute opération élémentaire coûte 1

Idée 1 : compter le nombre d'opérations (pas le temps), et considérer que toutes les opération élém. ont le MEME coût.

Ainsi, les calculs sont indépendants de la machine !

On considère que toutes les opérations élémentaires coûtent 1 :

- opération arithm/logiques (+,-,..,et,ou..)
- déclaration, affectations
- lecture d'une case d'un tableau (t[i])
- comparaison de deux types de base
- retour d'une valeur (return)

Attention : cela ne s'applique pas aux opérations suivantes :

- appel d'une fonction (x = f(n)) : compter les op. de f(n)
- entrée/sorties

# Idée 1 : toute opération élémentaire coûte 1

```
void f(){
   int x;//1
   x = (3+1);//2
   if(x > 1){//1
      int y = x;//2
   }
}
```

**Notation**

Le nombre d'opérations de l'algorithme sera noté $m$

Nombre d'opérations de $f$ : $m = 6$

# Idée 1 : toute opération élémentaire coûte 1

```
void f(int n){
  int x = 10; //1
  for(int i=1;i<=n;i++){
    x = x+1; //2
    x = x*2; //2
  }
}
```

Pour compter plus facilement, ré-ecrivons un code équivalent

# Idée 1 : toute opération élémentaire coûte 1

```
void f(int n){
  int x = 10; //1
  int i=1;//1
  while(i<=n){//1 par test
    x = x+1; //2
    x = x*2; //2
    i = i+1;//2
  }
}
```

m(n) =
2+
7+ //i=1
7+ //i=2
..
7+ //i=n
+1 //test sortie
= 7n+3

# Idée 1 : toute opération élémentaire coûte 1

```
void f(int n){
   int x = 10; //1
   int i=1;//1
   while(i<=n){//1 par test
      x = x+1; //2
      x = x*2; //2
      i = i+1;//2
   }
}
```

m(n) =
2+
7+ //i=1
7+ //i=2
..
7+ //i=n
+1 //test sortie
= 7n+3

# Idée 2 : compter dans le pire des cas

**Idée 2 : compter le nombre d'opérations dans le pire des cas**

```
boolean recherche(int x, int[] t){
  boolean trouve = false;
  int i = 0;
  while((!trouve) && (i<t.length)){
    trouve = (t[i]==x);
    i++;
  }
  return trouve;
}
```

- au lieu de prouver "pour tout x, pour tout tableau de taille n, nb op. de recherche(x,t) $=$ .."
- on va prouver "pour tout x, pour tout tableau de taille n, nb op. de recherche(x,t) $\leq$ .."

# Idée 2 : compter dans le pire des cas

```
boolean recherche(int x, int[] t){
  boolean trouve = false;
  int i = 0;
  while((!trouve) && (i<t.length)){
    trouve = (t[i]==x);
    i++;
  }
  return trouve;
}
```

$m(n) \leq 6 + 9n$

# Idée 3 : ignorer les constantes

## Idée 3 : ignorer les constantes

- obtenir "pour tout ....., $m(n) \leq 2n$"

  ou

- obtenir "pour tout ....., $m(n) \leq 4n$"

est "équivalent" vu notre modèle

- pourquoi la valeur des constantes n'a pas vraiment de sens pour nous ?
- car on compte toutes les opérations élémentaires comme valant 1, mais vrai coût des opérations élémentaires n'est sans doute pas le même ("+" coûte 1, mais "x" coûte 2 (?) ..)

# Idée 3 : ignorer les constantes

## Idée 3 : ignorer les constantes

- obtenir "pour tout ....., $m(n) \leq 2n$"

  ou

- obtenir "pour tout ....., $m(n) \leq 4n$"

est "équivalent" vu notre modèle

- pourquoi la valeur des constantes n'a pas vraiment de sens pour nous ?
- car on compte toutes les opérations élémentaires comme valant 1, mais vrai coût des opérations élémentaires n'est sans doute pas le même ("+" coûte 1, mais "x" coûte 2 (?) ..)

# Idée 3 : ignorer les constantes

## Idée 3 : ignorer les constantes

- obtenir "pour tout ....., $m(n) \leq 2n$"

  ou

- obtenir "pour tout ....., $m(n) \leq 4n$"

est "équivalent" vu notre modèle

- pourquoi la valeur des constantes n'a pas vraiment de sens pour nous ?
- car on compte toutes les opérations élémentaires comme valant 1, mais vrai coût des opérations élémentaires n'est sans doute pas le même ("+" coûte 1, mais "x" coûte 2 (?) ..)

# Idée 3 : ignorer les constantes

- c'est une bonne nouvelle !
- en effet, reprenons l'exemple précédent :

  (!trouve && (i < t.length)) //$m$ <= 2? 3? 4? 5?
- a présent, on dira $\exists c_0$ (par exemple 1000 ici) telle que

  (!trouve && (i < t.length)) //$m$ <= $c_0$

Recommençons l'analyse de l'exemple précédent.

# Idée 3 : ignorer les constantes

- c'est une bonne nouvelle !
- en effet, reprenons l'exemple précédent :

  (!trouve && (i < t.length)) //$m$ <= 2? 3? 4? 5?
- a présent, on dira $\exists c_0$ (par exemple 1000 ici) telle que

  (!trouve && (i < t.length)) //$m$ <= $c_0$

Recommençons l'analyse de l'exemple précédent.

# Idée 3 : ignorer les constantes

```
1 boolean recherche(int x, int[] t){
2   boolean trouve = false;
3   int i = 0;
4   while((!trouve) && (i<t.length)){
5     trouve = (t[i]==x);
6     i++;
7   }
8   return trouve;
9 }
```

A présent on dit:

- $\exists c_1$ tq les opérations des l2 et l3 coûtent $\leq c_1$
- $\exists c_2$ tq le test du while l4, l5 et l6 coûtent $\leq c_2$
- $\exists c_3$ tq les opérations des l7 coûtent $\leq c_3$

Remarquez que les $c_i$ ne dépendent pas des paramètres de l'algo (et donc de $n$), c'est pour cela qu'on les appelle des constantes.

# Idée 3 : ignorer les constantes

```
1 boolean recherche(int x, int[] t){
2   boolean trouve = false;
3   int i = 0;
4   while((!trouve) && (i<t.length)){
5     trouve = (t[i]==x);
6     i++;
7   }
8   return trouve;
9 }
```

On obtient donc

$\exists c_1, c_2, c_3$ tq $\forall (x, t)$ avec $t$ ayant $n$ cases,

$$\begin{aligned} m(n) &\leq c_1 + nbtour \times c_2 + c_3 \\ &\leq c_1 + c_2 n + c_3 \end{aligned}$$

# Idée 4 : conserver seulement le terme dominant

**Idée 4 : conserver seulement le terme dominant**

Si l'on obtient par exemple "$m(n) \leq c_1 + c_2 n + c_3 n^2 + c_4 n^3$" :

$$\begin{aligned} & c_1 + c_2 n + c_3 n^2 + c_4 n^3 \\ \leq\ & c_1 n^3 + c_2 n^3 + c_3 n^3 + c_4 n^3 \\ =\ & c n^3 \text{ avec } c = c_1 + c_2 + c_3 + c_4 \end{aligned}$$

(valable seulement pour $n \geq 1$, ce que l'on supposera partout)

Retour à l'exemple précédent :

# Idée 4 : conserver seulement le terme dominant

**Idée 4 : conserver seulement le terme dominant**

Si l'on obtient par exemple "$m(n) \leq c_1 + c_2 n + c_3 n^2 + c_4 n^3$" :

$$\begin{aligned} & c_1 + c_2 n + c_3 n^2 + c_4 n^3 \\ \leq\ & c_1 n^3 + c_2 n^3 + c_3 n^3 + c_4 n^3 \\ =\ & cn^3 \text{ avec } c = c_1 + c_2 + c_3 + c_4 \end{aligned}$$

(valable seulement pour $n \geq 1$, ce que l'on supposera partout)

Retour à l'exemple précédent :

# Idée 4 : conserver seulement le terme dominant

**Idée 4 : conserver seulement le terme dominant**

Si l'on obtient par exemple "$m(n) \leq c_1 + c_2 n + c_3 n^2 + c_4 n^3$" :

$$\begin{aligned} & c_1 + c_2 n + c_3 n^2 + c_4 n^3 \\ \leq\ & c_1 n^3 + c_2 n^3 + c_3 n^3 + c_4 n^3 \\ =\ & c n^3 \text{ avec } c = c_1 + c_2 + c_3 + c_4 \end{aligned}$$

(valable seulement pour $n \geq 1$, ce que l'on supposera partout)

Retour à l'exemple précédent :

# Idée 4 : conserver seulement le terme dominant

```
1 boolean recherche(int x, int[] t){
2   boolean trouve = false;
3   int i = 0;
4   while((!trouve) && (i<t.length)){
5     trouve = (t[i]==x);
6     i++;
7   }
8   return trouve;
9 }
```

$\exists c_1, c_2, c_3$ tq $\forall (x, t)$ avec $t$ ayant $n$ cases,

$$\begin{aligned} m(n) &\leq c_1 + nbtour \times c_2 + c_3 \\ &\leq c_1 + c_2 n + c_3 \\ &\leq c_1 n + c_2 n + c_3 n \\ &= cn \end{aligned}$$

(avec $c = c_1 + c_2 + c_3$)

# Idée 4 : conserver seulement le terme dominant

```
1 boolean recherche(int x, int[] t){
2   boolean trouve = false;
3   int i = 0;
4   while((!trouve) && (i<t.length)){
5     trouve = (t[i]==x);
6     i++;
7   }
8   return trouve;
9 }
```

$\exists c$ tq $\forall (x,t)$ avec $t$ ayant $n$ cases,

$$m(n) \leq cn$$

# Conclusion du modèle

## Résumé

Dans la complexité pire cas, l'objectif est donc de prouver des résultats du type :

- $\exists$ constante $c$ / pour toute entrée de "taille" $n$, $m(n) \leq cf(n)$

## Classification usuelle

On retrouvera souvent les catégories suivantes :

- $\exists c$ tq pour tout $n \geq 1$, $m(n) \leq clog(n)$ (algo logarithmique)
- $\exists c$ tq pour tout $n \geq 1$, $m(n) \leq cn$ (algo linéaire)
- $\exists c$ tq pour tout $n \geq 1$, $m(n) \leq cnlog(n)$
- $\exists c$ tq pour tout $n \geq 1$, $m(n) \leq cn^2$ (algo quadratique)
- $\exists c$ tq pour tout $n \geq 1$, $m(n) \leq cn^3$ (algo cubique)
- $\exists c$ tq pour tout $n \geq 1$, $m(n) \leq c2^n$ (algo exponentiel)

Si $m(n) \leq cn^a$ ($a$ constante) on parle d'algorithme polynomial.

# Conclusion du modèle

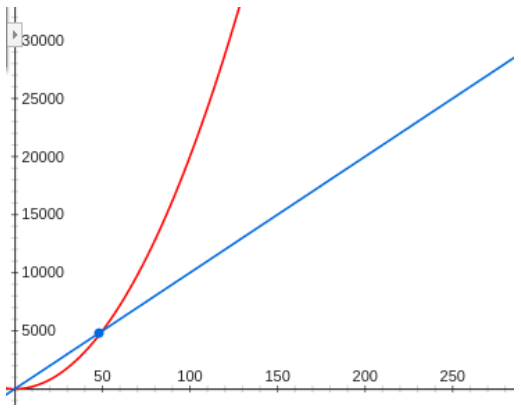
## But

Pour les complexités du type $m(n) \leq cn^a$ ($a$ constante), le but est de minimiser $a$ (peu importe $c$).

## Pourquoi ce but ?

Soient $A_1$ et $A_2$ tels que :

- $A_1$ coûte $m_1(n) \leq 100n$
- $A_2$ coûte $m_2(n) \leq 2n^2$




- quand $n$ est grand ($\geq 50$ ici), c'est la fonction avec le plus petit "$a$" qui est la plus petite ($m_1$ donc) !
- et de toute façon, quand $n$ petit, les deux algorithmes sont très rapides

# Conclusion du modèle

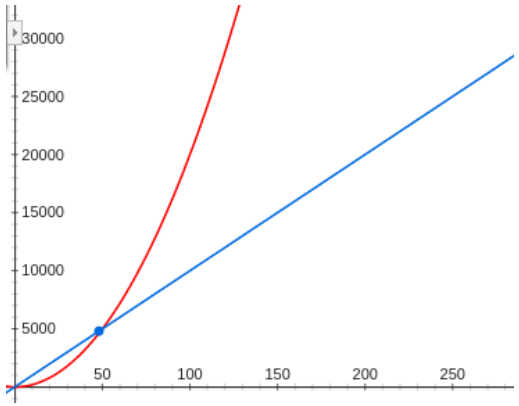
## But

Pour les complexités du type $m(n) \leq cn^a$ ($a$ constante), le but est de minimiser $a$ (peu importe $c$).

## Pourquoi ce but ?

Soient $A_1$ et $A_2$ tels que :

- $A_1$ coûte $m_1(n) \leq c_1 n^{a_1}$
- $A_2$ coûte $m_2(n) \leq c_2 n^{a_2}$




- quand $n$ est grand, c'est la fonction avec le plus petit $a$ qui est la plus petite
- et de toute façon, quand $n$ petit, les deux algorithmes sont très rapides

C'est donc un autre argument pour dire que la valeur des constantes $c_i$ ne nous intéresse pas.

# Classification usuelle

- Hypothèses:
  - ordinateur 100 Mips
  - traitement de 1 élément = 100 instructions machine

| | 10 | 50 | 100 | 500 | 1000 | 10000 |
|---|---|---|---|---|---|---|
| n | .00001 sec. | .00005 sec. | .0001 sec. | .0005 sec. | .001 sec. | .01 sec. |
| $n^2$ | .0001 sec. | .0025 sec. | .01 sec. | .25 sec. | 1 sec. | 1.6 min. |
| $n^3$ | .001 sec. | .125 sec. | 1 sec. | 2.08 min. | 16.6 min. | 11.57 jours |
| $n^5$ | .1 sec. | 5.2 min. | 2.7 heures | 1 année | 31.7 années | $31x10^3$ siècles |
| $2^n$ | .001 sec. | 35.7 années | $4x10^{14}$ siècles | ... | ... | ... |
| $3^n$ | .059 sec. | $2x10^8$ siècles | ... | ... | ... | ... |

source : Introduction à la complexité algorithmique : Y. Deville, UCL 1999

Morale : une avancée technologique ne suffira pas à compenser un algorithme de grande complexité

# Remarque

Calculer la complexité peut être très difficile ! Exemple :

```
1 boolean collatz(int n){//prérequis : n >= 1
2   while(n != 1){
3     if(n pair)
4       n = n/2;
5     else
6       n=3n+1;
7   }
8   return true;
9 }
```

Si l'on savait calculer la complexité (et même juste prouver que l'algorithme termine), on aurait résolu un grand problème des mathématiques!

On se limitera dans ce cours à des algos. dont la complexité est facile à déterminer.

# Outline

# Définition

## Définition

Soient $m$ et $f$ deux fonctions de $\mathbb{N}^*$ dans $\mathbb{R}^+$. On dit que $m = \mathcal{O}(f)$ (se lit "$m$ est en grand O de $f$") ssi

- $\exists$ constante $c$ / pour tout $n \geq 1$, $m(n) \leq cf(n)$

Notation adaptée ici, car cache la valeur de la constante.

## Classification usuelle

Reformulation des catégories précédentes :

- $m = \mathcal{O}(log(n))$ (algo logarithmique)
- $m = \mathcal{O}(n)$ (algo linéaire)
- $m = \mathcal{O}(nlog(n))$
- $m = \mathcal{O}(n^2)$ (algo quadratique)
- $m = \mathcal{O}(n^3)$ (algo cubique)
- $m \leq \mathcal{O}(2^n)$ (algo exponentiel)

# Quelques propriétés du $\mathcal{O}$

## Notation

On utilisera $\mathcal{O}(f)$ directement dans des expressions :
Ex : $m = \mathcal{O}(f_1) + \mathcal{O}(f_2)$ signifie $m = m_1 + m_2$ avec $m_i = \mathcal{O}(f_i)$

## Lemme

- $m = \mathcal{O}(f_1) + O(f_2) \Rightarrow m = \mathcal{O}(f_1 + f_2)$
- $m = \mathcal{O}(f_1) \times O(f_2) \Rightarrow m = \mathcal{O}(f_1 f_2)$
- $m = \mathcal{O}(cf_1) \Rightarrow m = \mathcal{O}(f_1)$ (avec $c$ une constante)
- $m = \mathcal{O}(f_1) + O(f_2) \Rightarrow m = \mathcal{O}(max(f_1, f_2))$

Preuve de $m = \mathcal{O}(f_1) + O(f_2) \Rightarrow m = \mathcal{O}(f_1 + f_2)$

- $m = m_1 + m_2$ et $\exists c_i$ tq $\forall n \geq 1,\ m_i(n) \leq c_i f_i(n)$
- donc $\forall n \geq 1,\ m(n) \leq c_1 f_1(n) + c_2 f_2(n) \leq c(f_1(n) + f_2(n))$
  avec $c = max(c_1, c_2)$

# Quelques propriétés du $\mathcal{O}$

## Notation

On utilisera $\mathcal{O}(f)$ directement dans des expressions :
Ex : $m = \mathcal{O}(f_1) + \mathcal{O}(f_2)$ signifie $m = m_1 + m_2$ avec $m_i = \mathcal{O}(f_i)$

## Lemme

- $m = \mathcal{O}(f_1) + O(f_2) \Rightarrow m = \mathcal{O}(f_1 + f_2)$
- $m = \mathcal{O}(f_1) \times O(f_2) \Rightarrow m = \mathcal{O}(f_1 f_2)$
- $m = \mathcal{O}(cf_1) \Rightarrow m = \mathcal{O}(f_1)$ (avec $c$ une constante)
- $m = \mathcal{O}(f_1) + O(f_2) \Rightarrow m = \mathcal{O}(max(f_1, f_2))$

Preuve de $m = \mathcal{O}(cf_1) \Rightarrow m = \mathcal{O}(f_1)$ (avec $c$ une constante)

- $m = cm_1$ et $\exists c_1$ tq $\forall n \geq 1$, $m_1(n) \leq c_1 f_1(n)$
- donc $\forall n \geq 1$, $m(n) \leq c \times c_1 f_1(n) \leq c_0 f_1(n)$ avec $c_0 = c \times c_1$

# Quelques propriétés du $\mathcal{O}$

## Notation

On utilisera $\mathcal{O}(f)$ directement dans des expressions :
Ex : $m = \mathcal{O}(f_1) + \mathcal{O}(f_2)$ signifie $m = m_1 + m_2$ avec $m_i = \mathcal{O}(f_i)$

## Lemme

- $m = \mathcal{O}(f_1) + O(f_2) \Rightarrow m = \mathcal{O}(f_1 + f_2)$
- $m = \mathcal{O}(f_1) \times O(f_2) \Rightarrow m = \mathcal{O}(f_1 f_2)$
- $m = \mathcal{O}(cf_1) \Rightarrow m = \mathcal{O}(f_1)$ (avec $c$ une constante)
- $m = \mathcal{O}(f_1) + O(f_2) \Rightarrow m = \mathcal{O}(max(f_1, f_2))$

Preuve de $m = \mathcal{O}(f_1) + O(f_2) \Rightarrow m = \mathcal{O}(max(f_1, f_2))$

- $m = m_1 + m_2$ et $\exists c_i$ tq $\forall n \geq 1$, $m_i(n) \leq c_i f_i(n)$
- donc $\forall n \geq 1$,

$$m(n) \leq c_1 f_1(n) + c_2 f_2(n) \leq c(f_1(n) + f_2(n)) \leq 2cf(n) \leq c' f(n)$$

avec $c = max(c_1, c_2)$, $f(n) = max(f_1(n), f_2(n))$ et $c' = 2c$

# Conséquence pour l'analyse du code

```
1 boolean recherche(int x, int[] t){
2   boolean trouve = false;
3   int i = 0;
4   while((!trouve) && (i<t.length)){
5     trouve = (t[i]==x);
6     i++;
7   }
8   return trouve;
9 }
```

Une deuxième méthode pour montrer que recherche est en $\mathcal{O}(n)$:

- soit $m_{init}$ coût des lignes 2 et 3
- soit $m_{boucle}$ coût des tests dans while et lignes 5, 6
- soit $m_{fin}$ coût ligne 8

# Conséquence pour l'analyse du code

```
1 boolean recherche(int x, int[] t){
2   boolean trouve = false;
3   int i = 0;
4   while((!trouve) && (i<t.length)){
5     trouve = (t[i]==x);
6     i++;
7   }
8   return trouve;
9 }
```

Une deuxième méthode pour montrer que recherche est en $\mathcal{O}(n)$:

$$\begin{aligned} m &\leq m_{init} + n \times m_{boucle} + m_{fin} \\ &= \mathcal{O}(1) + n \times \mathcal{O}(1) + \mathcal{O}(1) \\ &\Rightarrow \mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(1) \\ &\Rightarrow \mathcal{O}(n) + \mathcal{O}(1) \\ &\Rightarrow \mathcal{O}(n) \end{aligned}$$

# Conséquence pour l'analyse du code

```
boolean algo(...){
   .. une partie 0 init en m0=O(1)

   .. une partie 1 en m1=O(n^2)

   .. une partie 2 en m2=O(n)
}
```

$$
\begin{aligned}
m &= m_0 + m_1 + m_2 \\
&= \mathcal{O}(1) + \mathcal{O}(n^2) + \mathcal{O}(n) \\
&\Rightarrow \mathcal{O}(n^2) + \mathcal{O}(n) \\
&\Rightarrow \mathcal{O}(n^2)
\end{aligned}
$$

**Conclusion**

Quand le code se décompose en un nombre constant de parties, la partie la plus chère détermine à elle seule la complexité.

# Complexité en fonction de plusieurs variables

## Définition (par ex pour 2 variables)

Soient $m$ et $f$ deux fonctions de $\mathbb{N}^* \times \mathbb{N}^*$ dans $\mathbb{R}^+$. On dit que $m = \mathcal{O}(f)$ (se lit "$m$ est en grand O de $f$") ssi

- $\exists$ constante $c$ / pour tout $n_1 \geq 1$, $n_2 \geq 1$, $m(n_1, n_2) \leq cf(n_1, n_2)$

# Complexité en fonction de plusieurs variables

```
int[] tri(int []t) //t[i] >= 1
  int []res = new int[t.length]; //<= c_0 n
  int e = 0; //indice ecriture //c_1
  int x = 1; //c_2
  while(e < t.length){ //r: nb tour de while
    for(int i=0;i<t.length;i++){
      //c_3 pour tests dans for et
          instructions dans la boucle
      if(t[i]==x)
          res[e]=x; e++;
    }
    x++; //c_4
  }
  return res; //c_5
```

Soit $V$ la valeur max des $t[i]$. Montrons que $m = \mathcal{O}(nV)$.

$$m(V,n) \quad \leq \quad c_0 n + c_1 + c_2 + r(nc_3 + c_4) + c_5$$

# Complexité en fonction de plusieurs variables

$$\begin{aligned} m(V,n) &\leq c_0 n + c_1 + c_2 + r(nc_3 + c_4) + c_5 \\ &\leq rnc_0 + rnc_1 + rnc_2 + rn_c 3 + rnc_4 + rnc_5 \\ &\leq crn \quad (\text{avec} \quad c = c_1 + \cdots + c_5) \\ &\leq cVn \end{aligned}$$

On a donc obtenu que $m = \mathcal{O}(nV)$.

# Complexité en fonction de plusieurs variables

Parfois, introduire plusieurs paramètres permet d'exprimer plus finement la complexité :

- ex pour un problème où l'on prend un graphe $G$ en entrée (à $n_S$ sommets et $n_E$ arêtes)
- on peut avoir une complexité $m(G) \leq c(n_S + n_E)$,
- si l'on veut uniquement exprimer en $n_S$, on peut majorer $c(n_S + n_E) \leq c(n_s + n_s^2) \leq 2cn_S^2 = c' n_s^2$
- mais la deuxième expression est moins précise

# Outline

# Résultats négatifs

Jusqu'à présent, notre but était de montrer des énoncés du type $m = \mathcal{O}(n^2)$, c'est à dire :

- $\exists$ constante $c$ / pour tout tab $t$ de $n$ cases, $m(n) \leq cn^2$

Autrement dit, on montre que l'algorithme est "bon" (on garantit que pour toute entrée, $m(n) \leq$ ..)

## Problème

Imaginons que l'on ait prouvé que $m = \mathcal{O}(n^2)$. Comment prouver un résultat négatif, c'est à dire que :

- notre analyse est la meilleure possible ?
- ou autrement dit, que l'on a PAS $m = \mathcal{O}(n)$ ?
- c'est à dire que $\nexists$ constante $c'$ telle que pour tout tableau $t$ de $n$ cases, $m \leq c'n$ ?

# Résutlats négatifs : méthode de l'adversaire

1. pour tout $n$, définir un tableau $t_n$ de $n$ cases bien choisi qui fait faire "beaucoup" d'opérations à l'algorithme
2. prouver (en annotant le code) que $m \geq \ldots$ (par ex $m \geq \frac{n(n-1)}{2}$) (ici ne pas utiliser de $c_i$ : on peut juste affirmer "la l4 coûte au moins 1", et non "la l4 coûte au moins $c_i$ avec $c_i$ un grand entier")

Cela suffit à prouver

- $\nexists$ constante c' / pour tout tab $t$ de $n$ cases, $m \leq c'n$

En effet, supposons par contradiction que

- $\exists$ constante c' / pour tout tab $t$ de $n$ cases, $m \leq c'n$

Alors en particulier on a : pour tout $n$, avec le tableau $t_n$ :

$$\frac{n(n-1)}{2} \leq m(n) \leq c'n \quad \Rightarrow \quad \frac{(n-1)}{2} \leq c'$$

une contradiction, en passant à la limite quand $n \to +\infty$

# Exemple

```
1  public static void triBulle(int []t)
2    boolean pb= true;
3    while(pb){
4      pb = false;
5      for(int i=0;i<t.length-1;i++){
6        if(t[i]>t[i+1])
7          pb=true;
8          int tmp = t[i];
9          t[i]=t[i+1];
10         t[i+1]=tmp;
11     }
12   }
```

Prouvons par exemple que $\nexists$ constante c' / pour tout tab $t$ de $n$ cases, $m \leq c'n$

- pour tout $n$, soit $t_n = \{n-1, n-2, ..., 0\}$
- montrons que triBulle($t_n$) fait $m(n) \geq n(n-1)$ opération

# Exemple

```
1  public static void triBulle(int []t)
2    boolean pb= true;
3    while(pb){
4      pb = false;
5      for(int i=0;i<t.length-1;i++){
6        if(t[i]>t[i+1])
7          pb=true;
8          int tmp = t[i];
9          t[i]=t[i+1];
10         t[i+1]=tmp;
11     }
12   }
```

$$\begin{aligned} m &\geq nbtour \times \text{coût(lignes 4 .. 11)} \\ &\geq nbtour \times (n-1) \\ &= n \times (n-1) \end{aligned}$$

- coût(lignes 4 .. 11) $\geq$ (n-1) car on ignore l4 et on compte coût(lignes 6 .. 10) $\geq 1$
- nbtour = n : justification au tableau

# Exemple

On en déduit avec le même raisonnement que précédemment que $\nexists$ constante c' / pour tout tab $t$ de $n$ cases, $m \leq c'n$ :

En effet, supposons par contradiction que

- $\exists$ constante c' / pour tout tab $t$ de $n$ cases, $m \leq c'n$

Alors en particulier on a : pour tout $n$, avec le tableau $t_n$ :

$$n(n-1) \leq m(n) \leq c'n \Rightarrow (n-1) \leq c'$$

une contradiction, en passant à la limite quand $n \to +\infty$

# Résultats négatifs

## Bilan

Pour prouver qu'on ne peut pas avoir mieux que $m = \mathcal{O}(n^a)$

- trouver pour tout $n \geq 1$ un tableau $t_n$ où le nombre d'op. $m(n)$ vérifie $m(n) \geq P(n)$, où $P$ est un polynôme de degré $a$

- cela implique que pour tout $\epsilon > 0$, on a pas $m = \mathcal{O}(n^{a-\epsilon})$
- ex, si $m(n) \geq P(n)$ avec $P(n) = \frac{(n-3)^2}{5} - n + 25$, cela suffit pour $a = 2$, même si $P(n)$ est un peu plus petit que $n^2$

- l'étape 1 correspond à jouer le rôle de l'adversaire
- être un bon adversaire (qui trouve de "méchantes" entrées pour un algorithme donné) est une compétence utile pour un algorithmicien, au delà des calculs de complexité

# Outline

# Complexité des algorithmes récursifs

- pour calculer la complexité d'un algorithme itératif, on annote le code (avec des constantes $c_i$).
- pour calculer la complexité d'un algorithme récursif .. plus difficile, on a pas envie de faire la trace à la main en annotant !

# Complexité des algorithmes récursifs

Par ex, comment analyser par ex les algorithmes suivants ?

```
void triFusion(int[] t, int i, int j){
  if(i<j)
    int m=(i+j)/2;
    triFusion(t,i,m);
    triFusion(t,m+1,j);
    fusion(t,i,m+1,j);//O(n)
}

boolean rech(int x){
    //recherche dans classe arbre
  if(estVide())
   return false;
  else
  return val==x||filsG.rech(x)||filsD.rech(x);
}
```

# Technique standard

En général

1. on exécute à la main pour trouver une fonction candidate $f(n)$ telle que $m(n) \leq cf(n)$
2. on essayer de prouver l'inégalité obtenue par récurrence

Ici, étape 2 pas au programme (peut être compliqué!), on se contentera du 1

- pour triFusion, on a $\mathcal{O}(nlog(n))$ (arbre de hauteur $n$, et sur chaque niveau $\mathcal{O}(n)$ calculs en tout)
- pour rech, on a $\mathcal{O}(n)$

# Outline

# Critiques classiques du modèle

## Critique 1 : parcours partiel = parcours total

- en complexité pire cas, un parcours partiel et total ont la même complexité (par ex $\mathcal{O}(n)$ pour une recherche dans un tableau)
- c'est dommage ! N'oubliez pas de toujours faire des parcours partiels !

# Critiques classiques du modèle

## Critique 2 : meilleur algo en pratique != en théorie

- parfois, il existe un algorithme $A_1$ en $\mathcal{O}(f_1)$ et un autre algorithme $A_2$ en $\mathcal{O}(f_2)$, avec $f_1 >= f_2$, et pourtant $A_1$ est meilleur "en pratique" !

## Raison 1 : rareté du pire cas

Exemple : quickSort en $\mathcal{O}(n^2)$ est en pratique plus rapide triFusion, pourtant en $\mathcal{O}(nlog(n))$

- pourquoi ? Car très peu d'instances provoquent le pire cas !
- il existe d'autres modèles de complexité qui évitent cet inconvénient

# Critiques classiques du modèle

## Critique 2 : meilleur algo en pratique != en théorie

- parfois, il existe un algorithme $A_1$ en $\mathcal{O}(f_1)$ et un autre algorithme $A_2$ en $\mathcal{O}(f_2)$, avec $f_1 >= f_2$, et pourtant $A_1$ est meilleur "en pratique" !

## Raison 2 : constantes monstrueuses

Exemple : $100000000n^2$ vs $2n^3$ :

- l'argument de "quand $n$ devient grand (c'est à dire $n \geq v$), le $n^2$ est meilleur" est toujours vrai..
- mais si la valeur $v$ à partir de laquelle c'est vrai est astronomique, on préfèrera le $2n^3$

# Conclusion

## Démarche suggérée

- d'abord optimiser la puissance du $n$ (petit $a$ dans $n^a$)
- une fois votre meilleur $a$ atteint, optimiser les constantes
- ne pas oublier d'écrire des parcours partiels quand cela est possible
- testez plusieurs algorithmes sur vos vraies données pour comprendre si les pires cas se produisent vraiment