

Types énumérés, tri par fusion, récursivité

Algorithmes et structures de données, 2025-2026

P. Albuquerque (B410) et O. Malaspinas (A401), ISC, HEPIA
2025-10-28

En partie inspiré des supports de cours de P. Albuquerque

Les types énumérés

Types énumérés (1/2)

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

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

- On peut aussi donner des valeurs “custom”

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

- S'utilise comme un type standard et un entier

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

Types énumérés (2/2)

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

```
enum days 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 !

Le tri par fusion (merge sort)

Tri par fusion (merge sort)

- Tri par comparaison.
- Idée : deux listes triées sont fusionnées pour donner une liste triée plus longue.
- Itérativement, on trie d'abord les paires de nombres, puis les groupes de 4 nombres, ensuite de 8, et ainsi de suite jusqu'à obtenir un tableau trié.

Principe de l'algorithme

- Soit `taille` la taille du tableau à trier.
- Pour $i = 0$ à $\text{entier}(\log_2(\text{taille}))-1$:
 - Fusion des paires de sous-tableaux successifs de taille 2^i (ou moins pour l'extrémité)

Principe de l'algorithme

- Soit `taille` la taille du tableau à trier.
- Pour $i = 0$ à $\text{entier}(\log_2(\text{taille}))-1$:
 - Fusion des paires de sous-tableaux successifs de taille 2^i (ou moins pour l'extrémité)
- Remarques :
 - Pour l'étape i , les sous-tableaux de taille 2^i sont triés.
 - La dernière paire de sous-tableaux peut être incomplète (vide ou avec moins que 2^i éléments).

Exemple de tri par fusion

- Soit la liste de nombres entiers stockés dans un tableau de taille 9 :

0	1	2	3	4	5	6	7	8
5	-5	1	6	4	-6	2	-9	2

Exemple de tri par fusion

- Soit la liste de nombres entiers stockés dans un tableau de taille 9 :

0	1	2	3	4	5	6	7	8
5	-5	1	6	4	-6	2	-9	2

- Fusion des éléments successifs (ce qui revient à les mettre dans l'ordre) :

étape	0	1	2	3	4	5	6	7	8
0	5	-5	1	6	4	-6	2	-9	2
1	-5	5	1	6	-6	4	-9	2	2
2	-5	1	5	6	-9	-6	2	4	2
3	-9	-6	-5	1	2	4	5	6	2
4	-9	-6	-5	1	2	2	4	5	6

Pseudo-code (autrement)

```
rien tri_fusion(entier taille, entier tab[taille])
    entier tab_tmp[taille];
    entier nb_etapes = log_2(taille) + 1;
    pour etape de 0 a nb_etapes - 1:
        entier gauche = 0;
        entier t_tranche = 2**etape;
        tant que (gauche < taille):
            fusion(
                tab[gauche..gauche+t_tranche-1],
                tab[gauche+t_tranche..gauche+2*t_tranche-1],
                tab_tmp[gauche..gauche+2*t_tranche-1]);
            #bornes incluses
            gauche += 2*t_tranche;
    echanger(tab, tab_tmp);
```

Algorithme de fusion possible

Une idée ?

Algorithme de fusion possible

Une idée ?

- Parcourir les deux tableaux jusqu'à atteindre la fin d'un des deux
 - Comparer l'élément courant des 2 tableaux
 - Écrire le plus petit élément dans le tableau résultat
 - Avancer de 1 dans les tableaux du plus petit élément et résultat
- Copier les éléments du tableau restant dans le tableau résultat

La fonction de fusion (pseudo-code autrement)

Une idée ?

La fonction de fusion (pseudo-code autrement)

Une idée ?

hyp: tab_g et tab_d sont triés

```
rien fusion(entier tab_g[], entier tab_d[], entier res[]):  
    entier g = taille(tab_g)  
    entier d = taille(tab_d)  
    entier i_g = 0, i_d = 0  
    pour i = 0 à g + d:  
        si i_g < g et i_d < d:  
            si tab_g[i_g] < tab_d[i_d]:  
                res[i] = tab_g[i_g]  
                i_g = i_g + 1  
            sinon:  
                res[i] = tab_d[i_d]  
                i_d = i_d + 1  
        sinon si i_g < g:  
            res[i] = tab_g[i_g]  
            i_g = i_g + 1  
        sinon si i_d < d:  
            res[i] = tab_d[i_d]  
            i_d = i_d + 1
```


La récursivité

La factorielle : Code impératif

- Code impératif

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

Exemple de récursivité (1/2)

La factorielle

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

Exemple de récursivité (1/2)

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)` ?

Exemple de récursivité (1/2)

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 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 (2/2)

La factorielle

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

Exemple de récursivité (2/2)

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)` ?

Exemple de récursivité (2/2)

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 (1/4)

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 (2/4)

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 : trouver l' ε -machine pour un double

Exercice : trouver l' ϵ -machine pour un double

Rappelez-vous vous l'avez fait en style **impératif** plus tôt.

La récursivité (3/4)

Exercice : trouver l' ϵ -machine pour un double

Rappelez-vous vous l'avez fait en style **impératif** plus tôt.

```
double epsilon_machine(double eps) {  
    if (1.0 + eps != 1.0) {  
        return epsilon_machine(eps / 2.0);  
    } else {  
        return eps;  
    }  
}
```

La récursivité (4/4)

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 (4/4)

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);
```

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

// affiche: 1 0 1 1

La récursivité (4/4)

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);
```

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

// affiche: 1 0 1 1

Affiche la représentation binaire d'un nombre !

Exercice : réusinage et récursivité (1/4)

Réusiner le code du PGCD avec une fonction récursive

Étudier l'exécution

$$42 = 27 * 1 + 15$$

$$27 = 15 * 1 + 12$$

$$15 = 12 * 1 + 3$$

$$12 = 3 * 4 + 0$$

Exercice : réusinage et récursivité (2/4)

Réusiner le code du PGCD avec une fonction récursive

Étudier l'exécution

$42 = 27 * 1 + 15$		PGCD(42, 27)
$27 = 15 * 1 + 12$		PGCD(27, 15)
$15 = 12 * 1 + 3$		PGCD(15, 12)
$12 = 3 * 4 + 0$		PGCD(12, 3)

Exercice : réusinage et récursivité (3/4)

Réusiner le code du PGCD avec une fonction récursive

Étudier l'exécution

$42 = 27 * 1 + 15$		PGCD(42, 27)
$27 = 15 * 1 + 12$		PGCD(27, 15)
$15 = 12 * 1 + 3$		PGCD(15, 12)
$12 = 3 * 4 + 0$		PGCD(12, 3)

Effectuer l'empilage - dépilage

Exercice : réusinage et récursivité (3/4)

Réusiner le code du PGCD avec une fonction récursive

Étudier l'exécution

$42 = 27 * 1 + 15$		PGCD(42, 27)
$27 = 15 * 1 + 12$		PGCD(27, 15)
$15 = 12 * 1 + 3$		PGCD(15, 12)
$12 = 3 * 4 + 0$		PGCD(12, 3)

Effectuer l'empilage - dépilage

PGCD(12, 3)		3
PGCD(15, 12)		3
PGCD(27, 15)		3
PGCD(42, 27)		3

Exercice : réusinage et récursivité (4/4)

Écrire le code

Exercice : réusinage et récursivité (4/4)

Écrire le code

```
int pgcd(int n, int m) {  
    if (n % m > 0) {  
        return pgcd(m, n % m);  
    } else {  
        return m;  
    }  
}
```

La suite de Fibonacci (1/2)

Règle

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

Exercice : écrire la fonction Fib en récursif et impératif

La suite de Fibonacci (1/2)

Règle

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

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 (2/2)

Et en impératif (11 lignes)

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