Algorithmie

Les tris

BOULCH Alexandre

ONERA
THE FRENCH AEROSPACE LAB

Plan de la séance

Deux complexités différentes

- **La complexité en temps** : le nombre d'opérations élémentaires nécessaires pour effectuer l'algorithme
- **La complexité en espace** : la taille de la mémoire nécessaire à l'algorithme

Complexités classiques (en temps)

- $0(1)$: accès aux éléments d'un tableau
- $0(\log N)$: requête dans un arbre de recherche
- $0(N)$: parcours d'un tableau
- $0(N \log N)$: tris rapides
- $0(N^2)$: tris basiques
- $0(2^N)$: problèmes difficiles

Exemple : la suite de Fibonacci

Dans le cours d'introduction au C++, on a vu deux méthodes pour trouver les termes de la suite de Fibonnacci :

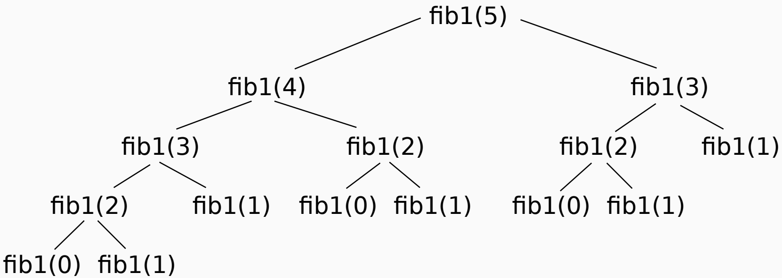
$$\begin{cases} f_0 & = f_1 = 1 \\ f_n & = f_{n-1} + f_{n-2} \end{cases}$$

Exemple : la suite de Fibonacci 2

La formulation du problème incite à l'utilisation de la récursivité.

```
int fib1(int n){
  if(n<2)
    return 1;
  return fib1(n-1)+fib1(n-2);
}
```

Exemple : la suite de Fibonacci 3



Exemple : la suite de Fibonacci 4

Ici on assimile la complexité au nombre d'additions (environ le nombre d'appels à la fonction).

- `fib1(0)` : 0
- `fib1(1)` : 0
- `fib1(2)` : 1
- `fib1(3)` : 2
- `fib1(4)` : 4
- `fib1(5)` : 7
- `fib1(6)` : 12
- `fib1(12)` : 20

Notant $a(n)$ le nombre d'additions à faire au rang n :

$$\begin{array}{ccccc} 2 * a(n-2) & \leq & a(n) & \leq & 2 * a(n-1) \\ 2^{\frac{n}{2}} & \leq & a(n) & \leq & 2^n \end{array}$$

Ceci donne une complexité $C(fib1)$ telle que :

$$O(2^{\frac{n}{2}}) \leq C(fib1) \leq O(2^n)$$

Impossible à calculer pour des n grands.

Exemple : la suite de Fibonacci 6

Une seconde méthode, non récursive :

```
int fib2(int n){
  int fnm2=1,fnm1=1 ;
  for(int i=2;i<=n;i++) {
    int fn=fnm2+fnm1;
    fnm2=fnm1;
    fnm1=fn;
  }
  return fnm1;
}
```

Ici on a une seule boucle. La boucle se compose d'opérations en temps constant.

La complexité $C(fib2)$ est $O(n)$.

Verdict
Le choix de la méthode d'implémentation peut beaucoup influer sur la performance.

Remarque
La récursivité n'est pas une mauvaise chose, elle est utile quand elle recalcule pas plusieurs fois la même chose.

Complexité exemples

https://fr.wikipedia.org/wiki/Analyse_de_la_complexit%C3%A9_des_algorithmes

Plan de la séance

Complexité minimale

La **complexité minimale** d'un algorithme est la complexité pour laquelle une preuve existe qu'on ne peut pas faire mieux.

Théorème
La complexité minimale d'un algorithme de tri, d'une liste $\{a_1, a_2, ...a_n\}$ à valeur dans un ensemble continu ou de grand cardinal est $O(n \log n)$.

- Tout algorithme de tri se base sur des comparaisons et des transpositions (le nombre de comparaisons est ici la complexité).
- Il doit être capable de trier quelque soit la liste en entrée, *ie* il doit pouvoir envisager les $n!$ permutations possibles

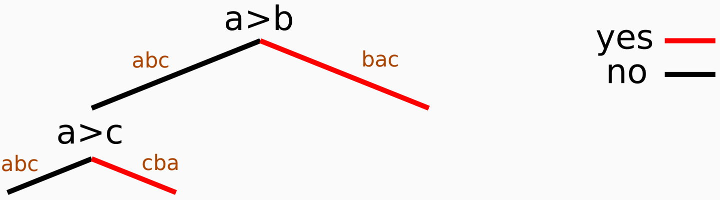
Lemme
On peut représenter un algorithme de tri sous la forme d'un arbre ou chaque noeud correspond à une comparaison, les arètes aux résultats des comparaisons et chaque feuille à une permutation.

Conséquences

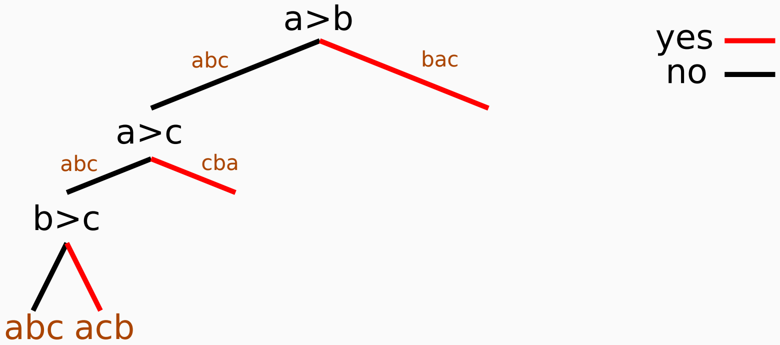
- L'arbre est un arbre binaire, il a 2^h feuilles (h est la hauteur de l'arbre)
- L'arbre a au minimum $n!$ feuilles
- La hauteur de l'arbre est le nombre de comparaisons nécessaires pour obtenir une liste triée

Exemple pour $n = 3$ et pour le tri à bulles : **abc**.

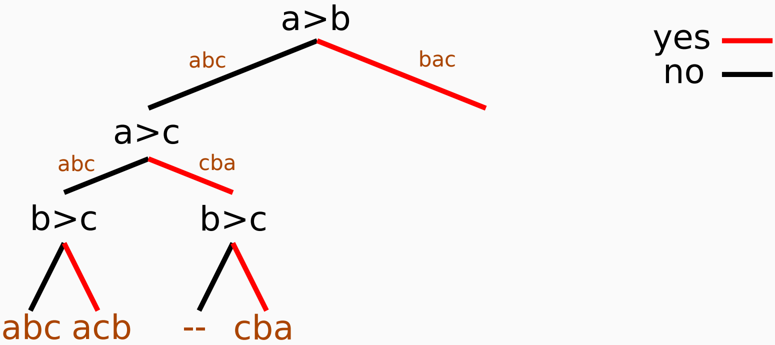
Exemple pour $n = 3$ et pour le tri à bulles : **abc**.



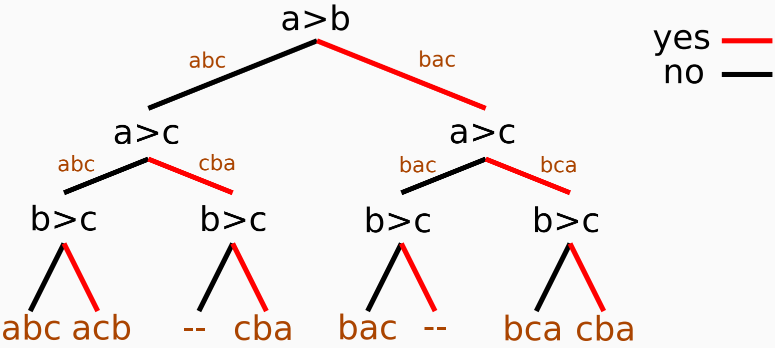
Exemple pour $n = 3$ et pour le tri à bulles : **abc**.



Exemple pour $n = 3$ et pour le tri à bulles : **abc**.



Exemple pour $n = 3$ et pour le tri à bulles : **abc**.



Complexité minimale

On a donc la relation suivante :

$$n! \leq 2^h$$

En utilisant la formule de Stirling : $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

On obtient :

$$h = O(n \log n)$$

Complexité minimale : exemple

Exemple du calcul de l'histogramme d'une image.

```
int histo[256];
for(int i=0; i<256; i++){
    histo[i] = 0;
}
for(int x=0; x<image.width(); x++){
    for(int y=0; y<image.height(); y++){
        histo[image(x,y)]++;
    }
}
```

ONERA THE FRENCH AEROSPACE LAB

Exemple du calcul de l'histogramme d'une image.

```
int histo[256];
for(int i=0; i<256; i++){
    histo[i] = 0;
}
for(int x=0; x<image.width(); x++){
    for(int y=0; y<image.height(); y++){
        histo[image(x,y)]++;
    }
}
```

Complexité : $O(n)$. La complexité minimale n'est pas liée à l'implémentation mais à la **tâche à effectuer**.

Plan de la séance

Algorithmes quadratiques : tri à bulles

```
for(int i=N; i>0; i--){
  for(int j=0; j<i-1; j++){
    if(t[j] > t[j+1]){
      swap(t[j], j[j+1])
    }
  }
}
```

```
for(int i=N; i>0; i--){
  for(int j=0; j<i-1; j++){
    if(t[j] > t[j+1]){
      swap(t[j], j[j+1])
    }
  }
}
```

Complexité

On fait $(N-1)+(N-2)+\ldots+1=\frac{(N-1)(N-2)}{6}$ comparaisons.

La complexité du tri à bulles est en $O(N^2)$.

Plan de la séance

QuickSort : principe

1. Prendre un élément (le pivot) et le placer à sa place dans le tableau, de sorte qu'avant lui les éléments soient plus petits, et après lui plus grands.
2. On relance l'étape précédente sur chacune des parties du tableau.

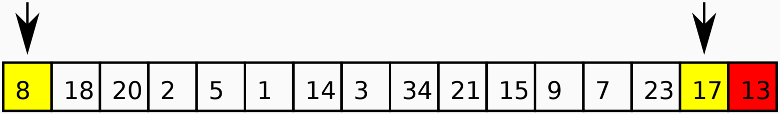
QuickSort : placer le pivot

8	18	20	2	5	1	14	3	34	21	15	9	7	23	17	13

QuickSort : placer le pivot

8	18	20	2	5	1	14	3	34	21	15	9	7	23	17	13

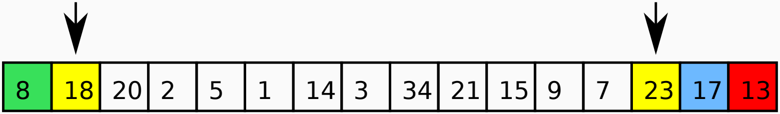
QuickSort : placer le pivot



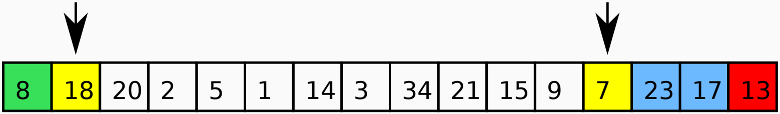
QuickSort : placer le pivot

8	18	20	2	5	1	14	3	34	21	15	9	7	23	17	13

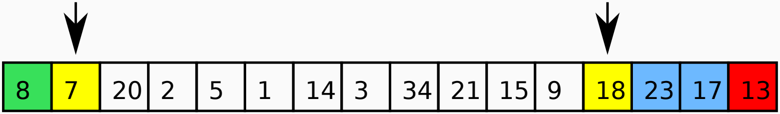
QuickSort : placer le pivot



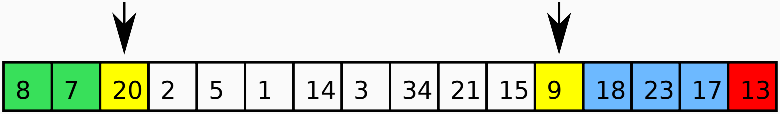
QuickSort : placer le pivot



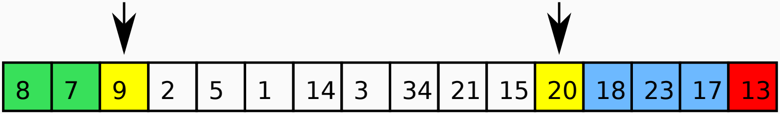
QuickSort : placer le pivot



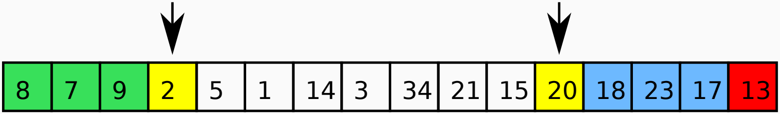
QuickSort : placer le pivot



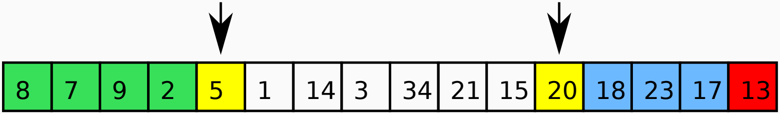
QuickSort : placer le pivot



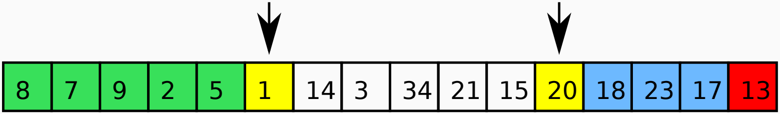
QuickSort : placer le pivot



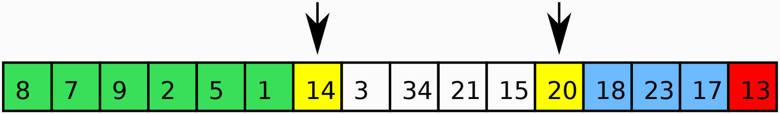
QuickSort : placer le pivot



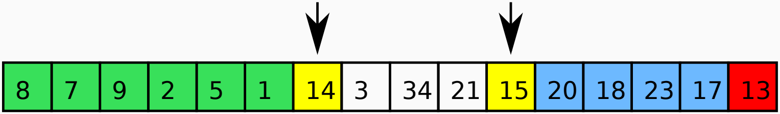
QuickSort : placer le pivot



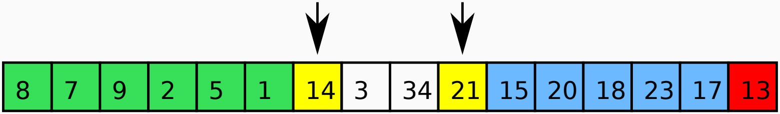
QuickSort : placer le pivot



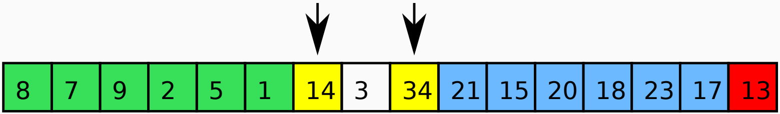
QuickSort : placer le pivot



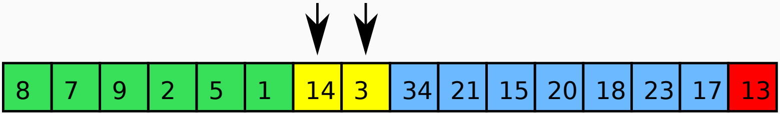
QuickSort : placer le pivot



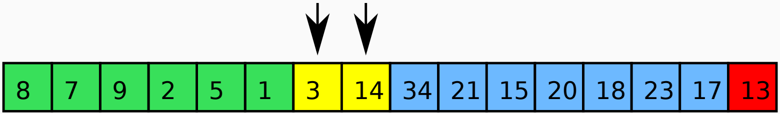
QuickSort : placer le pivot



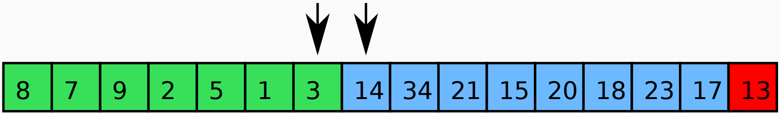
QuickSort : placer le pivot



QuickSort : placer le pivot



QuickSort : placer le pivot



QuickSort : placer le pivot

8	7	9	2	5	1	3	13	34	21	15	20	18	23	17	14

Complexité de Quicksort

Quicksort est un tri en $O(N \log(N))$.

Démonstration dans le chapitre 3.

QuickSort : complexité - Pire des cas

Le parcours du tableau implique $N-1$ comparaison. Donc :

$$C_N = (N-1) + C_i + C_{N-i-1}$$

Si on suppose que $i = N-1$ (déjà triée) :

$$C_N = (N-1) + C_{N-2}$$

Au rang suivant :

$$C_N = (N-1) + (N-2) + C_{N-3}$$

Au final, on a un tri à bulles :

$$C_n = O(N^2)$$

QuickSort : éviter le pire des cas

Pour eviter le pire des cas en moyenne on utilise généralement :

- un tirage du pivot au hasard
- un pivot au milieu du tableau
- un mélange de la liste au préalable

QuickSort : en pratique

- QuickSort est implémenté dans la STL (`#include <algorithm>`).
- Il existe des algorithmes en $N \log N$ quoi qu'il arrive (HeapSort, MergeSort ...), mais il sont moins rapide en moyenne.

File de priorité

La file de priorité est une structure de données :

- Accès à l'élément le plus prioritaire en $O(1)$
- Ajout d'un élément en $O(\log N)$
- Retrait d'un élément en $O(\log N)$

Étude au chapitre 4.

HeapSort remplit une file de priorité et puis retire les éléments un par un.

```
void HeapSort(std::vector<double> &v){
  FilePriorite f;
  for(int i=0; i<v.size(); i++){
    f.push(v[i]);
  }
  for(int i=0; i<v.size(); i++){
    v[i] = f.pop();
  }
}
```

HeapSort est un tri en $O(N \log N)$ dans tous les cas. Cependant en comparaison à QuickSort, il utilise plus de mémoire et est plus long en moyenne.

En pratique c'est QuickSort le plus utilisé.

Heapsort : complexités à retenir

- Tri : $O(N \log N)$
- Recherche dans un tableau trié : $O(\log N)$
- Recherche dans un tableau non trié : $O(N)$

Plan de la séance

Rappels

Complexité Minimale

Algorithmes quadratiques

QuickSort

Heapsort

Recherche dans un tableau

Recherche dans un tableau non trié

Tableau non trié
Pas d'a priori sur la structure du tableau. Il faut regarder chaque élément.

Complexité

$$O(N)$$

Recherche dichotomique

Le fait de savoir que le tableau est trié permet de réduire la complexité de la recherche à $O(\log(N))$.

```
int dichotomie(const std::vector<double>& V, double val){
    int a=0, b=v.size()-1;
    while(a<b){
      int c = (a+b)/2;
      if(V[c]==val)
          return c;
      if(V[c]<val)
          a = c+1;
      else
          b = c-1;
  }
  return (V[a]== val) ? a:-1;
}
```

Exemple - Tri à peigne

Algorithme
Comme le tri à bulle, mais on commence avec un intervalle de N (tri à bulle c'est 1). Puis on réduit l'intervalle par un facteur α (en général, empiriquement, $\alpha = 1.3$).

- beaucoup plus efficace que le tri à bulles
- tri en place (pas d'allocation de nouveau tableau)
- complexité pire des cas : $O(n^2)$
- Meilleur des cas : $O(n \log n)$

Algorithmes de tri.