

Algorithmie - Les tris

Nicolas Audebert

Mercredi 22 février 2017



Rappels

Complexité minimale

Algorithmes quadratiques

QuickSort

Tri par tas

Recherche dans un tableau

Deux complexités différentes

- ▶ La **complexité en temps** : le nombre d'opérations élémentaires constituant l'algorithme.
- ▶ La **complexité en espace** : le nombre de cases mémoires élémentaires occupées lors du déroulement de l'algorithme.

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

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

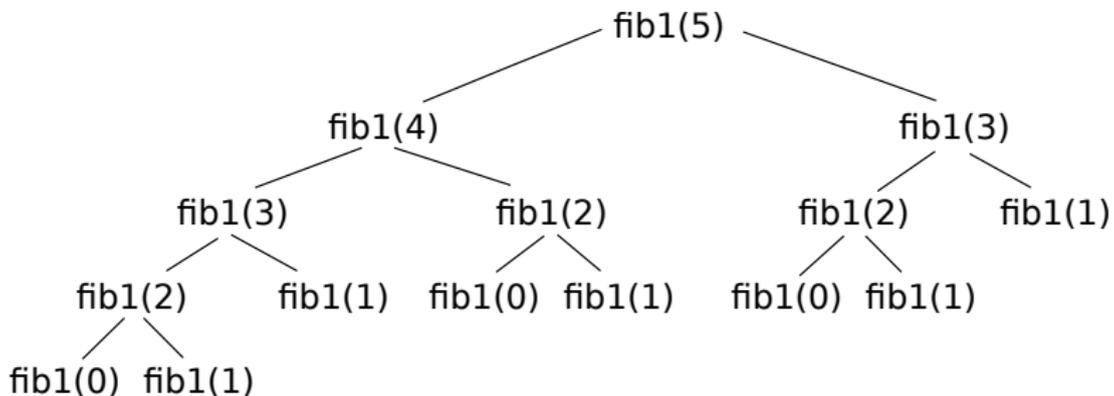
$$\begin{cases} f_0 = f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \end{cases}$$

Exemple : la suite de Fibonacci

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

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

Exemple : la suite de Fibonacci



Opération élémentaire = addition (+).

La complexité se mesure ici en nombre d'additions (*i.e.* en nombre d'appels à la fonction).

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

Exemple : la suite de Fibonacci

Si A_n représente le nombre d'additions à faire au rang n :

$$\begin{aligned} 2 \times A_{n-2} &\leq A_n \leq 2 \times A_{n-1} \\ 2^{\frac{n}{2}} &\leq A_n \leq 2^n \end{aligned}$$

Ceci donne une complexité $C_{\text{fibonacci}}$ telle que :

$$O(2^{\frac{n}{2}}) \leq C_{\text{fibonacci}} \leq O(2^n)$$

En pratique...

Impossible à calculer pour des n grands.

Une seconde méthode, non récursive :

```
int fibonacci(int n){
    // Initialisation des deux premiers termes
    int fn_m2 = 1, fn_m1 = 1 ;
    for(int i=2; i <= n; i++) {
        int fn = fn_m2 + fn_m1
        // Décalage du rang n-1 au rang n
        fn_m2 = fn_m1;
        fn_m1 = fn;
    }
    return fn_m1;
}
```

L'algorithme ainsi réécrit ne comporte qu'une seule boucle constituée uniquement d'opérations en temps constant.

La complexité $C_{\text{fibonacci}}$ est en $O(n)$.

Verdict

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

Remarque

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

Rappels

Complexité minimale

Algorithmes quadratiques

QuickSort

Tri par tas

Recherche dans un tableau

Théorème

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

Propriétés d'un algorithme de tri

- ▶ Tout algorithme de tri peut se ramener avec une succession de comparaisons et des transpositions (le nombre de comparaisons correspond ici à la mesure de complexité en temps).
- ▶ Tout algorithme de tri doit être capable de trier quelque soit la liste en entrée, *i.e.* il doit pouvoir envisager les $n!$ permutations possibles.

Lemme

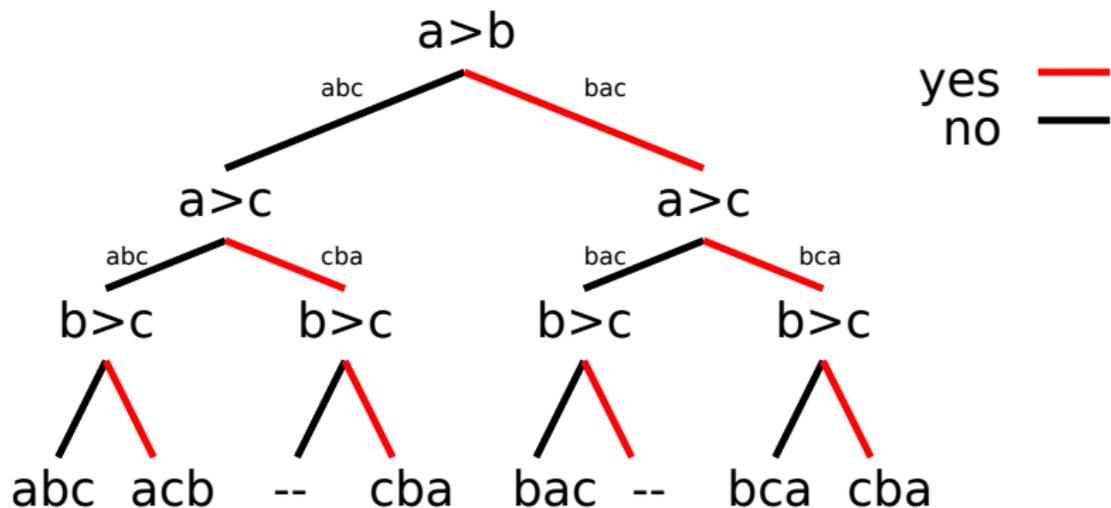
On peut représenter un algorithme de tri sous la forme d'un arbre où chaque noeud correspond à une comparaison, les arêtes aux résultats des comparaisons et chaque feuille à une permutation.

Conséquences

- ▶ L'arbre est un arbre binaire, il a 2^h feuilles, où 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.

Complexité minimale : preuve - Arbre de tri - Exemple

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



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 :

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

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)]++;
    }
}
```

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)]++;
    }
}
```

Chaque pixel doit être observé au moins une fois : $O(n)$.
La complexité minimale n'est pas liée à l'implémentation mais à la tâche à effectuer.

Rappels

Complexité minimale

Algorithmes quadratiques

QuickSort

Tri par tas

Recherche dans un tableau

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

Complexité

On fait $(n-1) + (n-2) + \dots + 1 = \frac{(n-1)(n-2)}{2}$ comparaisons.
La complexité du tri à bulles est en $O(n^2)$.

Rappels

Complexité minimale

Algorithmes quadratiques

QuickSort

Tri par tas

Recherche dans un tableau

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

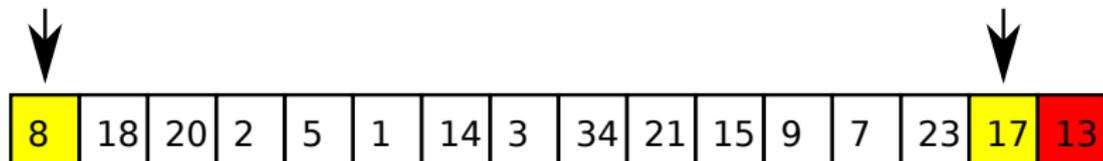
QuickSort : placer le pivot

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

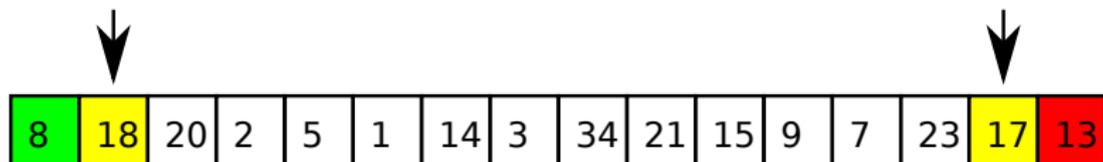
QuickSort : placer le pivot

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

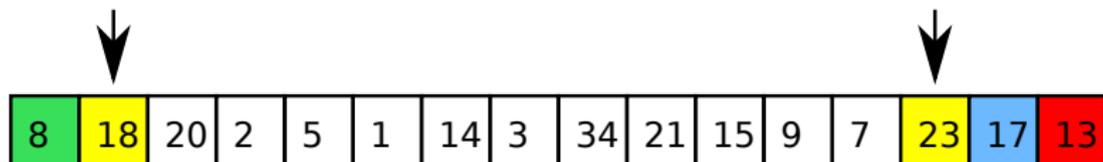
QuickSort : placer le pivot



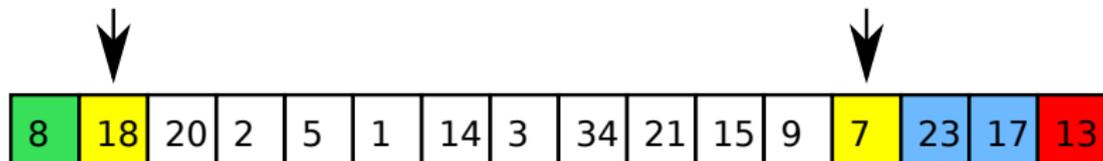
QuickSort : placer le pivot



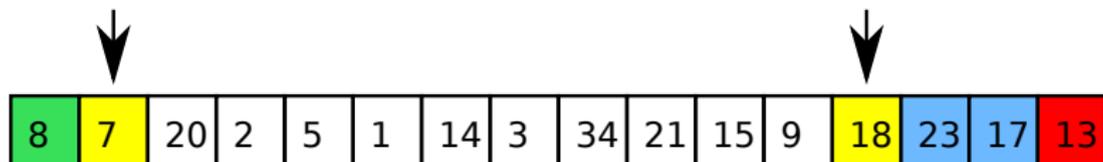
QuickSort : placer le pivot



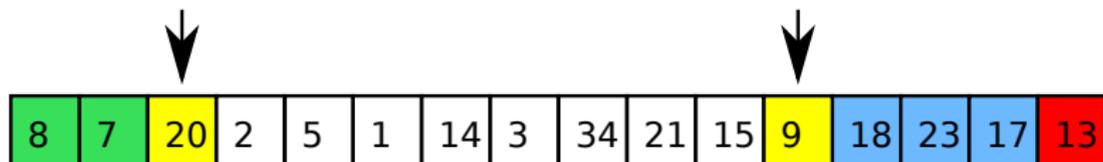
QuickSort : placer le pivot



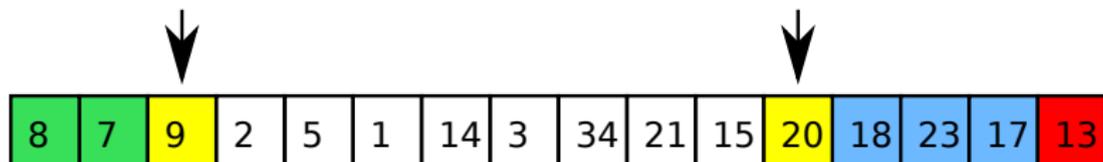
QuickSort : placer le pivot



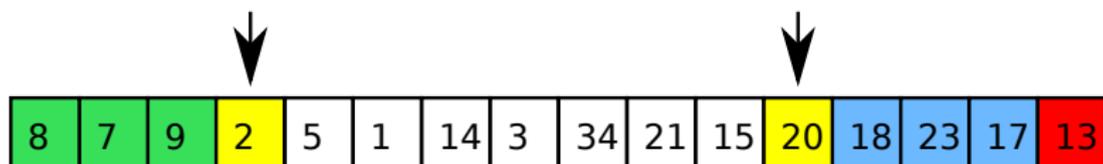
QuickSort : placer le pivot



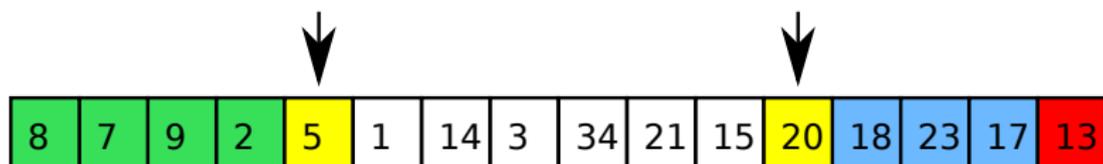
QuickSort : placer le pivot



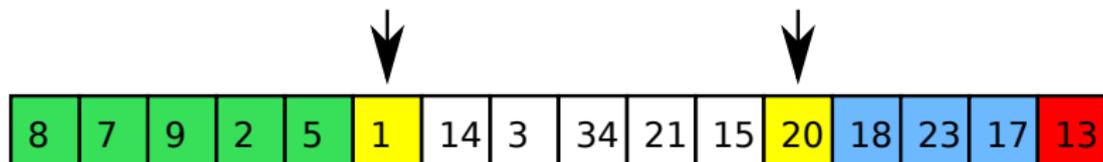
QuickSort : placer le pivot



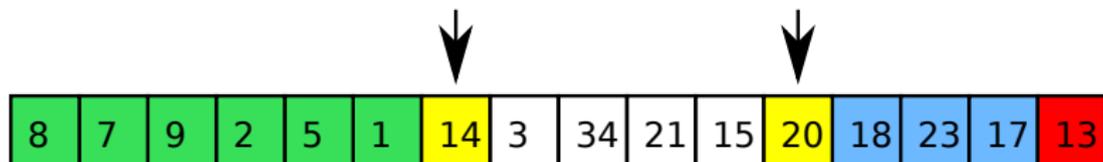
QuickSort : placer le pivot



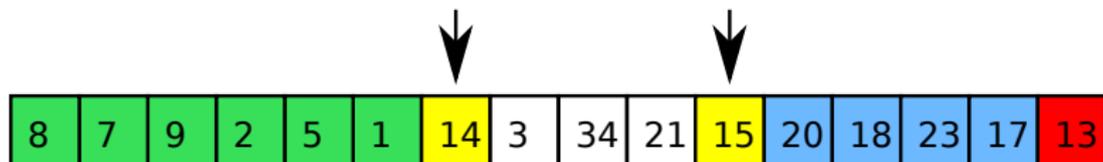
QuickSort : placer le pivot



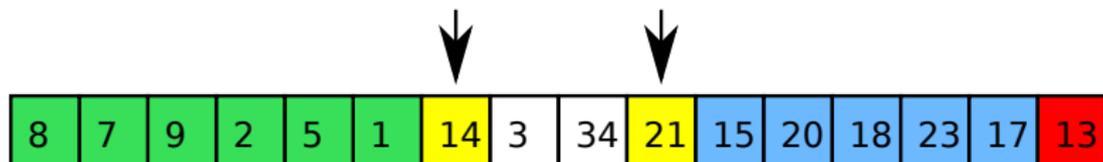
QuickSort : placer le pivot



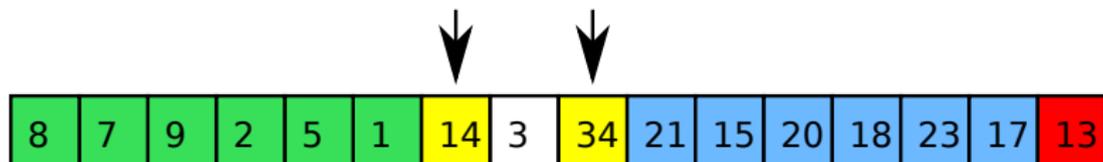
QuickSort : placer le pivot



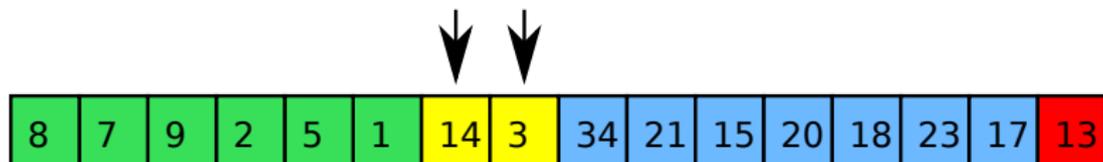
QuickSort : placer le pivot



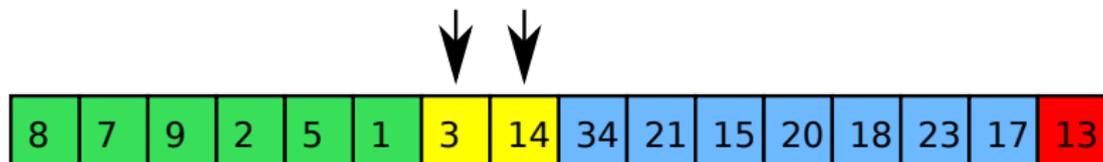
QuickSort : placer le pivot



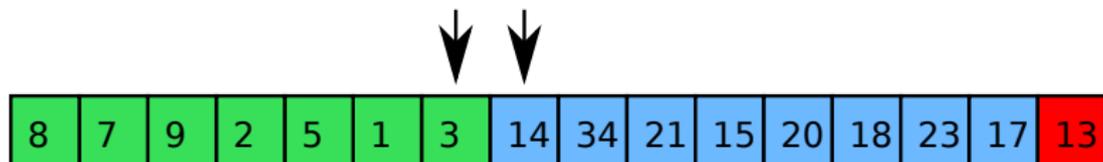
QuickSort : placer le pivot



QuickSort : placer le pivot



QuickSort : placer le pivot



QuickSort : placer le pivot



Quicksort est un tri en $O(n \log(n))$ **en moyenne**.

Démonstration dans le chapitre 3.

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 :

$$C_n = (n - 1) + (n - 2) + C_{n-3}$$

En fait, cela revient à effectuer un tri à bulles :

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

Pour éviter 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 est implémenté dans la STL (`#include <algorithm>`).
- ▶ Il existe des algorithmes en $n \log n$ quoi qu'il arrive (tri par tas, tri fusion, ...), mais ils sont moins rapides **en moyenne**.

Rappels

Complexité minimale

Algorithmes quadratiques

QuickSort

Tri par tas

Recherche dans un tableau

La file de priorité est une structure de données permettant :

- ▶ 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.

Le tri par tas 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();
    }
}
```

Le tri par tas 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.

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

Rappels

Complexité minimale

Algorithmes quadratiques

QuickSort

Tri par tas

Recherche dans un tableau

Tableau non trié

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

Complexité

$$O(n)$$

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 debut = 0, fin = v.size() - 1;
    while(debut < fin){
        int milieu = (debut + fin)/2;
        if(V[milieu] == val)
            return milieu;
        if(V[milieu] < val){
            debut = milieu + 1;
        } else {
            fin = milieu - 1;
        }
    }
    // On renvoie l'indice actuel si c'est la bonne valeur
    // ou -1 sinon car la valeur n'est pas dans le vecteur
    return (V[milieu] == val) ? a:-1;
}
```

Wikipédia

https://fr.wikipedia.org/wiki/Algorithme_de_tri

TP

Algorithmes de tri.