

Tris et complexité

Algorithmes et structures de données, 2025-2026

P. Albuquerque (B410) et O. Malaspinas (A401), ISC, HEPIA
2025-11-18

En partie inspiré des supports de cours de P. Albuquerque

Tri rapide ou quicksort

Tri rapide ou quicksort (1/8)

Idée : algorithme diviser pour régner (divide-and-conquer)

- Diviser : découper un problème en sous problèmes ;
- Régner : résoudre les sous-problèmes (souvent récursivement) ;
- Combiner : à partir des sous problèmes résolu, calculer la solution.

Le pivot

- Trouver le **pivot**, un élément qui divise le tableau en 2, tels que :
 1. Éléments à gauche sont **plus petits** que le pivot.
 2. Éléments à droite sont **plus grands** que le pivot.

Algorithme quicksort(tableau)

1. Choisir le pivot et l'amener à sa place :
 - Les éléments à gauche sont plus petits que le pivot.
 - Les éléments à droite sont plus grand que le pivot.
2. quicksort(tableau_gauche) en omettant le pivot.
3. quicksort(tableau_droite) en omettant le pivot.
4. S'il y a moins de deux éléments dans le tableau, le tableau est trié.

Algorithme quicksort(tableau)

1. Choisir le pivot et l'amener à sa place :
 - Les éléments à gauche sont plus petits que le pivot.
 - Les éléments à droite sont plus grand que le pivot.
2. quicksort(tableau_gauche) en omettant le pivot.
3. quicksort(tableau_droite) en omettant le pivot.
4. S'il y a moins de deux éléments dans le tableau, le tableau est trié.

Compris ?

Tri rapide ou quicksort (2/8)

Algorithme quicksort(tableau)

1. Choisir le pivot et l'amener à sa place :
 - Les éléments à gauche sont plus petits que le pivot.
 - Les éléments à droite sont plus grand que le pivot.
2. quicksort(tableau_gauche) en omettant le pivot.
3. quicksort(tableau_droite) en omettant le pivot.
4. S'il y a moins de deux éléments dans le tableau, le tableau est trié.

Compris ?

Non c'est normal, faisons un exemple.

Tri rapide ou quicksort (3/8)

Deux variables sont primordiales :

```
entier ind_min, ind_max; // les indices min/max des tableaux à trier
```

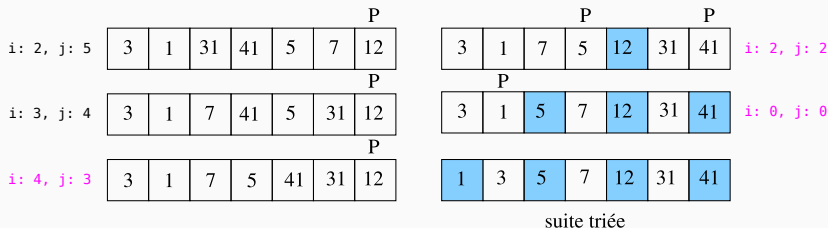


Figure 1 : Un exemple de quicksort.

Tri rapide ou quicksort (4/8)

Deux variables sont primordiales :

```
entier ind_min, ind_max; // les indices min/max des tableaux à trier
```

Pseudocode : quicksort

```
rien quicksort(entier tableau[], entier ind_min, entier ind_max)
    si (longueur(tab) > 1)
        ind_pivot = partition(tableau, ind_min, ind_max)
        si (longueur(tableau[ind_min:ind_pivot-1]) != 0)
            quicksort(tableau, ind_min, ind_pivot - 1)
        si (longueur(tableau[ind_pivot+1:ind_max]) != 0)
            quicksort(tableau, ind_pivot + 1, ind_max)
```


Tri rapide ou quicksort (5/8)

Pseudocode : partition

```
entier partition(entier tableau[], entier ind_min, entier ind_max)
    pivot = tableau[ind_max] // choix arbitraire
    i = ind_min
    j = ind_max-1
    tant que i < j:
        en remontant i trouver le premier élément > pivot
        en descendant j trouver le premier élément < pivot
        échanger(tableau[i], tableau[j])
        // les plus grands à droite
        // mettre les plus petits à gauche

    // on met le pivot "au milieu"
    échanger(tableau[i], tableau[ind_max])
    retourne i // on retourne l'indice pivot
```

Tri rapide ou quicksort (6/8)

Exercice : implémenter les fonctions quicksort et partition

Tri rapide ou quicksort (6/8)

Exercice : implémenter les fonctions quicksort et partition

```
void quicksort(int size, int array[size], int first,
               int last)
{
    if (first < last) {
        int midpoint = partition(size, array, first, last);
        if (first < midpoint - 1) {
            quicksort(size, array, first, midpoint - 1);
        }
        if (midpoint + 1 < last) {
            quicksort(size, array, midpoint + 1, last);
        }
    }
}
```

Tri rapide ou quicksort (7/8)

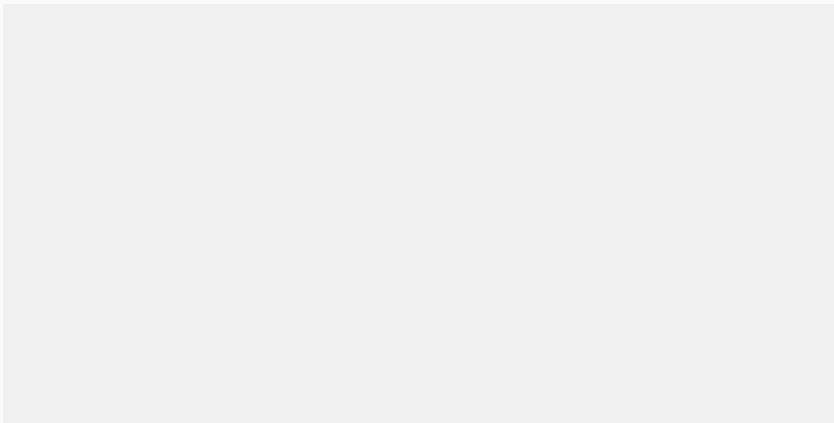
Exercice : implémenter les fonctions quicksort et partition

```
int partition(int size, int array[size], int first, int last) {
    int pivot = array[last];
    int i = first - 1, j = last;
    do {
        do {
            i += 1;
        } while (array[i] < pivot && i < j);
        do {
            j -= 1;
        } while (array[j] > pivot && i < j);
        if (j > i) {
            swap(&array[i], &array[j]);
        }
    } while (j > i);
    swap(&array[i], &array[last]);
    return i;
}
```

L'algorithme à la main

Exercice *sur papier*

- Trier par tri rapide le tableau [5, -2, 1, 3, 10, 15, 7, 4]



L'efficacité d'un algorithme

Efficacité d'un algorithmique

Comment mesurer l'efficacité d'un algorithme ?

Efficacité d'un algorithmique

Comment mesurer l'efficacité d'un algorithme ?

- Mesurer le temps CPU,
- Mesurer le temps d'accès à la mémoire,
- Mesurer la place prise mémoire,

Efficacité d'un algorithmique

Comment mesurer l'efficacité d'un algorithme ?

- Mesurer le temps CPU,
- Mesurer le temps d'accès à la mémoire,
- Mesurer la place prise mémoire,

Dépendant du **matériel**, du **compilateur**, des **options de compilation**, etc !

Mesure du temps CPU

```
#include <time.h>
struct timespec tstart={0,0}, tend={0,0};
clock_gettime(CLOCK_MONOTONIC, &tstart);
// some computation
clock_gettime(CLOCK_MONOTONIC, &tend);
printf("computation about %.5f seconds\n",
      ((double)tend.tv_sec + 1e-9*tend.tv_nsec) -
      ((double)tstart.tv_sec + 1e-9*tstart.tv_nsec));
```

Programme simple : mesure du temps CPU

Preuve sur un **petit exemple**

```
source_codes/complexity$ make bench
RUN ONCE -00
the computation took about 0.00836 seconds
RUN ONCE -03
the computation took about 0.00203 seconds
RUN THOUSAND TIMES -00
the computation took about 0.00363 seconds
RUN THOUSAND TIMES -03
the computation took about 0.00046 seconds
```

Et sur votre machine les résultats seront **différents**.

Programme simple : mesure du temps CPU

Preuve sur un **petit exemple**

```
source_codes/complexity$ make bench
RUN ONCE -00
the computation took about 0.00836 seconds
RUN ONCE -03
the computation took about 0.00203 seconds
RUN THOUSAND TIMES -00
the computation took about 0.00363 seconds
RUN THOUSAND TIMES -03
the computation took about 0.00046 seconds
```

Et sur votre machine les résultats seront **différents**.

Conclusion

- Nécessité d'avoir une mesure indépendante du/de la matériel/compilateur/façon de mesurer/météo.

Analyse de complexité algorithmique (1/4)

- On analyse le **temps** pris par un algorithme en fonction de la **taille** de l'entrée.

Exemple : recherche d'un élément dans une liste triée de taille N

```
int sorted_list[N];  
bool in_list = is_present(N, sorted_list, elem);
```

- Plus N est grand, plus l'algorithme prend de temps sauf si...

Analyse de complexité algorithmique (1/4)

- On analyse le **temps** pris par un algorithme en fonction de la **taille de l'entrée**.

Exemple : recherche d'un élément dans une liste triée de taille N

```
int sorted_list[N];  
bool in_list = is_present(N, sorted_list, elem);
```

- Plus N est grand, plus l'algorithme prend de temps sauf si...
- l'élément est le premier de la liste (ou à une position toujours la même).
- ce genre de cas pathologique ne rentre pas en ligne de compte.

Analyse de complexité algorithmique (2/4)

Recherche linéaire

```
bool is_present(int n, int tab[], int elem) {  
    for (int i = 0; i < n; ++i) {  
        if (tab[i] == elem) {  
            return true;  
        } else if (elem < tab[i]) {  
            return false;  
        }  
    }  
    return false;  
}
```

- Dans le **meilleurs des cas** il faut 1 comparaison.
- Dans le **pire des cas** (élément absent p.ex.) il faut n comparaisons.

Analyse de complexité algorithmique (2/4)

Recherche linéaire

```
bool is_present(int n, int tab[], int elem) {  
    for (int i = 0; i < n; ++i) {  
        if (tab[i] == elem) {  
            return true;  
        } else if (elem < tab[i]) {  
            return false;  
        }  
    }  
    return false;  
}
```

- Dans le **meilleurs des cas** il faut 1 comparaison.
- Dans le **pire des cas** (élément absent p.ex.) il faut n comparaisons.

La **complexité algorithmique** est proportionnelle à N : on double la taille du tableau \Rightarrow on double le temps pris par l'algorithme.

Analyse de complexité algorithmique (3/4)

Recherche dichotomique

```
bool is_present_binary_search(int n, int tab[], int elem) {  
    int left  = 0;  
    int right = n - 1;  
    while (left <= right) {  
        int mid = (right + left) / 2;  
        if (tab[mid] < elem) {  
            left = mid + 1;  
        } else if (tab[mid] > elem) {  
            right = mid - 1;  
        } else {  
            return true;  
        }  
    }  
    return false;  
}
```


Analyse de complexité algorithmique (4/4)

Recherche dichotomique

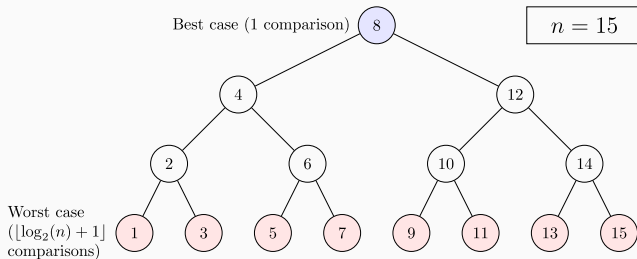


Figure 2 : Source : [Wikipédia](#)

Analyse de complexité algorithmique (4/4)

Recherche dichotomique

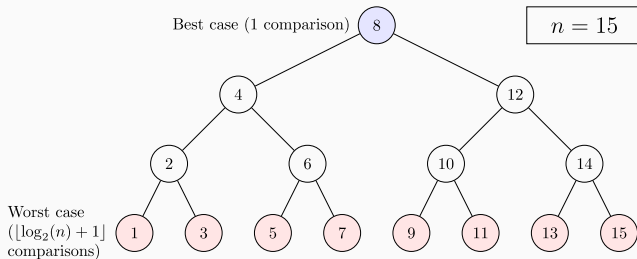


Figure 2 : Source : [Wikipédia](#)

- Dans le **meilleurs de cas** il faut 1 comparaison.
- Dans le **pire des cas** il faut $\log_2(N) + 1$ comparaisons

Analyse de complexité algorithmique (4/4)

Recherche dichotomique

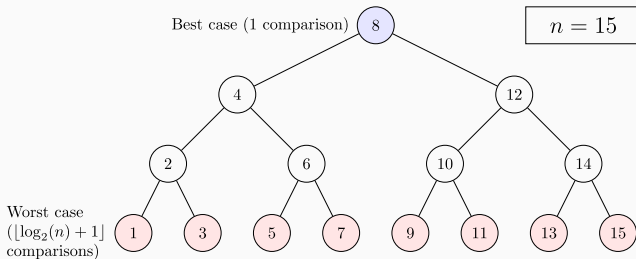


Figure 2 : Source : [Wikipédia](#)

- Dans le **meilleurs de cas** il faut 1 comparaison.
- Dans le **pire des cas** il faut $\log_2(N) + 1$ comparaisons

Linéaire vs dichotomique

- N vs $\log_2(N)$ comparaisons logiques.
- Pour $N = 1000000$: 1000000 vs 21 comparaisons.

Notation pour la complexité

Constante de proportionnalité

- Pour la recherche linéaire ou dichotomique, on a des algorithmes qui sont $\sim N$ ou $\sim \log_2(N)$
- Qu'est-ce que cela veut dire ?

Notation pour la complexité

Constante de proportionnalité

- Pour la recherche linéaire ou dichotomique, on a des algorithmes qui sont $\sim N$ ou $\sim \log_2(N)$
- Qu'est-ce que cela veut dire ?
- Temps de calcul est $t = C \cdot N$ (où C est le temps pris pour une comparaisons sur une machine/compilateur donné)
- La complexité ne dépend pas de C .

Le \mathcal{O} de Leibnitz

- Pour noter la complexité d'un algorithme on utilise le symbole \mathcal{O} (ou "grand Ô de").
- Les complexités les plus couramment rencontrées sont

Notation pour la complexité

Constante de proportionnalité

- Pour la recherche linéaire ou dichotomique, on a des algorithmes qui sont $\sim N$ ou $\sim \log_2(N)$
- Qu'est-ce que cela veut dire ?
- Temps de calcul est $t = C \cdot N$ (où C est le temps pris pour une comparaisons sur une machine/compilateur donné)
- La complexité ne dépend pas de C .

Le \mathcal{O} de Leibnitz

- Pour noter la complexité d'un algorithme on utilise le symbole \mathcal{O} (ou "grand Ô de").
- Les complexités les plus couramment rencontrées sont

$$\mathcal{O}(1), \quad \mathcal{O}(\log(N)), \quad \mathcal{O}(N), \quad \mathcal{O}(\log(N) \cdot N), \quad \mathcal{O}(N^2), \quad \mathcal{O}(N^3).$$

Ordres de grandeur

Table 1 : Valeurs approximatives de quelques fonctions usuelles de complexité.

$\log_2(N)$	\sqrt{N}	N	$N \log_2(N)$	N^2
3	3	10	30	10^2
6	10	10^2	$6 \cdot 10^2$	10^4
9	31	10^3	$9 \cdot 10^3$	10^6
13	10^2	10^4	$1.3 \cdot 10^5$	10^8
16	$3.1 \cdot 10^2$	10^5	$1.6 \cdot 10^6$	10^{10}
19	10^3	10^6	$1.9 \cdot 10^7$	10^{12}

Quelques exercices (1/3)

Complexité de l'algorithme de test de primalité naïf ?

```
for (i = 2; i < sqrt(N); ++i) {  
    if (N % i == 0) {  
        return false;  
    }  
}  
return true;
```


Quelques exercices (1/3)

Complexité de l'algorithme de test de primalité naïf ?

```
for (i = 2; i < sqrt(N); ++i) {  
    if (N % i == 0) {  
        return false;  
    }  
}  
return true;
```

Réponse

$$\mathcal{O}(\sqrt{N}).$$

Quelques exercices (2/3)

Complexité de trouver le minimum d'un tableau ?

```
int min = MAX;
for (i = 0; i < N; ++i) {
    if (tab[i] < min) {
        min = tab[i];
    }
}
return min;
```

Quelques exercices (2/3)

Complexité de trouver le minimum d'un tableau ?

```
int min = MAX;
for (i = 0; i < N; ++i) {
    if (tab[i] < min) {
        min = tab[i];
    }
}
return min;
```

Réponse

$\mathcal{O}(N)$.

Quelques exercices (3/3)

Complexité du tri par sélection ?

```
int ind = 0;
while (ind < SIZE-1) {
    min = find_min(tab[ind:SIZE]);
    swap(min, tab[ind]);
    ind += 1;
}
```

Quelques exercices (3/3)

Complexité du tri par sélection ?

```
int ind = 0;
while (ind < SIZE-1) {
    min = find_min(tab[ind:SIZE]);
    swap(min, tab[ind]);
    ind += 1;
}
```

Réponse

`min = find_min`

$$(N-1) + (N-2) + \dots + 2 + 1 = \sum_{i=1}^{N-1} i = N \cdot (N-1)/2 = \mathcal{O}(N^2).$$

Finalement