Algorithmique et structures de données

Algorithmes de tri

Julien Hauret

Lundi 30 janvier 2023

Plan de la séance

Rappels

Complexité minimale

Algorithmes quadratiques

QuickSort : le tri rapide

Tri par tas

Recherche dans un tableau

Les types de complexité

Deux complexités différentes

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

Complexités classiques (en temps)

- $O(1)$: accès aux éléments d'un tableau ;
- $O(\log n)$: recherche d'un élément dans une liste triée ;
- $O(n)$: parcours d'un tableau ;
- $O(n \log n)$: tris rapides ;
- $O(n^2)$: tris basiques ;
- $O(2^n)$: 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 :

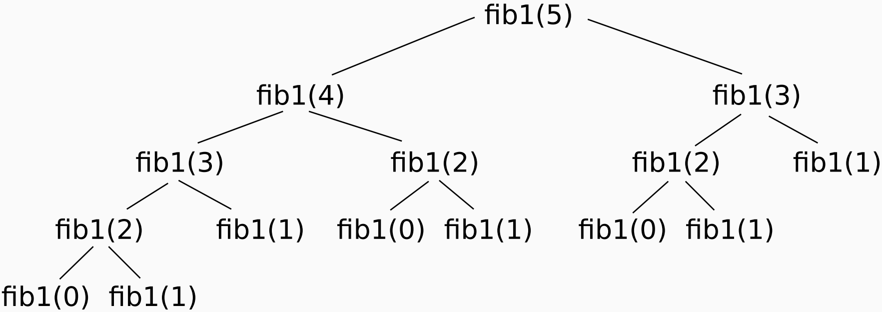
$$\begin{cases} f_0 & = f_1 = 1 \\ f_n & = f_{n-1} + f_{n-2} \end{cases}$$

Exemple : la suite de Fibonacci

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



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 :

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

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.

Exemple : la suite de Fibonacci

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 fnm1;
}
```

Exemple : la suite de Fibonacci

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 l'implémentation d'un même calcul peut beaucoup influer sur la performance.

Remarque

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

Plan de la séance

Rappels

Complexité minimale

Algorithmes quadratiques

QuickSort : le tri rapide

Tri par tas

Recherche dans un tableau

Complexité minimale

Théorème

Soit $L = \{a_1, a_2, \ldots, a_n\}$ un ensemble de n valeurs dans E un **ensemble continu** ou de grand cardinal.

La **complexité minimale** d'un algorithme de tri prenant en entrée L et renvoyant en sortie les valeurs a_i ordonnées par ordre croissant est $\Theta(n \log n)$ (linéarithmique).

Complexité minimale : preuve

Propriétés d'un algorithme de tri

1. Tout algorithme de tri peut se ramener à une succession de comparaisons et de transpositions d'éléments,
   - l'opération élémentaire pour la complexité en temps sera la comparaison entre deux éléments du tableau.
2. Tout algorithme de tri doit être capable de trier n'importe quelle liste arbitraire, c'est-à-dire de trier les $n!$ permutations possibles de n'importe quelle liste.

Complexité minimale : preuve - Arbre de tri

Lemme

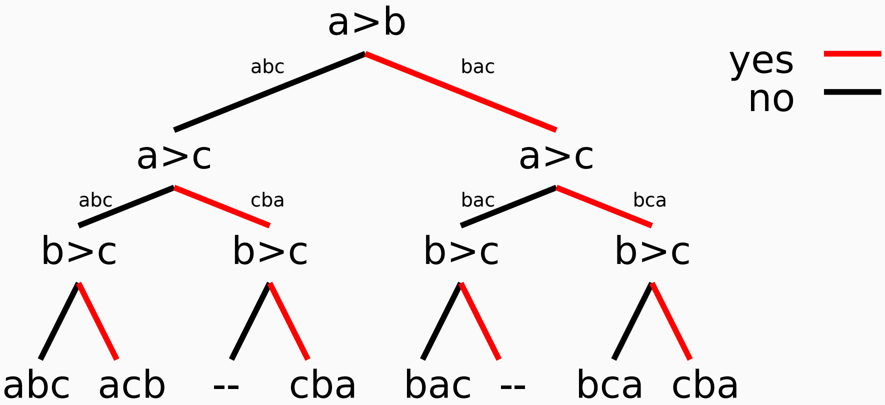
On peut représenter un algorithme de tri sous la forme d'un arbre :

- chaque nœud correspond à une comparaison,
- chaque nœud a deux arêtes, une pour chaque résultat de la comparaison,
- chaque feuille est une permutation possible de la liste d'entrée.

Conséquences

- L'arbre est un arbre binaire de hauteur h à 2^h feuilles.
- 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**.



Complexité minimale des tris (1/2)

Chaque feuille de l'arbre est :

- soit vide car correspond à un ordonnancement impossible,
- soit une des permutations possibles de la liste.

Par conséquent, le nombre de feuilles de l'arbre est **supérieur** au nombre de permutations de la liste d'entrée

$$n! \leq 2^h$$

ce qui implique $\log_2(n!) \leq h$.

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

$$h \geq n \cdot \log_2(n) - \frac{n}{\ln 2} = \Theta(n \cdot \log_2 n)$$

Complexité minimale des tris (1/2)

Chaque feuille de l'arbre est :

- soit vide car correspond à un ordonnancement impossible,
- soit une des permutations possibles de la liste.

Par conséquent, le nombre de feuilles de l'arbre est **supérieur** au nombre de permutations de la liste d'entrée

$$n! \leq 2^h$$

ce qui implique $\log_2(n!) \leq h$.

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

$$h \geq n \cdot \log_2(n) - \frac{n}{\ln 2} = \Theta(n \cdot \log_2 n)$$

Complexité minimale des tris (2/2)

D'une part, nous venons de montrer que

$$h \geq \Theta(n \cdot \log_2 n) \quad .$$

D'autre part, il existe des algorithmes de tri de complexité $\Theta(n \log n)$ donc $h \geq \Theta(n \cdot \log n)$.

Finalement, il vient :

$$\boxed{h = \Theta(n \log n).}$$

h étant la hauteur de l'arbre mais aussi le nombre de comparaisons entre éléments de la liste.

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)]++;
    }
}
```

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.

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)]++;
    }
}
```

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.

Remarque à méditer

Le théorème de la complexité minimale des algorithmes de tri n'est valable que pour des tableaux à valeurs dans de grands ensembles (de cardinal infini ou presque).

Exercice

- Proposer un algorithme de tri en $O(n)$ pour un tableau de n éléments à valeurs entières dans $[\![0, k]\!]$.

Plan de la séance

Le 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], t[j+1]);
        }
    }
}
```

Complexité

Le tri à bulles réalise $(n-1)+(n-2)+\ldots+1 = \frac{(n-1)(n-2)}{2}$ comparaisons.

La complexité du tri à bulles est en $O(n^2)$ en moyenne et dans le pire des cas.

Autres algorithmes classiques en $O(n^2)$

Le tri par insertion et le tri par sélection (cf. TP).

Plan de la séance

QuickSort : principe

1. Choisir un élément du tableau, il devient **le pivot**.
2. Placer le pivot à la position i de sorte que tous les éléments d'indice inférieurs à i soient plus petits que le pivot et tous les éléments d'indice supérieurs à i soient plus grands.
3. Réitérer le procédé sur chacune des deux sous-parties du tableau.

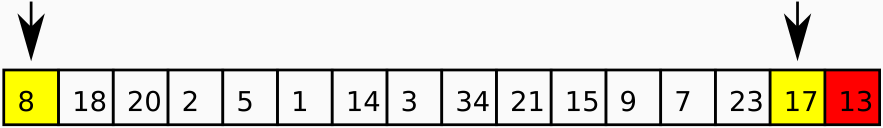
QuickSort : placer le pivot

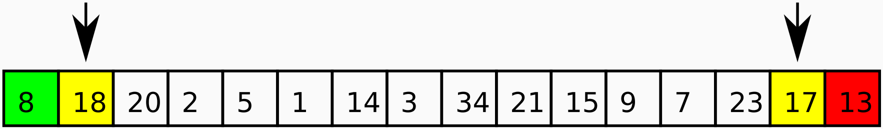
8	18	20	2	5	1	14	3	34	21	15	9	7	23	17	13

QuickSort : placer le pivot

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

QuickSort : placer le pivot

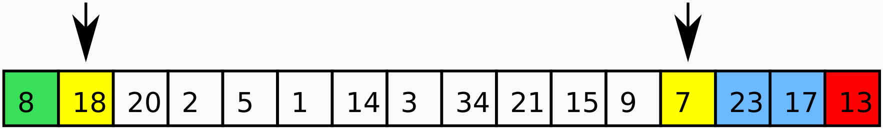



8 18 20 2 5 1 14 3 34 21 15 9 7 23 17 13

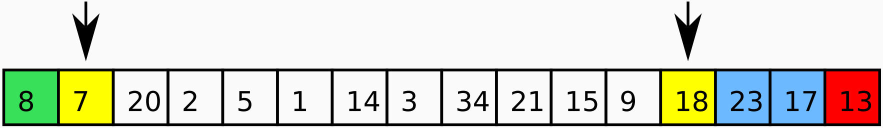
QuickSort : placer le pivot

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

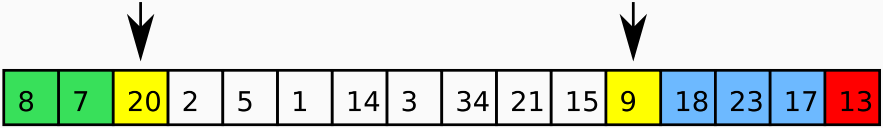
QuickSort : placer le pivot



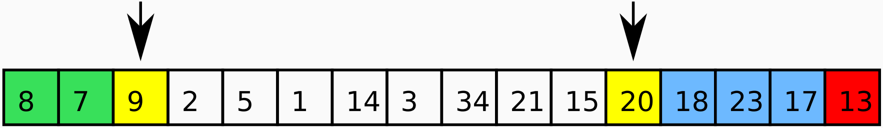
QuickSort : placer le pivot



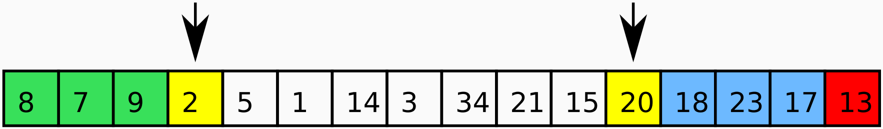
QuickSort : placer le pivot



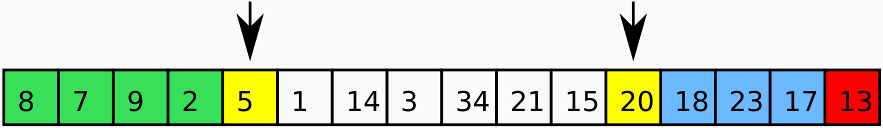
QuickSort : placer le pivot



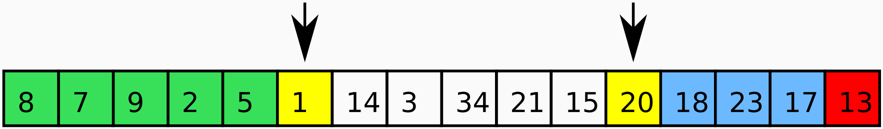
QuickSort : placer le pivot



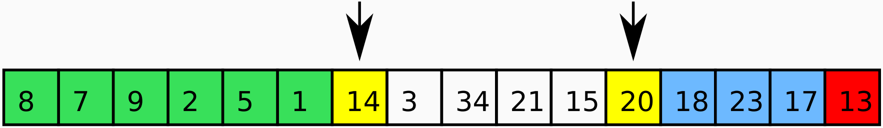
QuickSort : placer le pivot

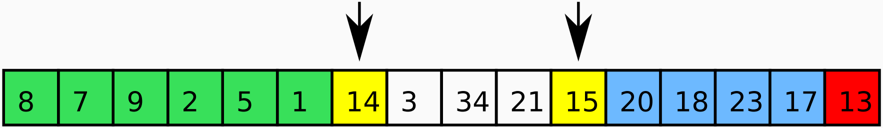


QuickSort : placer le pivot

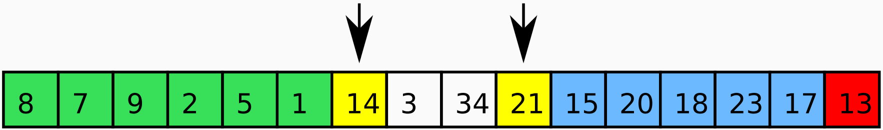


QuickSort : placer le pivot

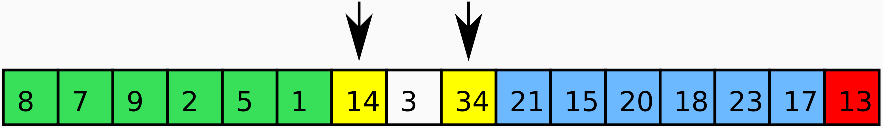



8
7
9
2
5
1
14
3
34
21
15
20
18
23
17
13

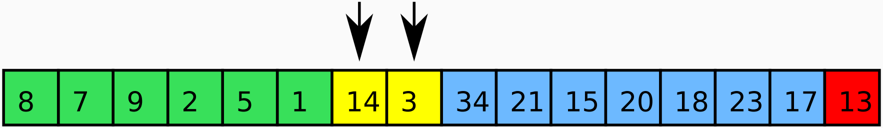
QuickSort : placer le pivot



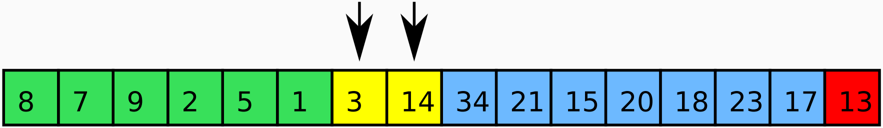
QuickSort : placer le pivot



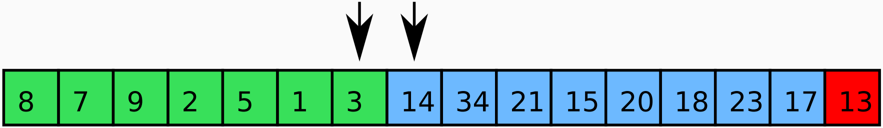
QuickSort : placer le pivot



QuickSort : placer le pivot



QuickSort : placer le pivot



QuickSort : placer le pivot

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

Complexité de Quicksort

Théorème

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

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) :

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

Au rang suivant :

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

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

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

QuickSort : éviter le pire des cas

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 : en pratique

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

Plan de la séance

File de priorité

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 de la file de priorité au chapitre 4.

Tri par tas

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();
  }
}
```

Conclusion

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 par tas : 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

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 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;
}
```

Codons!

Travaux pratiques

Implémentation de quelques algorithmes de tri en C++.