

# Module algorithmie et programmation

Réversivité et complexité

---

Pierre Künzli

Adapté des cours de Paul Albuquerque, Guido Bologna et Orestis Malaspinas

# Les types énumérés

---

# Types énumérés

- Un **type énuméré** : ensemble de *variantes* (valeurs constantes).
- En C les variantes sont des entiers numérotés à partir de 0.

```
typedef enum days {  
    monday, tuesday, wednesday,  
    thursday, friday, saturday, sunday  
}days_t;
```

- On peut aussi donner des valeurs “custom”

```
typedef enum days {  
    monday = 2, tuesday = 8, wednesday = -2,  
    thursday = 1, friday = 3, saturday = 12, sunday = 9  
}days_t;
```

- S'utilise comme un type standard et un entier

```
enum days d = monday;  
(d + 2) == tuesday + tuesday; // true
```

# Types énumérés

- Très utile dans les `switch ... case`

```
days_t d = monday;
switch (d) {
    case monday:
        // trucs
        break;
    case tuesday:
        printf("0 ou 1\n");
        break;
}
```

- Le compilateur vous prévient qu'il en manque !

# Utilisation des types énumérés

**Réusiner votre couverture de la reine avec des `enum`**

A faire à la maison comme exercice !

# La récursivité

---

# Exemple de récursivité

## La factorielle

- Code impératif

```
int factorial(int n) {  
    int f = 1;  
    for (int i = 1; i < n; ++i) {  
        f *= i;  
    }  
    return f;  
}
```

- Code récursif

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

# Exemple de récursivité

- Code récursif

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

Que se passe-t-il quand on fait `factorial(4)` ?



# Exemple de récursivité

- Code récursif

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

Que se passe-t-il quand on fait `factorial(4)` ?

On empile les appels

---

			factorial(1)
		factorial(2)	factorial(2)
	factorial(3)	factorial(3)	factorial(3)
factorial(4)	factorial(4)	factorial(4)	factorial(4)

---

# Exemple de récursivité

## La factorielle

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

Que se passe-t-il quand on fait `factorial(4)` ?

On dépile les calculs

---

1

`factorial(2)`     $2 * 1 = 2$

`factorial(3)`    `factorial(3)`     $3 * 2 = 6$

`factorial(4)`    `factorial(4)`    `factorial(4)`     $4 * 6 = 24$

---

# La récursivité

## Formellement

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

## Pour la factorielle, qui est qui ?

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

# La récursivité

## Formellement

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

## Pour la factorielle, qui est qui ?

```
int factorial(int n) {  
    if (n > 1) { // Condition de récursivité  
        return n * factorial(n - 1);  
    } else { // Condition d'arrêt  
        return 1;  
    }  
}
```

## Exercice

**Ecrire un code récursif permettant de calculer la longueur d'une chaîne de caractères**

## Exercice

Ecrire un code récursif permettant de calculer la longueur d'une chaîne de caractères

```
int length(char* str, int i){  
    if(str[i] == '\\0') return i;  
    return length(str, i+1);  
}
```

ou

```
int length(char* str){  
    if(*str == '\\0') return 0;  
    return 1 + length(str+1);  
}
```

# La récursivité

**Exercice : que fait ce code récursif ?**

```
void recurse(int n) {  
    printf("%d ", n % 2);  
    if (n / 2 != 0) {  
        recurse(n / 2);  
    } else {  
        printf("\n");  
    }  
}  
recurse(13);
```

# La récursivité

**Exercice : que fait ce code récursif ?**

```
void recurse(int n) {  
    printf("%d ", n % 2);  
    if (n / 2 != 0) {  
        recurse(n / 2);  
    } else {  
        printf("\n");  
    }  
}  
  
recurse(13);
```

```
binaire(13): n = 13, n % 2 = 1, n / 2 = 6,  
    binaire(6): n = 6, n % 2 = 0, n / 2 = 3,  
        binaire(3): n = 3, n % 2 = 1, n / 2 = 1,  
            binaire(1): n = 1, n % 2 = 1, n / 2 = 0.
```

*// affiche: 1 1 0 1*



# La récursivité

**Exercice : que fait ce code récursif ?**

```
void recurse(int n) {  
    printf("%d ", n % 2);  
    if (n / 2 != 0) {  
        recurse(n / 2);  
    } else {  
        printf("\n");  
    }  
}  
  
recurse(13);
```

```
binaire(13): n = 13, n % 2 = 1, n / 2 = 6,  
    binaire(6): n = 6, n % 2 = 0, n / 2 = 3,  
        binaire(3): n = 3, n % 2 = 1, n / 2 = 1,  
            binaire(1): n = 1, n % 2 = 1, n / 2 = 0.
```

*// affiche: 1 1 0 1 -> convertit un nombre en binaire*

# La suite de Fibonacci

## Règle

$$\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2), \quad \text{Fib}(0) = 0, \quad \text{Fib}(1) = 1.$$

0, 1, 1, 2, 3, 5, 8, 13, ...

**Exercice : écrire la fonction `Fib` en récursif et impératif**

# La suite de Fibonacci

## Règle

$$\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2), \quad \text{Fib}(0) = 0, \quad \text{Fib}(1) = 1.$$

0, 1, 1, 2, 3, 5, 8, 13, ...

**Exercice : écrire la fonction Fib en récursif et impératif**

**En récursif (6 lignes)**

```
int fib(int n) {  
    if (n > 1) {  
        return fib(n - 1) + fib(n - 2);  
    }  
    return n;  
}
```

# La suite de Fibonacci

## Et en impératif (11 lignes)

```
int fib_imp(int n) {  
    int fib0 = 1;  
    int fib1 = 1;  
    int fib  = n == 0 ? 0 : fib1;  
    for (int i = 2; i < n; ++i) {  
        fib  = fib0 + fib1;  
        fib0 = fib1;  
        fib1 = fib;  
    }  
    return fib;  
}
```

# Efficacité algorithmique

---

# Efficacité algorithmique

Comment mesurer l'efficacité d'un algorithme ?

# Efficacité algorithmique

Comment mesurer l'efficacité d'un algorithme ?

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

# Efficacité algorithmique

Comment mesurer l'efficacité d'un algorithme ?

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

# Analyse de complexité algorithmique

- 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

- 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

## 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

## 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

## 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

## Recherche dichotomique

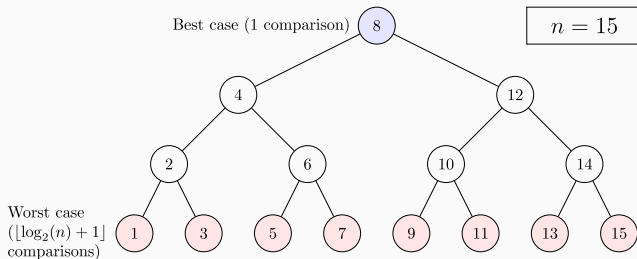


Fig. 1 : Source : [Wikipédia](#)



# Analyse de complexité algorithmique

## Recherche dichotomique

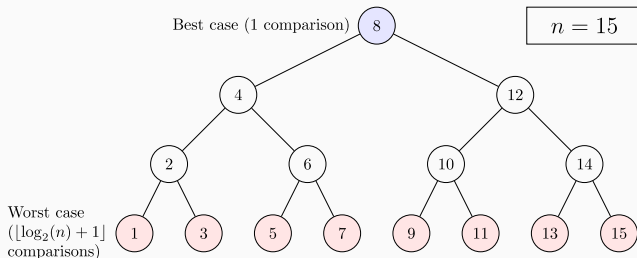


Fig. 1 : Source : [Wikipédia](#)

- 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

## Recherche dichotomique

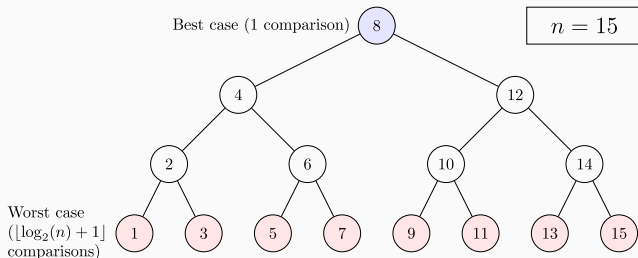


Fig. 1 : Source : [Wikipédia](#)

- 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 20 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

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

**Tab. 3 :** 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

## Complexité de trouver le minimum d'un tableau ?

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

# Quelques exercices

## Complexité de trouver le minimum d'un tableau ?

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

## Réponse

$\mathcal{O}(N)$ .



# Quelques exercices

## Complexité du tri par sélection ?

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

# Quelques exercices

## Complexité du tri par sélection ?

```
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

$$\mathcal{O}(N^2 \text{ comparaisons}) + \mathcal{O}(N \text{ swaps}) = \mathcal{O}(N^2).$$