Algorithmique avancée : introduction à la récursivité & complexité

Marin Bougeret

IUT Montpellier

Outline

Introduction

- but : parler du temps d'exécution d'un algorithme
- cette question a été ignorée jusqu'à maintenant car ..
- nous testions les algorithmes sur des petites entrées

Que sont des "grandes entrées" ?
Ce sont des données de "la vraie vie" :

- chercher un mot dans un document
- traitement d'images
- graphe d'inter-connections de villes
- séquences d'adn
- recherche sur le web
- ..

Introduction

- but : parler du temps d'exécution d'un algorithme
- cette question a été ignorée jusqu'à maintenant car ..
- nous testions les algorithmes sur des petites entrées

Que sont des "grandes entrées" ?
Ce sont des données de "la vraie vie" :

- chercher un mot dans un document
- traitement d'images
- graphe d'inter-connections de villes
- séquences d'adn
- recherche sur le web
- ..

Introduction

- but : parler du temps d'exécution d'un algorithme
- cette question a été ignorée jusqu'à maintenant car ..
- nous testions les algorithmes sur des petites entrées

Que sont des "grandes entrées" ?
Ce sont des données de "la vraie vie" :

- chercher un mot dans un document
- traitement d'images
- graphe d'inter-connections de villes
- séquences d'adn
- recherche sur le web
- ..

Introduction

Sur des "grandes entrées" :

- on peut observer des ralentissements
- pire : certains algorithmes deviennent inutilisables !

Exemples en direct :

- meilleure sous somme en n^2 et en n
- stable maximum en $n^2 2^n$

- parler du temps d'exec est donc capital
- but : pouvoir écrire puis prouver qu'un algorithme est rapide
- pas uniquement un souci de théoricien :
  - être capable d'estimer au premier coup d'oeil si un algorithme est très rapide ou inutilisable est important
  - permet de comprendre pourquoi il y a de la recherche en informatique ! :)

Introduction

Sur des "grandes entrées" :

- on peut observer des ralentissements
- pire : certains algorithmes deviennent inutilisables !

Exemples en direct :

- meilleure sous somme en n^2 et en n
- stable maximum en $n^2 2^n$

- parler du temps d'exec est donc capital
- but : pouvoir écrire puis prouver qu'un algorithme est rapide
- pas uniquement un souci de théoricien :
  - être capable d'estimer au premier coup d'oeil si un algorithme est très rapide ou inutilisable est important
  - permet de comprendre pourquoi il y a de la recherche en informatique ! :)

Introduction

Sur des "grandes entrées" :

- on peut observer des ralentissements
- pire : certains algorithmes deviennent inutilisables !

Exemples en direct :

- meilleure sous somme en n^2 et en n
- stable maximum en $n^2 2^n$

- parler du temps d'exec est donc capital
- but : pouvoir écrire puis prouver qu'un algorithme est rapide
- pas uniquement un souci de théoricien :
  - être capable d'estimer au premier coup d'oeil si un algorithme est très rapide ou inutilisable est important
  - permet de comprendre pourquoi il y a de la recherche en informatique ! :)

Introduction

Sur des "grandes entrées" :

- on peut observer des ralentissements
- pire : certains algorithmes deviennent inutilisables !

Exemples en direct :

- meilleure sous somme en n^2 et en n
- stable maximum en $n^2 2^n$

- parler du temps d'exec est donc capital
- but : pouvoir écrire puis prouver qu'un algorithme est rapide
- pas uniquement un souci de théoricien :
  - être capable d'estimer au premier coup d'oeil si un algorithme est très rapide ou inutilisable est important
  - permet de comprendre pourquoi il y a de la recherche en informatique ! :)

Introduction

Sur des "grandes entrées" :

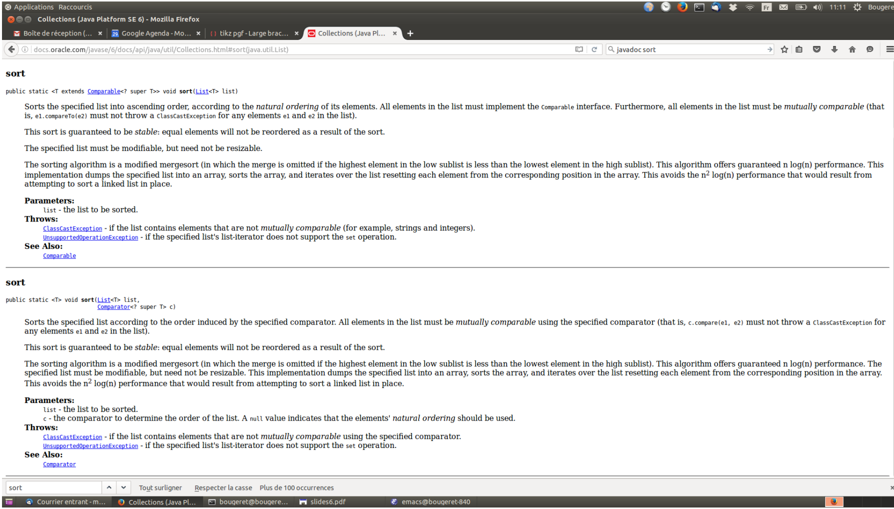
- on peut observer des ralentissements
- pire : certains algorithmes deviennent inutilisables !

Exemples en direct :

- meilleure sous somme en n^2 et en n
- stable maximum en $n^2 2^n$

- parler du temps d'exec est donc capital
- but : pouvoir écrire puis prouver qu'un algorithme est rapide
- pas uniquement un souci de théoricien :
  - être capable d'estimer au premier coup d'oeil si un algorithme est très rapide ou inutilisable est important
  - permet de comprendre pourquoi il y a de la recherche en informatique ! :)

Extrait de la javadoc



sort

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the *natural ordering* of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

The specified list must be modifiable, but need not be resizable.

The sorting algorithm is a modified mergesort (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist). This algorithm offers guaranteed n log(n) performance. This implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array. This avoids the n^2 log(n) performance that would result from attempting to sort a linked list in place.

Parameters:
`list` - the list to be sorted.
Throws:
`ClassCastException` - if the list contains elements that are not *mutually comparable* (for example, strings and integers).
`UnsupportedOperationException` - if the specified list's list-iterator does not support the `set` operation.
See Also:
`Comparable`

sort

```
public static <T> void sort(List<T> list,
                            Comparator<? super T> c)
```

Sorts the specified list according to the order induced by the specified comparator. All elements in the list must be *mutually comparable* using the specified comparator (that is, `c.compare(e1, e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

The sorting algorithm is a modified mergesort (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist). This algorithm offers guaranteed n log(n) performance. The specified list must be modifiable, but need not be resizable. This implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array. This avoids the n^2 log(n) performance that would result from attempting to sort a linked list in place.

Parameters:
`list` - the list to be sorted.
`c` - the comparator to determine the order of the list. A `null` value indicates that the elements' *natural ordering* should be used.
Throws:
`ClassCastException` - if the list contains elements that are not *mutually comparable* using the specified comparator.
`UnsupportedOperationException` - if the specified list's list-iterator does not support the `set` operation.
See Also:
`Comparator`

Outline

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 !

Ainsi, toutes les opérations élémentaires suivantes 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)

Irréaliste de compter 1 partout ? Certes, mais nous verrons pourquoi cela n'est pas gênant

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

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

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 !

Ainsi, toutes les opérations élémentaires suivantes 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)

Irréaliste de compter 1 partout ? Certes, mais nous verrons pourquoi cela n'est pas gênant

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

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

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 !

Ainsi, toutes les opérations élémentaires suivantes 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)

Irréaliste de compter 1 partout ? Certes, mais nous verrons pourquoi cela n'est pas gênant

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

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

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 !

Ainsi, toutes les opérations élémentaires suivantes 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)

Irréaliste de compter 1 partout ? Certes, mais nous verrons pourquoi cela n'est pas gênant

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

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

Exemple 1 : sans boucle

```
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 : $m = 6$

Exemple 2 : avec boucle

```
void f(){
  ..
  for(int i=1;i<=100;i++){
    x = x+1; //2
    x = x*2; //2
  }
}
```

- il y a donc $c_{boucle} = 4$ opérations par tour de boucle
- attention, il faut aussi compter les opérations de la parenthèse du for
- pour ce faire, ré-ecrivons un code équivalent

Exemple 2 : avec boucle

```
void f(){
  ..
  int i=1;
  while(i<=100){//1 par test
    .. //c_{boucle};
    i = i+1;//2
  }
}
```

m =
2+//init i
1+c_{boucle}+2 //i=1
1+c_{boucle}+2 //i=2
..
1+c_{boucle}+2
+1 (test i=101)
$= 2 + nbtour(3 + c_{boucle}) + 1$ (= 2+100(3+4)+1)

Exemple 2 : avec boucle

```
void f(){
  ..
  int i=1;
  while(i<=100){//1 par test
    .. //c_{boucle};
    i = i+1;//2
  }
}
```

$m =$
2+//init i
1+c_{boucle}+2 //i=1
1+c_{boucle}+2 //i=2
..
1+c_{boucle}+2
+1 (test i=101)
$= 2 + nbtour(3 + c_{boucle}) + 1$ (= 2+100(3+4)+1)

Exemple 2 : avec boucle

```
void f(){
  ..
  for(int i=1;i<=100;i++){
    x = x+1; //2
    x = x*2; //2
  }
}
```

Remarque

- cela était bien fastidieux !
- nous verrons plus tard que l'approximation $m = c_{boucle} * 100 = 400$ est satisfaisante

Idée 2 : compter le nombre d'opérations en fonction des entrées

```
int somme(int n){
  int res = 0;//2
  for(int i=1;i<=n;i++){
    res = res+i;//2
  }
  return res;//1
}
```

$m =$

$2+$

$= 6 + 5n$

Idée 2 : compter le nombre d'opérations en fonction des entrées

```
int somme(int n){
  int res = 0;//2
  for(int i=1;i<=n;i++){
    res = res+i;//2
  }
  return res;//1
}
```

$m =$

$$2+(3+\text{nbtour}(3+c_{boucle}))+$$

$$= 6 + 5n$$

Idée 2 : compter le nombre d'opérations en fonction des entrées

```
int somme(int n){
  int res = 0;//2
  for(int i=1;i<=n;i++){
    res = res+i;//2
  }
  return res;//1
}
```

$$m = 2+(3+\text{nbtour}(3+c_{boucle}))+1$$
$$= 6+5n$$

Idée 2 : compter le nombre d'opérations en fonction des entrées

```
int somme(int n){
  int res = 0;//2
  for(int i=1;i<=n;i++){
    res = res+i;//2
  }
  return res;//1
}
```

$m =$

$$2+(3+\text{nbtour}(3+c_{boucle}))+1$$

$$= 6 + 5n$$

Idée 2 : compter le nombre d'opérations en fonction des entrées

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

Rmq : on ne compte que en fonction de n=t.length (pas de x).

$$m = 9 + (9 \times nbtour)$$

Mais que vaut *nbtour* ? Dépend des valeurs dans le tableau !

Idée 2 : compter le nombre d'opérations en fonction des entrées

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

Rmq : on ne compte que en fonction de n=t.length (pas de x).

$$m \quad = \quad 9 + (9 \times \mathit{nbtour})$$

Mais que vaut *nbtour* ? Dépend des valeurs dans le tableau !

Idée 2 : compter le nombre d'opérations en fonction des entrées

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

Rmq : on ne compte que en fonction de n=t.length (pas de x).

$$m = 9 + (9 \times nbtour)$$

Mais que vaut *nbtour* ? Dépend des valeurs dans le tableau !

Idée 3 : compter le nombre d'opérations dans le pire des cas

- 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 3 : compter le nombre d'opérations dans le pire des cas

- 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 3 : 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;
}
```

$$m \leq 9 + 9n$$

Idée 4 : ignorer les constantes

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

  ou

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

est "équivalent" vu notre modèle

- pourquoi la valeur des constantes n'est pas intéressante ?

$\Rightarrow$ car de toute façon le vrai coût des opérations élémentaires n'est pas le même ("+" coûte 1, mais "x" coûte 2, ..)

- mais prendre en compte ce vrai coût est beaucoup trop compliqué, et dépendrai de la machine (cf idée 1)

Idée 4 : ignorer les constantes

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

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

est "équivalent" vu notre modèle

- pourquoi la valeur des constantes n'est pas intéressante ?

$\Rightarrow$ car de toute façon le vrai coût des opérations élémentaires n'est pas le même ("+" coûte 1, mais "x" coûte 2, ..)

- mais prendre en compte ce vrai coût est beaucoup trop compliqué, et dépendrai de la machine (cf idée 1)

Idée 4 : ignorer les constantes

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

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

est "équivalent" vu notre modèle

- pourquoi la valeur des constantes n'est pas intéressante ?

$\Rightarrow$ car de toute façon le vrai coût des opérations élémentaires n'est pas le même ("+" coûte 1, mais "x" coûte 2, ..)

- mais prendre en compte ce vrai coût est beaucoup trop compliqué, et dépendrai de la machine (cf idée 1)

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

  (!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.

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

  (!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 4 : 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 4 : ignorer les constantes

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

On obtient donc

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

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

Idée 5 : conserver seulement le terme dominant

Par exemple, si l'on obtient "$m \leq 6 + 2n + 3n^2 + n^3$" :

$$\begin{aligned} & 6 + 2n + 3n^2 + n^3 \\ \leq\ & 6n^3 + 2n^3 + 3n^3 + n^3 \\ =\ & 12n^3 \end{aligned}$$

Plus généralement, si l'on obtient "$m \leq c_1 + c_2 n + c_3 n^2 + c_4 n^3$" :

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

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

Idée 5 : conserver seulement le terme dominant

Par exemple, si l'on obtient "$m \leq 6 + 2n + 3n^2 + n^3$" :

$$\begin{aligned} & 6 + 2n + 3n^2 + n^3 \\ \leq\ & 6n^3 + 2n^3 + 3n^3 + n^3 \\ =\ & 12n^3 \end{aligned}$$

Plus généralement, si l'on obtient "$m \leq c_1 + c_2 n + c_3 n^2 + c_4 n^3$" :

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

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

Idée 5 : conserver seulement le terme dominant

Par exemple, si l'on obtient "$m \leq 6 + 2n + 3n^2 + n^3$" :

$$\begin{aligned} & 6 + 2n + 3n^2 + n^3 \\ \leq\ & 6n^3 + 2n^3 + 3n^3 + n^3 \\ =\ & 12n^3 \end{aligned}$$

Plus généralement, si l'on obtient "$m \leq c_1 + c_2 n + c_3 n^2 + c_4 n^3$" :

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

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

Idée 5 : conserver seulement le terme dominant

Par exemple, si l'on obtient "$m \leq 6 + 2n + 3n^2 + n^3$" :

$$\begin{aligned} & 6 + 2n + 3n^2 + n^3 \\ \leq\ & 6n^3 + 2n^3 + 3n^3 + n^3 \\ =\ & 12n^3 \end{aligned}$$

Plus généralement, si l'on obtient "$m \leq c_1 + c_2 n + c_3 n^2 + c_4 n^3$" :

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

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

Idée 5 : conserver seulement le terme dominant

Par exemple, si l'on obtient "$m \leq 6 + 2n + 3n^2 + n^3$" :

$$
\begin{aligned}
& 6 + 2n + 3n^2 + n^3 \\
\leq\ & 6n^3 + 2n^3 + 3n^3 + n^3 \\
=\ & 12n^3
\end{aligned}
$$

Plus généralement, si l'on obtient "$m \leq c_1 + c_2 n + c_3 n^2 + c_4 n^3$" :

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

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

Idée 5 : conserver seulement le terme dominant

Conclusion

On ne s'intéresse qu'au terme le plus grand asymptotiquement.

- $m \leq 12n^3$, plutôt que
- $m \leq 6 + 2n + 3n^2 + n^3$

Est-ce grave de majorer si violemment ?

- non, car ce sont les grandes valeurs de n qui nous intéressent ! (pour des petites valeurs, tous les algorithmes sont immédiats!)
- pour $n \geq 200$, $6 + 2n + 3n^2$ représente moins de $0,5\%$ de n^3 !!

But

On va donc se contenter de prouver des résultats du type :

- $\exists$ constante c / pour tout .. de taille n, $m \leq cf(n)$

Par ex : $m \leq cn^2$, $m \leq cn^3$, $m \leq c2^n$.

Application sur l'exemple précédent

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

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

$$\begin{aligned} m &\leq c_1 \\ &\quad + nbtour \times c_2 \\ &\quad + c_3 \\ &\leq c_1 + c_2 n + c_3 \\ &\leq c_1 n + c_2 n + c_3 n \\ &= cn \text{ avec } c = c_1 + c_2 + c_3 \end{aligned}$$

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

Ce qui se ré-écrit :

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

$$m \leq cn$$

Attention à l'ordre des quantificateurs !

Ce que nous avons prouvé :

1) $\exists c \in \mathbb{N}$ tq $\forall (x, t)$ avec t ayant n cases, $m \leq cn$

est très différent de :

2) $\forall (x, t)$ avec t ayant n cases, $\exists c \in \mathbb{N}$ tq $m \leq cn$

En effet, pensez à la différence entre ces deux énnoncés

- $\exists x \in \mathbb{N}$ tq $\forall y \in \mathbb{N}$, $y \leq x$ (faux, il n'y a pas de valeur x "fixée" plus grande que tous les autres entiers y)

vs

- $\forall y \in \mathbb{N}$, $\exists x \in \mathbb{N}$ tq $y \leq x$ (vrai, pour chaque y, on peut trouver un x (dont la valeur dépend de y) tq $y \leq x$)

Attention à l'ordre des quantificateurs !

Ce que nous avons prouvé :

1) $\exists c \in \mathbb{N}$ tq $\forall (x, t)$ avec t ayant n cases, $m \leq cn$

est très différent de :

2) $\forall (x, t)$ avec t ayant n cases, $\exists c \in \mathbb{N}$ tq $m \leq cn$

Conclusion

- l'énnoncé 2 est très "faible" et n'est pas intéressant : pour chaque (x, t) avec t ayant n cases, on pourra bien trouver un $c \in \mathbb{N}$ (mais dont la valeur peut dépendre de n) tq $m \leq cn$.
- en effet, le fait que c puisse dépendre de n nous permet de choisir un c "très grand" (par ex $c = 2^{2^n}$).
- on pourrait d'ailleurs même prouver $\forall (x, t)$ avec t ayant n cases, $\exists c \in \mathbb{N}$ tq $m \leq c$
- l'important dans 1) est que la valeur de c ne dépend pas des paramètres de l'algo, c'est pour cela qu'on l'appelle constante

Classification usuelle

Ainsi, on retrouvera souvent les catégories suivantes :

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

algo poly

- En vrai, on utilise la notation $\mathcal{O}$ et on dit par exemple qu'un algo est linéaire quand $m = \mathcal{O}(n)$
- Ici : on s'économise l'introduction et le temps d'apprentissage de $\mathcal{O}$, mais l'esprit est exactement le même!

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	31×10^3 siècles
2^n	.001 sec.	35.7 années	4×10^{14} siècles	...	...	...
3^n	.059 sec.	2×10^8 siècles	...	...	...	...

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

Classification usuelle

Une autre façon de le dire : une avancée technologique ne suffira pas à compenser un algorithme de grande complexité

Influence de l'évolution technologie

- Hypothèses:
  - ordinateur 100 Mips
  - traitement de 1 élément = 100 instructions machine
- N_i = taille du "plus grand"' exemple dont la solution peut être calculée en *1 heure* de temps calcul

Complexité	ordinateur d'aujourd'hui	ordinateur 100 x	ordinateur 1000x
n	$N_1 = 3.6 \times 10^9$	$100\ N_1$	$1000\ N_1$
n^2	$N_2 = 600000$	$10\ N_2$	$31.6\ N_2$
n^3	$N_3 = 1530$	$4.64\ N_3$	$10\ N_3$
n^5	$N_4 = 81$	$2.5\ N_4$	$3.98\ N_4$
2^n	$N_5 = 31$	$N_5 + 6$	$N_5 + 10$
3^n	$N_6 = 20$	$N_6 + 4$	$N_6 + 6$

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

Outline

Ex complet 2

But : $\exists$ constante c / pour tout tableau t de $n \geq 1$ cases, $m \leq cn^2$

```
1 int compterCouplesEgaux(int []t)
2   int res = 0;
3   for(int i=0;i<t.length;i++){
4     for(int j=0;j<t.length;j++){
5       if ((i!=j)&&t[i]==t[j]){res = res+1;}
6     }
7   }
8   return res/2;
```

$\exists c_1, c_2, c_3$ tq pour tout t de n cases :

$$
\begin{aligned}
m &\leq c_1 && \text{l2}\\
&\quad +nbtour \times c_2 && \text{l3-7}\\
&\quad +c_3 && \text{l8}\\
&\leq c_1 + c_2 n^2 + c_3 &&\\
&\leq c_1 n^2 + c_2 n^2 + c_3 n^2 &&
\end{aligned}
$$

avec $c = c_1 + c_2 + c_3$

Retour sur meilleure sous somme : v1

But : $\exists$ constante c / pour tout tableau t de $n \geq 1$ cases, $m \leq cn^2$

```
int v1(int [] t)
  int best = t[0];
  for(int i = 0;i < t.length;i++){
    //calcul de best s.s. demarrant par t[i]
    int somme=0;
    for(int j=i;j<t.length;j++){
       somme = somme+t[j];
       if(somme > best){best=somme;}
    }
  }
  return best;
```

$\exists c_1, c_2, c_3$ tq pour tout t de n cases :

- pour chaque i fixé on effectue $m_i \leq c_1 + nbtour_i c_2 \leq c_1 + nc_2$
- et $m \leq c_3 + nbtour(c_1 + nc_2)$

Et donc :

- $m \leq c_3 + c_1 n + c_2 n^2 \leq c_3 n^2 + c_1 n^2 + c_2 n^2 = cn^2$ avec $c = ..$

Retour sur meilleure sous somme : v2

But : $\exists$ constante c / pour tout tableau t de $n \geq 1$ cases, $m \leq cn$

```
int v2(int [] t)
  int aux = t[0];
  int best = aux;
  for(int i=1;i<t.length;i++){
    //aux contient meilleur SS qui termine en i
       -1
    //best contient meilleur SS de t[0..(i-1)]
    aux = Math.max(aux+t[i],t[i]);
    if(aux > best){best = aux;}
  }
  return best;
```

$\exists c_1, c_2, c_3$ tq pour tout t de n cases :

- $m \leq c_1 + n(c_2)$

Et donc :

- $m \leq c_1 n + c_2 n = nc$ avec $c = c_1 + c_2$

Retour sur stable maximum

But : $\exists$ constante c / pour tout tab g de $n \times n$ cases, $m \leq cn^2 2^n$

```
public static int maxIS(boolean [][]g)
  int[] cour = new int[g.length];
  for(int i=0;i<cour.length;i++){
    cour[i]=0;
  }
  int best=0;
  int indiceZero = hasNext(cour); //<= c0 n
  while(indiceZero != -1){
    next(cour,indiceZero); //donne l'ensemble
       suivant (ex 010111 -> 011000)
    if(estStable(g,cour)){ //<= c1 n^2 : faire
       au tableau
      int nbs = nbSommets(cour); //<= c2 n
      if(nbs > best){best=nbs;}
    }
    indiceZero = hasNext(cour); //<= c3 n
  }
  return best;
```

Outline

Résultats négatifs

Jusqu'à présent, notre but était de montrer des énoncés du type :

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

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

Problème

Imaginons que l'on ait prouvé :

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

On voudrait à présent être sûr que l'on n'aurait pas pu fournir une meilleure analyse. On voudrait donc prouver (par exemple) :

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

Comment prouver un tel énoncé ?

Méthode

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 $m \geq ...$ (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 :

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

une contradiction, prendre par ex $n = 200c' + 10$

Exemple

```java
public static void triBulle(int []t)
  boolean pb= true;
  while(pb){
    pb = false;
    for(int i=0;i<t.length-1;i++){
      if(t[i]>t[i+1])
        pb=true;
        int tmp = t[i];
        t[i]=t[i+1];
        t[i+1]=tmp;
    }
  }
```

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 $m \geq n(n-1)$ 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   }
```

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

- 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 :

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

une contradiction, prendre par ex $n = c' + 2$

Résultats négatifs

Bilan informel

1. pour prouver qu'on ne peut pas avoir mieux que $m \leq cn^2$ ..
2. il suffit de trouver pour tout $n \geq 1$ un tableau t_n où le nombre d'opérations m vérifie $m \geq P(n)$, où P est un polynôme du 2nd degré
3. Par ex, si $m \geq P(n)$ avec $P(n) = \frac{(n-3)^2}{5} - n$, cela suffit, même si $P(n)$ est un peu plus petit que n^2

Plus généralement, la même technique s'applique aussi pour prouver qu'on ne peut pas avoir mieux que $m \leq cn^a$, pour tout a.

Résultats négatifs

Choses exigibles

1. ce que l'on vous demandera : "trouver pour tout n un tableau t_n tq $m \geq \frac{(n-3)^2}{5} - n$" (par exemple)
2. ce que l'on ne vous demandera pas : "en déduire que $\nexists c' tq...$"

(c'est en fait 1 qui est le plus difficile (et le plus intéressant), 2 s'obtient juste avec le petit calcul vu précédemment)

Outline

- but : apprendre à reconnaître une complexité en un coup d'oeil
- cela ne dispense pas de faire la preuve proprement ensuite

```
for(int i=1;i<=n;i++){
   .. //nb op constant
}
```

$m \leq cn$

```
for(int i=1;i<=n;i++){
   for(int j=1;j<=n;j++){
      .. //nb op constant
   }
}
```

$m \leq cn^2$

```
for(int i=1;i<=n;i++){
   for(int j=i;j<=n;j++){
      .. //nb op constant
   }
}
```

$m \leq cn^2$

```
for(int i=1;i<=n;i++){
   for(int j=0;j<=n;j++){
      .. //nb op constant
   }
}
```

$m \leq cn^2$

Remarque sur l'ex en bas à droite

Il y a bien $n(n+1) > n^2$ tours de boucle, mais la "magie" des contantes nous permet de dire que $\exists c$ tq pour tout n, $m \leq cn^2$.

```
for(int i=1;i<=n;i++){
    //nb op constant
    ..
    ..
}
for(int i=1;i<=n;i++){
    //nb op constant
    ..
}
for(int i=1;i<=n;i++){
    //nb op constant
    ..
    ..
}
for(int i=1;i<=n;i++){
    //nb op constant
    ..
}
```

$m \leq cn$

Conclusion : plusieurs boucles de suite $\Leftrightarrow$ une boucle

```
for(int i=1;i<=n;i++){
  for(int j=1;j<=n;j++){
   //nb op constant
  }
}
for(int i=1;i<=n;i++){
   //nb op constant
}
for(int i=1;i<=100n;i++){
   //nb op constant
}
for(int i=1;i<=143434343434n;i++){
   //nb op constant
}
for(int i=1;i<=n;i++){
   //nb op constant
}
```

$m \leq cn^2$

Conclusion : c'est la boucle imbriquée qui domine et donne la complexité

```
int i=0;
while(i<n){

  ..
  if(..){
    i--;
  }
  else{
    i++;
  }
}
```

Attention! On ne peut pas dire $m \leq cn$!