# R1.01 – Chapitre 9

# Tris et complexité

Guilherme Dias da Fonseca

# Complexité asymptotique

- Imaginons un algorithme dont l'entrée a une taille $n$
- Si le temps d'exécution $t(n)$ est proportionnel à $n$, on dit que sa complexité est $O(n)$
- On ignore les constantes :

$$\left.\begin{array}{l} t(n) = n \\ t(n) = 13n \\ t(n) = n/5 \\ t(n) = 7 + n \end{array}\right\} = O(n)$$

- Est-ce qu'il y a des algorithmes plus lents que $O(n)$ ?

```
1 int f(int v[], int n) {
2   int sum = 0;
3   for(int i=0; i<n; i++)
4     sum += v[i];
5   return sum;
6 }
```

```
1 int g(int v[], int n) {
2   if(n == 0)
3     return 0;
4   return g(v,n-1) + v[n-1];
5 }
```

Deux algorithmes $O(n)$

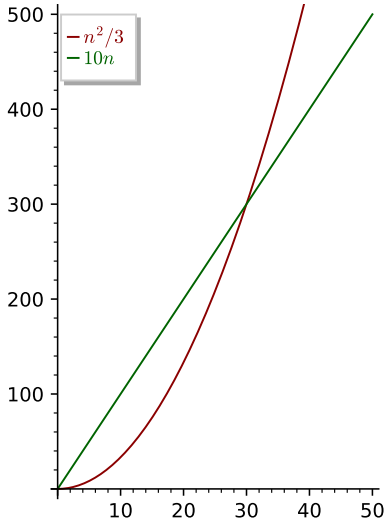
# Complexité quadratique



- Voici un algorithme $O(n^2)$ :
  (le temps est proportionnel à $n^2$)

```
1 bool h(int v[], int n) {
2   for(int i=0; i<n-1; i++)
3     for(int j=i+1; j<n; j++)
4       if(v[i] == v[j])
5         return true;
6   return false;
7 }
```

- Pour $n$ suffisamment grand, un algorithme $O(n^2)$ va être plus lent que n'importe quel algorithme linéaire ($O(n)$)



# Pire des cas



- Mais cet algorithme n'est pas toujours quadratique...
  (par exemple, si tous les éléments sont égaux)

```
1 bool h(int v[], int n) {
2   for(int i=0; i<n-1; i++)
3     for(int j=i+1; j<n; j++)
4       if(v[i] == v[j])
5         return true;
6   return false;
7 }
```

- On considère toujours le pire des cas !
- Il y a d'autres types de complexité, mais on n'en parlera pas (best, average, randomized, competitive, amortized...)

- Entrée : Tableau avec $n$ elements
- Sortie : Les éléments triés du plus petit au plus gros

| 17 | 33 | 2 | 90 | 4 | 8 | 25 | 13 | 8 | 2 |
|---|---|---|---|---|---|---|---|---|---|



| 2 | 2 | 4 | 8 | 8 | 13 | 17 | 25 | 33 | 90 |
|---|---|---|---|---|---|---|---|---|---|

- Les éléments peuvent être des `int`, `double`, `std::string`...
  (on peut comparer les `std::string` en C++ : `"apple" < "zebra"`)
- Il y a plusieurs algorithmes, dont les plus rapides ont complexité $O(n \log n)$

# Tri en C++ avec std::sort



```cpp
#include <iostream>                 // std::cout
#include <algorithm>                // std::sort

int main () {
  int n = 7;
  int v[n] = {16, 12, 7, 13, 11, 18, 4};

  std::sort(v, v + n);

  for(int i = 0; i < n; i++)
    std::cout << v[i] << ' ';       // 4 7 11 12 13 16 18
  std::cout << std::endl;

  return 0;
}
```

# Tri de std::vector avec std::sort

```cpp
#include <iostream>          // std::cout
#include <algorithm>         // std::sort
#include <vector>            // std::vector

int main () {
  std::vector<int> v = {16, 12, 7, 13, 11, 18, 4};

  std::sort(v.begin(), v.end());

  for(int x : v)
    std::cout << x << ' ';   // 4 7 11 12 13 16 18
  std::cout << std::endl;

  return 0;
}
```

# Itérateurs

- Qu'est ce que c'est `begin()` et `end()` ?
- `begin()` donne le début du `std::vector`
- `end()` donne une position après la fin
- `++it` passe `it` à la position suivante
  (éviter d'utiliser `it++` avec des itérateurs)
- `*it` donne la valeur associée à `it`
- On peut comparer avec `!=` (ne pas utiliser `<`)

Le code ci-dessous :

```
1  for(int x : v)
2    std::cout << x << ' ';
```

Est équivalent à :

```
1  for(auto it = v.begin(); it != v.end(); ++it)
2    std::cout << *it << ' ';
```

# Trier des struct

```
1  struct Point {
2    double x,y;
3  };
4
5  int main () {
6    std::vector<Point> v = {{2,3},{1,1},{2,4},{3,2}};
7
8    // Ne marche pas, on ne sait pas comparer deux Point
9    // std::sort(v.begin(), v.end());
10
11   for (Point p : v)
12     std::cout << '(' << p.x << ',' << p.y << ')';
13   std::cout << std::endl;
14
15   return 0;
16 }
```

# Trier des struct avec comparateur (pointeur de fonction)



```
1  struct Point { double x,y; };
2
3  bool smaller_x(const Point &p, const Point &q) {
4    return p.x < q.x;
5  }
6
7  int main () {
8    std::vector<Point> v = {{2,3},{1,1},{2,4},{3,2}};
9
10   std::sort(v.begin(), v.end(), smaller_x);
11
12   // (1,1)(2,4)(2,3)(3,2) ou (1,1)(2,3)(2,4)(3,2)
13   for (Point p : v)
14     std::cout << '(' << p.x << ',' << p.y << ')';
15   std::cout << std::endl;
```

# std::stable_sort



- Le comparateur précédent n'impose pas d'ordre quand $x$ est égal, même si $y$ est different
- On a alors deux résultats possibles (1,1)(2,4)(2,3)(3,2) ou (1,1)(2,3)(2,4)(3,2)
- `std::stable_sort` est stable : préserve l'ordre original dans ce cas
- Peut être un peu plus lent que `std::sort` et utilise plus de mémoire (utilise un autre algorithme complétement different)

```
1  std::vector<Point> v = {{2,3},{1,1},{2,4},{3,2}};
2  std::stable_sort(v.begin(), v.end(), smaller_x);
3  // (1,1)(2,3)(2,4)(3,2)
```

# Trier des struct avec comparateur (lambda)



On peut écrire le comparateur directement avec un lambda :

```
[](const Point &p, const Point &q){ return p.x < q.x; }
```

```
1  struct Point {
2    double x,y;
3  };
4
5  int main () {
6    std::vector<Point> v = {{2,3},{1,1},{2,4},{3,2}};
7
8    std::sort(v.begin(), v.end(),
9              [](const Point &p, const Point &q)
10             { return p.x < q.x; });
```

# Lambda avec capture

- Le lambda peut capturer des variables
- Disons qu'on veut trier les points par distance à partir d'un point `source`
- Dans les `[]` on liste les variables séparées par une virgule avec `&` avant si on veut capturer par référence

```
1  double dist(const Point &p, const Point &q) {
2    return sqrt((p.x-q.x) * (p.x-q.x) +
3                (p.y-q.y) * (p.y-q.y));
4  }
5  int main () {
6    std::vector<Point> v = {{2,3},{1,1},{2,4},{3,2}};
7    Point source = {0,0};
8
9    std::sort(v.begin(), v.end(),
10             [&source](const Point &p, const Point &q)
11             { return dist(p,source) < dist(q,source); });
```

# std::erase_if (seulement c++20)



- Il y a d'autres utilités pour les lambdas (ou les pointeurs de fonction)
- Disons qu'on a un vector d'entiers et on veut supprimer les nombres pairs

```
1 std::vector<int> v = {7,3,14,2,13,4,9};
2 std::erase_if(v, [](int x){ return x%2 == 0; });
3 // v = {7,3,13,9}
```

- Il y a aussi `for_each`, `apply`, `accumulate`, `reduce`...

# Template de fonction

Pour recevoir un lambda comme paramètre on fait un `template` de fonction :

```
1 template <class T>
2 int apply(T f, int x) {
3   return f(x);
4 }
```

Ensuite on peut l'utiliser :

```
1 auto twice = [](int x){ return 2*x; };
2 std::cout << apply(twice, 13) << std::endl;
3
4 int k = 3;
5 auto mul_k = [k](int x){ return k*x; };
6 std::cout << apply(mul_k, 13) << std::endl;
```

Notez que j'utilise `auto` pour le type du lambda

# Template de fonction

- Un `template` est un modèle de fonction pour plusieurs types différents
- Il doit être dans le `.hpp`, et pas dans le `.cpp`

À la place d'écrire :

```
1 int square(int x) { return x*x; }
2 long square(long x) { return x*x; }
3 float square(float x) { return x*x; }
4 double square(double x) { return x*x; }
5 // ...
```

On écrit tout simplement :

```
1 template <class T>
2 T square(T x) { return x*x; }
```

et le compilateur va créer la fonction si besoin

# Trier des struct à plusieurs attributs

On veut trier par $x$ et, briser l'égalité par $y$ :

```
1 bool smaller(const Point &p, const Point &q) {
2   if(p.x < q.x)
3     return true;
4   if(p.x > q.x)
5     return false;
6   return p.y < q.y; // Ici p.x == q.x
7 }
```

- Le code est long et encore plus long si on a 3 ou plus attributs...
- Il y a d'autres façons de le faire :
  - On trie par $y$ et ensuite on fait un tri stable par $x$
  - On utilise une `std::tuple`

# Tuple

```
1 bool smaller(const Point &p, const Point &q) {
2  return std::make_tuple(p.x,p.y)<std::make_tuple(q.x,q.y);
3 }
```

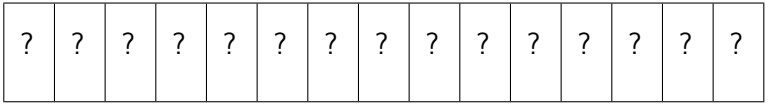
- Fonctionne avec n'importe quel nombre d'attributs
- Les attributs peuvent avoir des types différents
- Un tuple est une liste de variables
- La comparaison est faite dans ordre

```
1 std::tuple<int,char,double> t=std::make_tuple(7,'c',.5);
2 int i = std::get<0>(t);
3 char c = std::get<1>(t);
4 double d = std::get<2>(t);
```

# Applications de tri

- Trouver les doublons en $O(n)$ parce qu'ils deviennent consécutifs
- Retrouver un élément en $O(\log n)$ avec
  la recherche dichotomique (binary search) :
  A chaque pas, on élimine la moitié du tableau, alors temps $O(\log n)$

Trouver si 18 est dans un tableau trié :

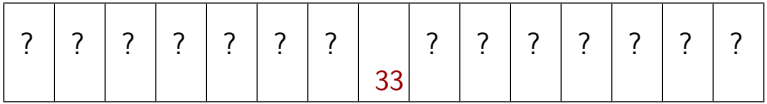


# Applications de tri



- Trouver les doublons en $O(n)$ parce qu'ils deviennent consécutifs
- Retrouver un élément en $O(\log n)$ avec
  la recherche dichotomique (binary search) :
  A chaque pas, on élimine la moitié du tableau, alors temps $O(\log n)$

Trouver si 18 est dans un tableau trié :

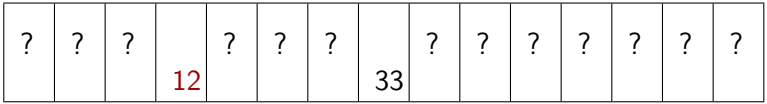


# Applications de tri



- Trouver les doublons en $O(n)$ parce qu'ils deviennent consécutifs
- Retrouver un élément en $O(\log n)$ avec
  la recherche dichotomique (binary search) :
  A chaque pas, on élimine la moitié du tableau, alors temps $O(\log n)$

Trouver si 18 est dans un tableau trié :

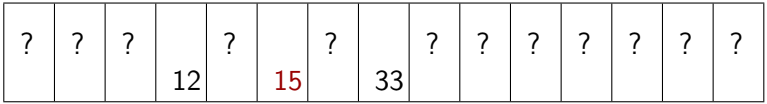


# Applications de tri



- Trouver les doublons en $O(n)$ parce qu'ils deviennent consécutifs
- Retrouver un élément en $O(\log n)$ avec
  la recherche dichotomique (binary search) :
  A chaque pas, on élimine la moitié du tableau, alors temps $O(\log n)$

Trouver si 18 est dans un tableau trié :



# Applications de tri



- Trouver les doublons en $O(n)$ parce qu'ils deviennent consécutifs
- Retrouver un élément en $O(\log n)$ avec
  la recherche dichotomique (binary search) :
  A chaque pas, on élimine la moitié du tableau, alors temps $O(\log n)$

Trouver si 18 est dans un tableau trié :

| ? | ? | ? | 12 | ? | 15 | 18 | 33 | ? | ? | ? | ? | ? | ? | ? |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

# Applications de tri

- Trouver les doublons en $O(n)$ parce qu'ils deviennent consécutifs
- Retrouver un élément en $O(\log n)$ avec
  la recherche dichotomique (binary search) :
  A chaque pas, on élimine la moitié du tableau, alors temps $O(\log n)$

Trouver si 18 est dans un tableau trié :

| 7 | 9 | 10 | 12 | 13 | 15 | 18 | 33 | 44 | 47 | 53 | 55 | 62 | 70 | 90 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

# std::binary_search

```cpp
#include <iostream>        // std::cout
#include <algorithm>       // std::binary_search
#include <vector>          // std::vector

int main() {
  std::vector<int> foin = {1, 3, 4, 5, 9};
  std::vector<int> aiguilles = {1, 2, 3};

  for (int x : aiguilles) {
    std::cout << "Chercher " << x << " : ";
    if (std::binary_search(foin.begin(), foin.end(), x))
      std::cout << "trouvé " << x << std::endl;
    else
      std::cout << x << " non trouvé" << std::endl;
  }
```

# std::lower_bound

- `std::binary_search` renvoie seulement un booléen
- `std::lower_bound` renvoie un itérateur sur la première position trouvée ou, si l'élément n'est pas trouvé, la position où il devrait être ajouté
- `std::upper_bound` renvoie un itérateur sur la dernière position trouvée ou, si l'élément n'est pas trouvé, la position où il devrait être ajouté
- Pour lister toutes les occurrences de $x$ dans un vector trié $v$ on fait :

```
1 for(auto it = std::lower_bound(v.begin(),v.end(),x);
2     it != v.end() && *it == x;
3     ++it) {
4   std::cout << *it << std::endl;
5 }
```

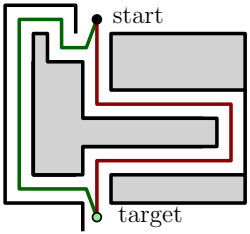
# Algorithmes de tris



- Plusieurs algorithmes basés sur des différents paradigmes
- Un paradigme est une idée générale pour développer un algorithme
- On va voir ces algorithmes sans code

## Paradigme glouton (greedy)

- On écrit la sortie de petit à petit, sans effacer
- On prend le choix le plus intéressant à court terme
- Donne des mauvaises solutions pour plusieurs problèmes




- Le tri sélection choisi le plus petit élément restant à chaque pas
- On ajoute cet élément à la fin du tableau de sortie

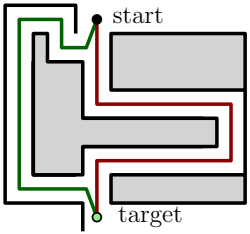
| 17 | 33 | 2 | 90 | 4 | 8 | 25 | 13 | 8 | 2 |
|---|---|---|---|---|---|---|---|---|---|

$\Downarrow$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|

## Paradigme glouton (greedy)

- On écrit la sortie de petit à petit, sans effacer
- On prend le choix le plus intéressant à court terme
- Donne des mauvaises solutions pour plusieurs problèmes




- Le tri sélection choisi le plus petit élément restant à chaque pas
- On ajoute cet élément à la fin du tableau de sortie

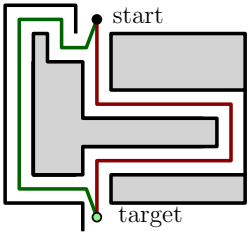
| 17 | 33 | | 90 | 4 | 8 | 25 | 13 | 8 | 2 |
|---|---|---|---|---|---|---|---|---|---|

⇓

| 2 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|

## Paradigme glouton (greedy)

- On écrit la sortie de petit à petit, sans effacer
- On prend le choix le plus intéressant à court terme
- Donne des mauvaises solutions pour plusieurs problèmes




- Le tri sélection choisi le plus petit élément restant à chaque pas
- On ajoute cet élément à la fin du tableau de sortie

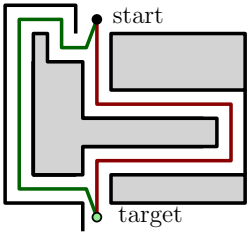
| 17 | 33 | | 90 | 4 | 8 | 25 | 13 | 8 | |
|---|---|---|---|---|---|---|---|---|---|

⇓

| 2 | 2 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|

## Paradigme glouton (greedy)

- On écrit la sortie de petit à petit, sans effacer
- On prend le choix le plus intéressant à court terme
- Donne des mauvaises solutions pour plusieurs problèmes




- Le tri sélection choisi le plus petit élément restant à chaque pas
- On ajoute cet élément à la fin du tableau de sortie

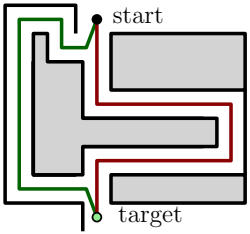
| 17 | 33 | | 90 | | 8 | 25 | 13 | 8 | |
|---|---|---|---|---|---|---|---|---|---|

⇓

| 2 | 2 | 4 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|

## Paradigme glouton (greedy)

- On écrit la sortie de petit à petit, sans effacer
- On prend le choix le plus intéressant à court terme
- Donne des mauvaises solutions pour plusieurs problèmes




- Le tri sélection choisi le plus petit élément restant à chaque pas
- On ajoute cet élément à la fin du tableau de sortie

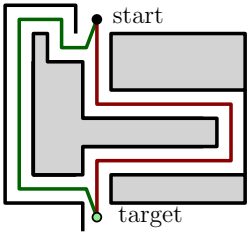
| 17 | 33 | | 90 | | | 25 | 13 | 8 | |
|---|---|---|---|---|---|---|---|---|---|

⇓

| 2 | 2 | 4 | 8 | | | | | | |
|---|---|---|---|---|---|---|---|---|---|

## Paradigme glouton (greedy)

- On écrit la sortie de petit à petit, sans effacer
- On prend le choix le plus intéressant à court terme
- Donne des mauvaises solutions pour plusieurs problèmes




- Le tri sélection choisi le plus petit élément restant à chaque pas
- On ajoute cet élément à la fin du tableau de sortie

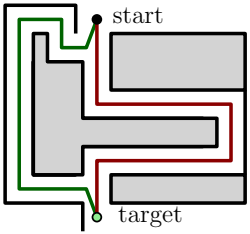
| 17 | 33 | | 90 | | | 25 | 13 | | |
|---|---|---|---|---|---|---|---|---|---|

⇓

| 2 | 2 | 4 | 8 | 8 | | | | | |
|---|---|---|---|---|---|---|---|---|---|

## Paradigme glouton (greedy)

- On écrit la sortie de petit à petit, sans effacer
- On prend le choix le plus intéressant à court terme
- Donne des mauvaises solutions pour plusieurs problèmes




- Le tri sélection choisi le plus petit élément restant à chaque pas
- On ajoute cet élément à la fin du tableau de sortie

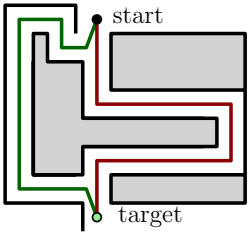
| 17 | 33 | | 90 | | | 25 | | | |
|---|---|---|---|---|---|---|---|---|---|

⇓

| 2 | 2 | 4 | 8 | 8 | 13 | | | | |
|---|---|---|---|---|---|---|---|---|---|

## Paradigme glouton (greedy)

- On écrit la sortie de petit à petit, sans effacer
- On prend le choix le plus intéressant à court terme
- Donne des mauvaises solutions pour plusieurs problèmes




- Le tri sélection choisi le plus petit élément restant à chaque pas
- On ajoute cet élément à la fin du tableau de sortie

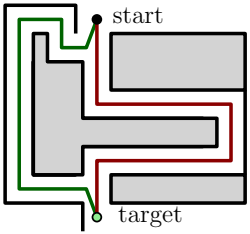
| | 33 | | 90 | | | 25 | | | |
|---|---|---|---|---|---|---|---|---|---|

⇓

| 2 | 2 | 4 | 8 | 8 | 13 | 17 | | | |
|---|---|---|---|---|---|---|---|---|---|

## Paradigme glouton (greedy)

- On écrit la sortie de petit à petit, sans effacer
- On prend le choix le plus intéressant à court terme
- Donne des mauvaises solutions pour plusieurs problèmes




- Le tri sélection choisi le plus petit élément restant à chaque pas
- On ajoute cet élément à la fin du tableau de sortie

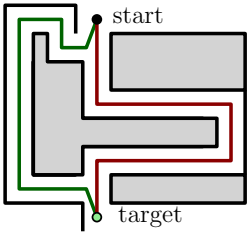
| | 33 | | 90 | | | | | | |
|---|---|---|---|---|---|---|---|---|---|

⇓

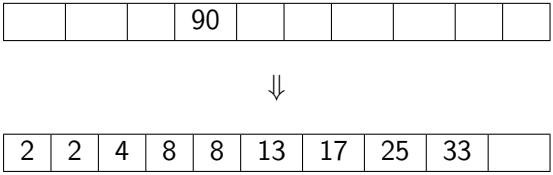
| 2 | 2 | 4 | 8 | 8 | 13 | 17 | 25 | | |
|---|---|---|---|---|---|---|---|---|---|

## Paradigme glouton (greedy)

- On écrit la sortie de petit à petit, sans effacer
- On prend le choix le plus intéressant à court terme
- Donne des mauvaises solutions pour plusieurs problèmes

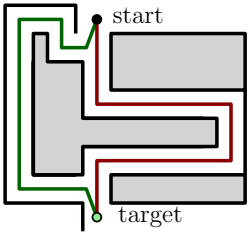



- Le tri sélection choisi le plus petit élément restant à chaque pas
- On ajoute cet élément à la fin du tableau de sortie

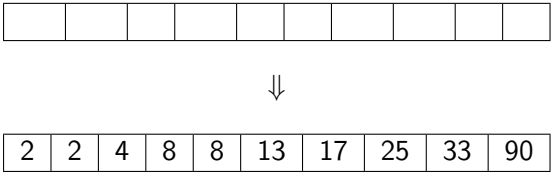


## Paradigme glouton (greedy)

- On écrit la sortie de petit à petit, sans effacer
- On prend le choix le plus intéressant à court terme
- Donne des mauvaises solutions pour plusieurs problèmes



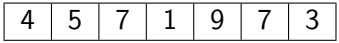

- Le tri sélection choisi le plus petit élément restant à chaque pas
- On ajoute cet élément à la fin du tableau de sortie



# Tri sélection



- On peut le faire in place :
  On écrit directement sur le tableau d'origine
- On utilise `std::swap`
- La complexité de temps est $O(n^2)$
- Contrairement à la version précédente,
  cette version n'est pas stable
- Un des tris les plus rapides pour les tout
  petits tableaux
- Fait $O(n)$ swaps, mais $O(n^2)$ comparaisons



# Tri sélection



- On peut le faire in place :
  On écrit directement sur le tableau d'origine
- On utilise `std::swap`
- La complexité de temps est $O(n^2)$
- Contrairement à la version précédente, cette version n'est pas stable
- Un des tris les plus rapides pour les tout petits tableaux
- Fait $O(n)$ swaps, mais $O(n^2)$ comparaisons

| 4 | 5 | 7 | 1 | 9 | 7 | 3 |
|---|---|---|---|---|---|---|

| 1 | 5 | 7 | 4 | 9 | 7 | 3 |
|---|---|---|---|---|---|---|

# Tri sélection



- On peut le faire in place :
  On écrit directement sur le tableau d'origine
- On utilise `std::swap`
- La complexité de temps est $O(n^2)$
- Contrairement à la version précédente, cette version n'est pas stable
- Un des tris les plus rapides pour les tout petits tableaux
- Fait $O(n)$ swaps, mais $O(n^2)$ comparaisons

| | | | | | | |
|---|---|---|---|---|---|---|
| 4 | 5 | 7 | 1 | 9 | 7 | 3 |
| 1 | 5 | 7 | 4 | 9 | 7 | 3 |
| 1 | 3 | 7 | 4 | 9 | 7 | 5 |

# Tri sélection



- On peut le faire in place :
  On écrit directement sur le tableau d'origine
- On utilise `std::swap`
- La complexité de temps est $O(n^2)$
- Contrairement à la version précédente, cette version n'est pas stable
- Un des tris les plus rapides pour les tout petits tableaux
- Fait $O(n)$ swaps, mais $O(n^2)$ comparaisons

| | | | | | | |
|---|---|---|---|---|---|---|
| 4 | 5 | 7 | 1 | 9 | 7 | 3 |
| 1 | 5 | 7 | 4 | 9 | 7 | 3 |
| 1 | 3 | 7 | 4 | 9 | 7 | 5 |
| 1 | 3 | 4 | 7 | 9 | 7 | 5 |

# Tri sélection



- On peut le faire in place :
  On écrit directement sur le tableau d'origine
- On utilise `std::swap`
- La complexité de temps est $O(n^2)$
- Contrairement à la version précédente, cette version n'est pas stable
- Un des tris les plus rapides pour les tout petits tableaux
- Fait $O(n)$ swaps, mais $O(n^2)$ comparaisons

| | | | | | | |
|---|---|---|---|---|---|---|
| 4 | 5 | 7 | 1 | 9 | 7 | 3 |
| 1 | 5 | 7 | 4 | 9 | 7 | 3 |
| 1 | 3 | 7 | 4 | 9 | 7 | 5 |
| 1 | 3 | 4 | 7 | 9 | 7 | 5 |
| 1 | 3 | 4 | 5 | 9 | 7 | 7 |

# Tri sélection



- On peut le faire in place :
  On écrit directement sur le tableau d'origine
- On utilise `std::swap`
- La complexité de temps est $O(n^2)$
- Contrairement à la version précédente, cette version n'est pas stable
- Un des tris les plus rapides pour les tout petits tableaux
- Fait $O(n)$ swaps, mais $O(n^2)$ comparaisons

# Tri sélection



- On peut le faire in place :
  On écrit directement sur le tableau d'origine
- On utilise `std::swap`
- La complexité de temps est $O(n^2)$
- Contrairement à la version précédente, cette version n'est pas stable
- Un des tris les plus rapides pour les tout petits tableaux
- Fait $O(n)$ swaps, mais $O(n^2)$ comparaisons

| | | | | | | |
|---|---|---|---|---|---|---|
| 4 | 5 | 7 | 1 | 9 | 7 | 3 |
| 1 | 5 | 7 | 4 | 9 | 7 | 3 |
| 1 | 3 | 7 | 4 | 9 | 7 | 5 |
| 1 | 3 | 4 | 7 | 9 | 7 | 5 |
| 1 | 3 | 4 | 5 | 9 | 7 | 7 |
| 1 | 3 | 4 | 5 | 7 | 9 | 7 |
| 1 | 3 | 4 | 5 | 7 | 7 | 9 |

| 4 | 2 | 7 | 1 | 9 | 7 | 3 |
|---|---|---|---|---|---|---|

## Paradigme recherche locale

On regarde un petit morceau de la solution et, si possible, l'améliore

- On parcourt le tableau et si $v[i] > v[i+1]$, on échange
- Algorithme très lent
- Complexité $O(n^2)$
- Fait $O(n^2)$ swaps et $O(n^2)$ comparaisons
- Stable
- In place

| 4 | 2 | 7 | 1 | 9 | 7 | 3 |
|---|---|---|---|---|---|---|

| 2 | 4 | 7 | 1 | 9 | 7 | 3 |
|---|---|---|---|---|---|---|

## Paradigme recherche locale

On regarde un petit morceau de la solution et, si possible, l'améliore

- On parcourt le tableau et si $v[i] > v[i+1]$, on échange
- Algorithme très lent
- Complexité $O(n^2)$
- Fait $O(n^2)$ swaps et $O(n^2)$ comparaisons
- Stable
- In place

## Paradigme recherche locale

On regarde un petit morceau de la solution et, si possible, l'améliore

- On parcourt le tableau et si $v[i] > v[i+1]$, on échange
- Algorithme très lent
- Complexité $O(n^2)$
- Fait $O(n^2)$ swaps et $O(n^2)$ comparaisons
- Stable
- In place

| 4 | 2 | 7 | 1 | 9 | 7 | 3 |
|---|---|---|---|---|---|---|

| 2 | 4 | 7 | 1 | 9 | 7 | 3 |
|---|---|---|---|---|---|---|

| 2 | 4 | 1 | 7 | 9 | 7 | 3 |
|---|---|---|---|---|---|---|

## Paradigme recherche locale

On regarde un petit morceau de la solution et, si possible, l'améliore

- On parcourt le tableau et si $v[i] > v[i+1]$, on échange
- Algorithme très lent
- Complexité $O(n^2)$
- Fait $O(n^2)$ swaps et $O(n^2)$ comparaisons
- Stable
- In place

| | | | | | | |
|---|---|---|---|---|---|---|
| 4 | 2 | 7 | 1 | 9 | 7 | 3 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 4 | 7 | 1 | 9 | 7 | 3 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 4 | 1 | 7 | 9 | 7 | 3 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 4 | 1 | 7 | 7 | 9 | 3 |

## Paradigme recherche locale

On regarde un petit morceau de la solution et, si possible, l'améliore

- On parcourt le tableau et si $v[i] > v[i+1]$, on échange
- Algorithme très lent
- Complexité $O(n^2)$
- Fait $O(n^2)$ swaps et $O(n^2)$ comparaisons
- Stable
- In place

| | | | | | | |
|---|---|---|---|---|---|---|
| 4 | 2 | 7 | 1 | 9 | 7 | 3 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 4 | 7 | 1 | 9 | 7 | 3 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 4 | 1 | 7 | 9 | 7 | 3 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 4 | 1 | 7 | 7 | 9 | 3 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 4 | 1 | 7 | 7 | 3 | 9 |

## Paradigme recherche locale

On regarde un petit morceau de la solution et, si possible, l'améliore

- On parcourt le tableau et si $v[i] > v[i+1]$, on échange
- Algorithme très lent
- Complexité $O(n^2)$
- Fait $O(n^2)$ swaps et $O(n^2)$ comparaisons
- Stable
- In place

| | | | | | | |
|---|---|---|---|---|---|---|
| 4 | 2 | 7 | 1 | 9 | 7 | 3 |
| 2 | 4 | 7 | 1 | 9 | 7 | 3 |
| 2 | 4 | 1 | 7 | 9 | 7 | 3 |
| 2 | 4 | 1 | 7 | 7 | 9 | 3 |
| 2 | 4 | 1 | 7 | 7 | 3 | 9 |
| 2 | 1 | 4 | 7 | 7 | 3 | 9 |

## Paradigme recherche locale

On regarde un petit morceau de la solution et, si possible, l'améliore

- On parcourt le tableau et si $v[i] > v[i+1]$, on échange
- Algorithme très lent
- Complexité $O(n^2)$
- Fait $O(n^2)$ swaps et $O(n^2)$ comparaisons
- Stable
- In place

| | | | | | | |
|---|---|---|---|---|---|---|
| 4 | 2 | 7 | 1 | 9 | 7 | 3 |
| 2 | 4 | 7 | 1 | 9 | 7 | 3 |
| 2 | 4 | 1 | 7 | 9 | 7 | 3 |
| 2 | 4 | 1 | 7 | 7 | 9 | 3 |
| 2 | 4 | 1 | 7 | 7 | 3 | 9 |
| 2 | 1 | 4 | 7 | 7 | 3 | 9 |
| 2 | 1 | 4 | 7 | 3 | 7 | 9 |

## Paradigme recherche locale

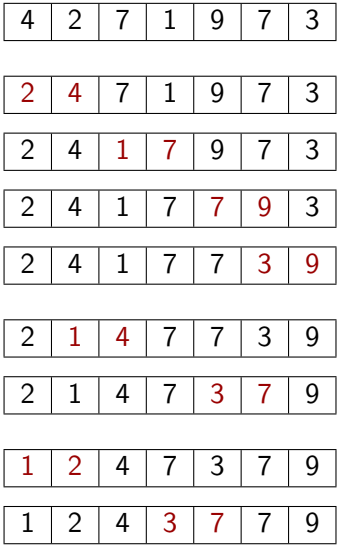
On regarde un petit morceau de la solution et, si possible, l'améliore

- On parcourt le tableau et si $v[i] > v[i+1]$, on échange
- Algorithme très lent
- Complexité $O(n^2)$
- Fait $O(n^2)$ swaps et $O(n^2)$ comparaisons
- Stable
- In place

## Paradigme recherche locale

On regarde un petit morceau de la solution et, si possible, l'améliore

- On parcourt le tableau et si $v[i] > v[i+1]$, on échange
- Algorithme très lent
- Complexité $O(n^2)$
- Fait $O(n^2)$ swaps et $O(n^2)$ comparaisons
- Stable
- In place



## Paradigme incrémental

On examine l'entrée petit à petit et on résout le morceau examiné

- Un tableau de taille 1 est déjà trié : 6

## Paradigme incrémental

On examine l'entrée petit à petit et on résout le morceau examiné

- Un tableau de taille 1 est déjà trié : | 6 |
- Pour ajouter 9 c'est facile : | 6 | 9 |

## Paradigme incrémental

On examine l'entrée petit à petit et on résout le morceau examiné

- Un tableau de taille 1 est déjà trié : | 6 |
- Pour ajouter 9 c'est facile : | 6 | 9 |
- Pour ajouter 7 il faut faire un swap : | 6 | 7 | 9 |

## Paradigme incrémental

On examine l'entrée petit à petit et on résout le morceau examiné

- Un tableau de taille 1 est déjà trié : | 6 |
- Pour ajouter 9 c'est facile : | 6 | 9 |
- Pour ajouter 7 il faut faire un swap : | 6 | 7 | 9 |
- Pour ajouter 2 il faut faire trois swaps : | 2 | 6 | 7 | 9 |

# Tri insertion



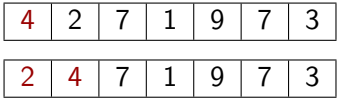
| 4 | 2 | 7 | 1 | 9 | 7 | 3 |
|---|---|---|---|---|---|---|

- Complexité $O(n^2)$
- Si le tableau est déjà trié c'est $O(n)$
- Bien pour des tableaux déjà plutôt triés
- Stable
- In place

# Tri insertion



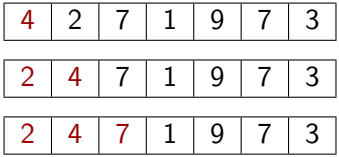
- Complexité $O(n^2)$
- Si le tableau est déjà trié c'est $O(n)$
- Bien pour des tableaux déjà plutôt triés
- Stable
- In place



# Tri insertion



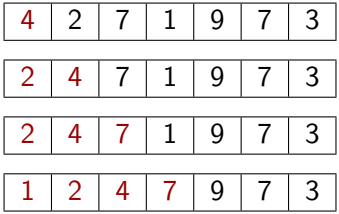
- Complexité $O(n^2)$
- Si le tableau est déjà trié c'est $O(n)$
- Bien pour des tableaux déjà plutôt triés
- Stable
- In place



# Tri insertion



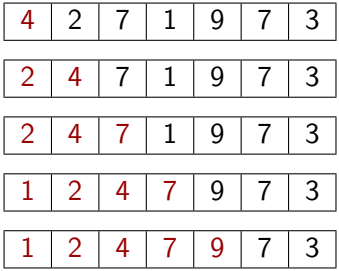
- Complexité $O(n^2)$
- Si le tableau est déjà trié c'est $O(n)$
- Bien pour des tableaux déjà plutôt triés
- Stable
- In place



# Tri insertion



- Complexité $O(n^2)$
- Si le tableau est déjà trié c'est $O(n)$
- Bien pour des tableaux déjà plutôt triés
- Stable
- In place



# Tri insertion

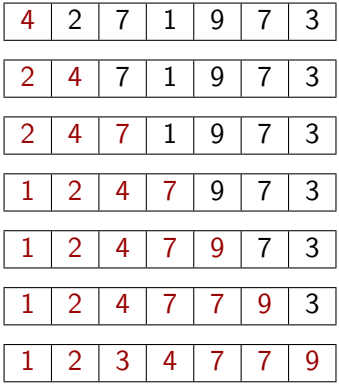


- Complexité $O(n^2)$
- Si le tableau est déjà trié c'est $O(n)$
- Bien pour des tableaux déjà plutôt triés
- Stable
- In place

# Tri insertion



- Complexité $O(n^2)$
- Si le tableau est déjà trié c'est $O(n)$
- Bien pour des tableaux déjà plutôt triés
- Stable
- In place



# Diviser pour régner (divide and conquer)



## Paradigme diviser pour régner

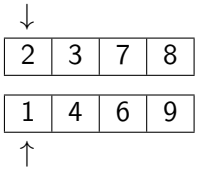
1. Diviser le problèmes en 2 (ou plus) sous-problèmes plus petits
2. Résoudre les sous-problèmes avec des appels récursifs, sauf si le problème est très petit
3. Combiner les solutions des sous-problèmes pour obtenir la solution du problème original

- Produit deux algorithmes de tri différents :
  - **mergesort** (*tri fusion*) : divise sans effort, mais combiner devient plus dur
  - **quicksort** (*tri rapide*) : effort pour diviser, mais combiner devient facile

# Merge (fusion)



- Avant de voir le tri fusion, on voit la fusion
  - Entrée : deux tableaux triés
  - Sortie : un seul tableau trié avec tous les éléments

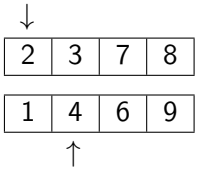



- Complexité $O(n)$ pour produire un tableau de taille $n$

# Merge (fusion)



- Avant de voir le tri fusion, on voit la fusion
  - Entrée : deux tableaux triés
  - Sortie : un seul tableau trié avec tous les éléments



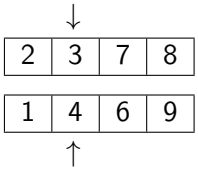


- Complexité $O(n)$ pour produire un tableau de taille $n$

# Merge (fusion)



- Avant de voir le tri fusion, on voit la fusion
  - Entrée : deux tableaux triés
  - Sortie : un seul tableau trié avec tous les éléments

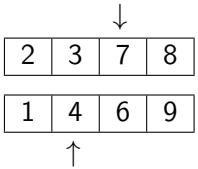




- Complexité $O(n)$ pour produire un tableau de taille $n$

# Merge (fusion)

- Avant de voir le tri fusion, on voit la fusion
  - Entrée : deux tableaux triés
  - Sortie : un seul tableau trié avec tous les éléments

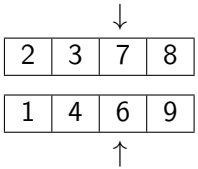


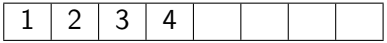


- Complexité $O(n)$ pour produire un tableau de taille $n$

# Merge (fusion)

- Avant de voir le tri fusion, on voit la fusion
  - Entrée : deux tableaux triés
  - Sortie : un seul tableau trié avec tous les éléments



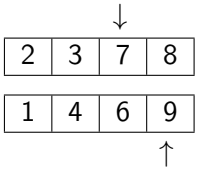



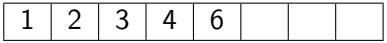

- Complexité $O(n)$ pour produire un tableau de taille $n$

# Merge (fusion)



- Avant de voir le tri fusion, on voit la fusion
  - Entrée : deux tableaux triés
  - Sortie : un seul tableau trié avec tous les éléments



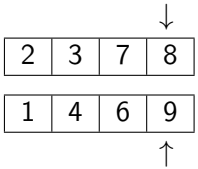



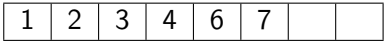

- Complexité $O(n)$ pour produire un tableau de taille $n$

# Merge (fusion)



- Avant de voir le tri fusion, on voit la fusion
  - Entrée : deux tableaux triés
  - Sortie : un seul tableau trié avec tous les éléments



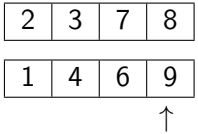



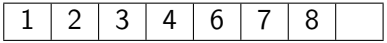

- Complexité $O(n)$ pour produire un tableau de taille $n$

# Merge (fusion)



- Avant de voir le tri fusion, on voit la fusion
  - Entrée : deux tableaux triés
  - Sortie : un seul tableau trié avec tous les éléments



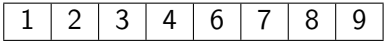




- Complexité $O(n)$ pour produire un tableau de taille $n$

# Merge (fusion)



- Avant de voir le tri fusion, on voit la fusion
  - Entrée : deux tableaux triés
  - Sortie : un seul tableau trié avec tous les éléments

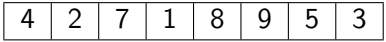




- Complexité $O(n)$ pour produire un tableau de taille $n$

# Mergesort (tri fusion)



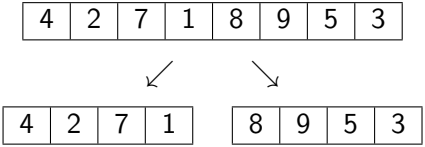
1. Si taille 1, c'est déjà trié, on renvoie directement



# Mergesort (tri fusion)



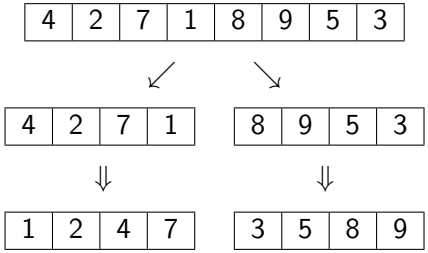
1. Si taille 1, c'est déjà trié, on renvoie directement
2. Divise le tableau en 2 moitiés



# Mergesort (tri fusion)



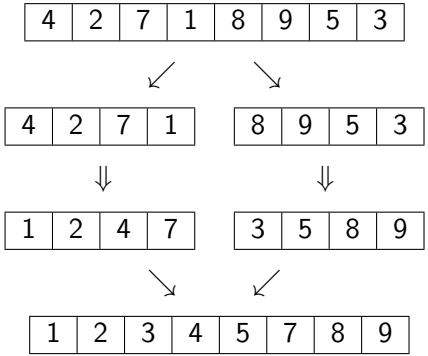
1. Si taille 1, c'est déjà trié, on renvoie directement
2. Divise le tableau en 2 moitiés
3. Résout chaque moitié récursivement



# Mergesort (tri fusion)



1. Si taille 1, c'est déjà trié, on renvoie directement
2. Divise le tableau en 2 moitiés
3. Résout chaque moitié récursivement
4. Appelle la fonction merge et renvoie son résultat

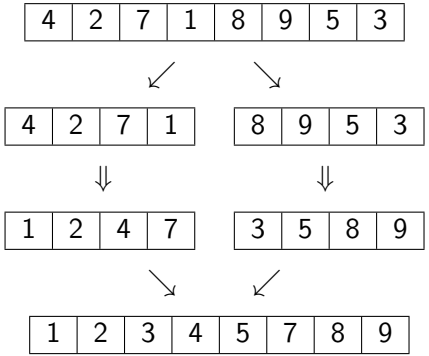


# Mergesort (tri fusion)



1. Si taille 1, c'est déjà trié, on renvoie directement
2. Divise le tableau en 2 moitiés
3. Résout chaque moitié récursivement
4. Appelle la fonction merge et renvoie son résultat

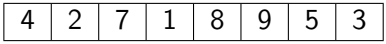
- Complexité $O(n \log n)$
- N'est pas inplace
- stable



# Quicksort (tri rapide)



1. Si taille 0 ou 1, c'est déjà trié, on renvoie directement



# Quicksort (tri rapide)



1. Si taille 0 ou 1, c'est déjà trié, on renvoie directement
2. Choisi un pivot $p$ du tableau

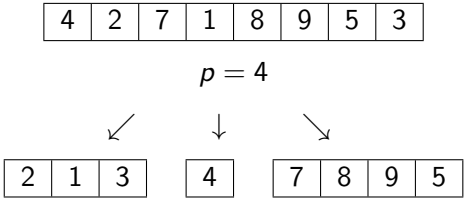
| 4 | 2 | 7 | 1 | 8 | 9 | 5 | 3 |
|---|---|---|---|---|---|---|---|

$p = 4$

# Quicksort (tri rapide)



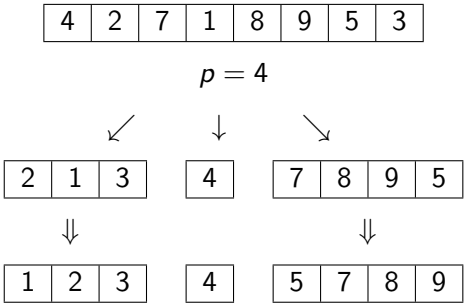
1. Si taille 0 ou 1, c'est déjà trié, on renvoie directement
2. Choisi un pivot $p$ du tableau
3. Divise le tableau en : $< p$, $= p$ et $> p$



# Quicksort (tri rapide)

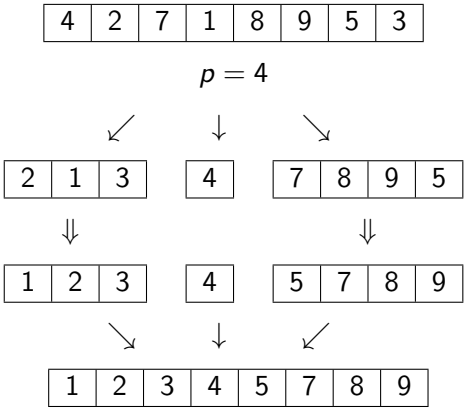


1. Si taille 0 ou 1, c'est déjà trié, on renvoie directement
2. Choisi un pivot $p$ du tableau
3. Divise le tableau en : $< p$, $= p$ et $> p$
4. Trie les deux tableaux récursivement



# Quicksort (tri rapide)

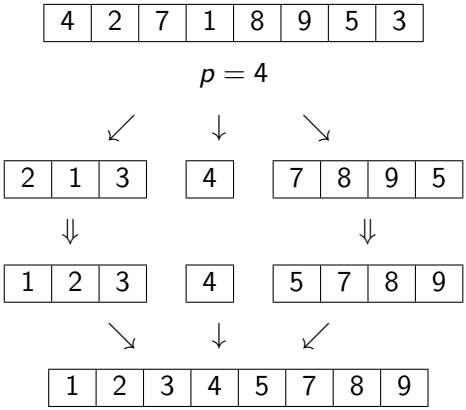
1. Si taille 0 ou 1, c'est déjà trié, on renvoie directement
2. Choisi un pivot $p$ du tableau
3. Divise le tableau en : $< p$, $= p$ et $> p$
4. Trie les deux tableaux récursivement
5. Concatène les trois tableaux



# Quicksort (tri rapide)

1. Si taille 0 ou 1, c'est déjà trié, on renvoie directement
2. Choisi un pivot $p$ du tableau
3. Divise le tableau en : $< p$, $= p$ et $> p$
4. Trie les deux tableaux récursivement
5. Concatène les trois tableaux

- Complexité randomizée $O(n \log n)$ si le pivot est choisi aléatoirement
- Peut être inplace ou stable, mais pas les deux



# Avenir

- Vous pouvez voir des animations des algorithmes de tri sur : `https://brett-desbenoit.pedaweb.univ-amu.fr/extranet/cours/slides/S1.02/`
- Bien travaillez sur les SAÉ, elles ont une notation très importante
- En S2 on continue le C++ et on va voir comment écrire :
  - des classes d'objets comme `std::string`
  - des templates comme `std::vector<?>`
  - des interfaces graphiques
- Et comment organiser le code orienté objet avec héritage et polymorphisme