

# Récurtivité, représentation des nombres, tris, et complexité

Algorithmes et structures de données, 2025-2026

---

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

En partie inspiré des supports de cours de P. Albuquerque

## La récursivité

- Une condition de récursivité - qui *réduit* les cas successifs vers...
- Une condition d'arrêt - qui retourne un résultat

## Un exemple

```
int factorial(int n) {  
    if (n > 1) {  
        return n * factorial(n - 1);  
    } else {  
        return 1;  
    }  
}
```

L'exponentiation rapide ou  
indienne

# Exponentiation rapide ou indienne (1/3)

**But : Calculer  $x^n$**

- Quel est l'algorithme le plus simple que vous pouvez imaginer ?

# Exponentiation rapide ou indienne (1/3)

**But :** Calculer  $x^n$

- Quel est l'algorithme le plus simple que vous pouvez imaginer ?

```
double pow(double x, int n) {  
    if (0 == n) {  
        return 1.0;  
    }  
    double p = x;  
    for (int i = 1; i < n; ++i) {  
        p = p * x; // p *= x  
    }  
    return p;  
}
```

- Combien de multiplications et d'assignations en fonction de  $n$  ?

# Exponentiation rapide ou indienne (1/3)

**But :** Calculer  $x^n$

- Quel est l'algorithme le plus simple que vous pouvez imaginer ?

```
double pow(double x, int n) {  
    if (0 == n) {  
        return 1.0;  
    }  
    double p = x;  
    for (int i = 1; i < n; ++i) {  
        p = p * x; // p *= x  
    }  
    return p;  
}
```

- Combien de multiplications et d'assignations en fonction de  $n$  ?
- $n$  assignations et  $n$  multiplications.

## Exponentiation rapide ou indienne (2/3)

- Proposez un algorithme naïf et récursif

## Exponentiation rapide ou indienne (2/3)

- Proposez un algorithme naïf et récursif

```
double pow(double x, int n) {  
    if (n != 0) {  
        return x * pow(x, n-1);  
    } else {  
        return 1.0;  
    }  
}
```



# Exponentiation rapide ou indienne (3/3)

## Le vrai algorithme

- Si  $n$  est pair : calculer  $x^{n/2} \cdot x^{n/2}$  ;
- Si  $n$  est impair : calculer  $x \cdot x^{n-1}$ .

**Exercice : écrire l'algorithme récursif correspondant**

# Exponentiation rapide ou indienne (3/3)

## Le vrai algorithme

- Si  $n$  est pair : calculer  $x^{n/2} \cdot x^{n/2}$  ;
- Si  $n$  est impair : calculer  $x \cdot x^{n-1}$ .

## Exercice : écrire l'algorithme récursif correspondant

```
double pow(double x, int n) {  
    if (0 == n) {  
        return 1.0;  
    } else if (n % 2 == 0) {  
        return pow(x, n / 2) * pow(x, n/2);  
    } else {  
        return x * pow(x, (n-1));  
    }  
}
```

# La représentation des nombres

# Représentation des nombres (1/2)

- Le nombre 247.

**Nombres décimaux : les nombres en base 10**

$10^2$	$10^1$	$10^0$
2	4	7

$$247 = 2 \cdot 10^2 + 4 \cdot 10^1 + 7 \cdot 10^0.$$

## Représentation des nombres (2/2)

- Le nombre 11110111.

### Nombres binaires : les nombres en base 2

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
1	1	1	1	0	1	1	1

$$1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

## Représentation des nombres (2/2)

- Le nombre 11110111.

### Nombres binaires : les nombres en base 2

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
1	1	1	1	0	1	1	1

$$1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

$$= 247.$$

## Conversion de décimal à binaire (1/2)

Convertir 11 en binaire ?

# Conversion de décimal à binaire (1/2)

## Convertir 11 en binaire ?

- On décompose en puissances de 2 en partant de la plus grande possible

$$11 / 8 = 1, \quad 11 \% 8 = 3$$

$$3 / 4 = 0, \quad 3 \% 4 = 3$$

$$3 / 2 = 1, \quad 3 \% 2 = 1$$

$$1 / 1 = 1, \quad 1 \% 1 = 0$$

- On a donc

$$1011 \Rightarrow 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11.$$



## Conversion de décimal à binaire (2/2)

**Convertir un nombre arbitraire en binaire : 247 ?**

- Par groupe établir un algorithme.

# Conversion de décimal à binaire (2/2)

Convertir un nombre arbitraire en binaire : 247 ?

- Par groupe établir un algorithme.

## Algorithme

1. Initialisation

```
num = 247
N = 0
tant que ( $2^{(N+1)} < \text{num}$ ) {
    N += 1
}
```

# Conversion de décimal à binaire (2/2)

Convertir un nombre arbitraire en binaire : 247 ?

- Par groupe établir un algorithme.

## Algorithme

### 1. Initialisation

```
num = 247
N = 0
tant que ( $2^{(N+1)} < \text{num}$ ) {
    N += 1
}
```

### 2. Boucle

```
tant que ( $N \geq 0$ ) {
    bit = num /  $2^N$ 
    num = num %  $2^N$ 
    N -= 1
}
```

# Les additions en binaire

Que donne l'addition 1101 avec 0110 ?

- L'addition est la même que dans le système décimal

1101	8+4+0+1 = 13
+ 0110	+ 0+4+2+0 = 6
-----	-----
10011	16+0+0+2+1 = 19

- Les entiers sur un ordinateur ont une précision **fixée** (ici 4 bits).
- Que se passe-t-il donc ici ?

# Les additions en binaire

Que donne l'addition 1101 avec 0110 ?

- L'addition est la même que dans le système décimal

1101		$8+4+0+1 = 13$
+ 0110	+	$0+4+2+0 = 6$
-----		-----
10011		$16+0+0+2+1 = 19$

- Les entiers sur un ordinateur ont une précision **fixée** (ici 4 bits).
- Que se passe-t-il donc ici ?

**Dépassement de capacité : le nombre est “tronqué”**

- 10011 (19) -> 0011 (3).
- On fait “le tour”.

## Entier non-signés minimal/maximal

- Quel est l'entier non-signé maximal représentable avec 4 bits ?

## Entier non-signés minimal/maximal

- Quel est l'entier non-signé maximal représentable avec 4 bits ?

$$(1111)_2 = 8 + 4 + 2 + 1 = 15$$

- Quel est l'entier non-signé minimal représentable avec 4 bits ?

## Entier non-signés minimal/maximal

- Quel est l'entier non-signé maximal représentable avec 4 bits ?

$$(1111)_2 = 8 + 4 + 2 + 1 = 15$$

- Quel est l'entier non-signé minimal représentable avec 4 bits ?

$$(0000)_2 = 0 + 0 + 0 + 0 = 0$$

- Quel est l'entier non-signé min/max représentable avec N bits ?



## Entier non-signés minimal/maximal

- Quel est l'entier non-signé maximal représentable avec 4 bits ?

$$(1111)_2 = 8 + 4 + 2 + 1 = 15$$

- Quel est l'entier non-signé minimal représentable avec 4 bits ?

$$(0000)_2 = 0 + 0 + 0 + 0 = 0$$

- Quel est l'entier non-signé min/max représentable avec N bits ?

$$0 \text{ et } 2^N - 1.$$

- Donc `uint32_t`? maximal est ?

## Entier non-signés minimal/maximal

- Quel est l'entier non-signé maximal représentable avec 4 bits ?

$$(1111)_2 = 8 + 4 + 2 + 1 = 15$$

- Quel est l'entier non-signé minimal représentable avec 4 bits ?

$$(0000)_2 = 0 + 0 + 0 + 0 = 0$$

- Quel est l'entier non-signé min/max représentable avec N bits ?

$$0 \text{ et } 2^N - 1.$$

- Donc uint32\_t? maximal est ?

$$2^{32} - 1 = 4'294'967'295$$

# Les multiplications en binaire (1/2)

Que donne la multiplication de 1101 avec 0110 ?

- La multiplication est la même que dans le système décimal

1101	13
* 0110	* 6
-----	-----
0000	78
11010	
110100	
+ 0000000	
-----	-----
1001110	64+0+0+8+4+2+0

## Les multiplications en binaire (2/2)

Que fait la multiplication par 2 ?

## Les multiplications en binaire (2/2)

### Que fait la multiplication par 2 ?

- Décalage de 1 bit vers la gauche !

```
      0110
*     0010
-----
      0000
+     01100
-----
     01100
```

## Les multiplications en binaire (2/2)

**Que fait la multiplication par 2 ?**

- Décalage de 1 bit vers la gauche !

```
      0110
*     0010
-----
      0000
+     01100
-----
     01100
```

**Que fait la multiplication par  $2^N$  ?**

## Les multiplications en binaire (2/2)

### Que fait la multiplication par 2 ?

- Décalage de 1 bit vers la gauche !

$$\begin{array}{r} \phantom{0}0110 \\ * \phantom{0}0010 \\ \hline \phantom{0}0000 \\ + \phantom{0}01100 \\ \hline \phantom{0}01100 \end{array}$$

### Que fait la multiplication par $2^N$ ?

- Décalage de  $N$  bits vers la gauche !

## Entiers signés (1/2)

Pas de nombres négatifs encore...

**Comment faire ?**



# Entiers signés (1/2)

Pas de nombres négatifs encore...

**Comment faire ?**

**Solution naïve :**

- On ajoute un bit de signe (le bit de poids fort) :

00000010: +2

10000010: -2

**Problèmes ?**

# Entiers signés (1/2)

Pas de nombres négatifs encore...

**Comment faire ?**

**Solution naïve :**

- On ajoute un bit de signe (le bit de poids fort) :

00000010: +2

10000010: -2

**Problèmes ?**

- Il y a deux zéros (pas trop grave) : 10000000 et 00000000
- Les additions sont différentes que pour les non-signés (très grave)

00000010	2
+ 10000100	+ -4
-----	----
10000110 = -6	!= -2

# Entiers signés (2/2)

## Beaucoup mieux

- Complément à un :
  - on inverse tous les bits : 1001 => 0110.

## Encore un peu mieux

- Complément à deux :
  - on inverse tous les bits,
  - on ajoute 1 (on ignore les dépassements).

# Entiers signés (2/2)

## Beaucoup mieux

- Complément à un :
  - on inverse tous les bits : 1001 => 0110.

## Encore un peu mieux

- Complément à deux :
  - on inverse tous les bits,
  - on ajoute 1 (on ignore les dépassements).
- Comment écrit-on -4 en 8 bits ?

# Entiers signés (2/2)

## Beaucoup mieux

- Complément à un :
  - on inverse tous les bits : 1001 => 0110.

## Encore un peu mieux

- Complément à deux :
  - on inverse tous les bits,
  - on ajoute 1 (on ignore les dépassements).
- Comment écrit-on -4 en 8 bits ?

4 = 00000100

-4 => -----  
00000100

11111011  
+ 00000001

-----  
11111100

# Le complément à 2 (1/2)

## Questions :

- Comment on écrit  $+0$  et  $-0$  ?
- Comment calcule-t-on  $2 + (-4)$  ?
- Quel est le complément à 2 de 1000 0000 ?

# Le complément à 2 (1/2)

## Questions :

- Comment on écrit +0 et -0 ?
- Comment calcule-t-on  $2 + (-4)$  ?
- Quel est le complément à 2 de 1000 0000 ?

## Réponses

- Comment on écrit +0 et -0 ?

+0 = 00000000

-0 = 11111111 + 00000001 = 100000000  $\Rightarrow$  00000000

- Comment calcule-t-on  $2 + (-4)$  ?

00000010	2
+ 11111100	+ -4
-----	-----
11111110	-2

- En effet

11111110  $\Rightarrow$  00000001 + 00000001 = 00000010 = 2.

## Le complément à 2 (2/2)

Quels sont les entiers représentables en 8 bits ?



## Le complément à 2 (2/2)

**Quels sont les entiers représentables en 8 bits ?**

01111111 => 127

10000000 => -128 // par définition

**Quels sont les entiers représentables sur  $N$  bits ?**

## Le complément à 2 (2/2)

**Quels sont les entiers représentables en 8 bits ?**

01111111 => 127

10000000 => -128 // par définition

**Quels sont les entiers représentables sur  $N$  bits ?**

$$-2^{N-1} \dots 2^{N-1} - 1.$$

**Remarque : dépassement de capacité en C**

- Comportement indéfini !

# 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 résolus, calculer la solution.

## **Le pivot**

- Trouver le **pivot**, un élément qui divise le tableau en 2, tel que :
  1. Les éléments à gauche sont **plus petits** que le pivot.
  2. Les 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.

## 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 ?

## 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.

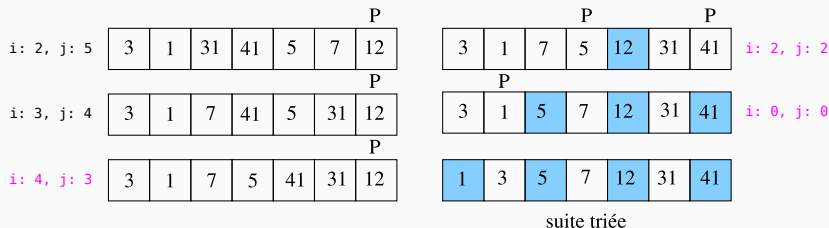
Compris ?

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)

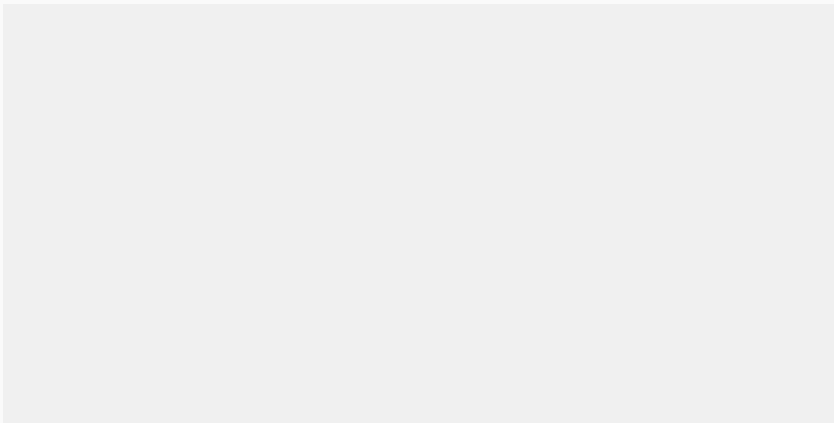
## Exercice : implémenter les fonctions quicksort et partition

```
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

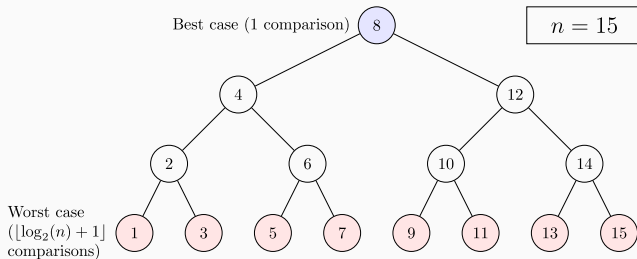


Figure 2 – Source : [Wikipédia](#)

# Analyse de complexité algorithmique (4/4)

## Recherche dichotomique

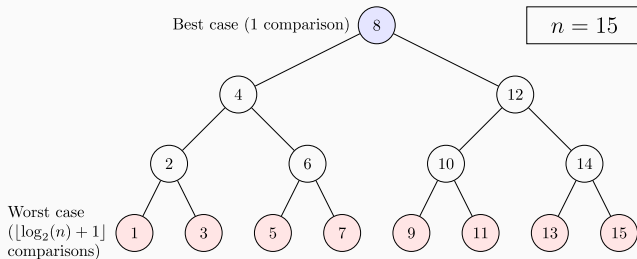


Figure 2 – Source : [Wikipédia](#)

- 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

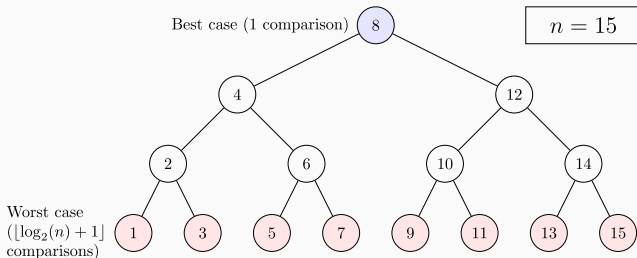


Figure 2 – Source : [Wikipédia](#)

- 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

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

**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