Récursivité et complexité

Algorithmique et structures de données, 2022-2023

P. Albuquerque (B410), P. Künzli et O. Malaspinas (A401), ISC, HEPIA

2022-11-09

En partie inspirés des supports de cours de P. Albuquerque

La récursivité (1/2)

- Code récursif

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

La récursivité (1/2)

- Code récursif

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

- Code impératif

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

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

```c
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

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

Exercice: écrire la fonction Fib en récursif et impératif

La suite de Fibonacci (1/2)

Règle

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

Exercice: écrire la fonction Fib en récursif et impératif

En récursif (6 lignes)

```c
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 = 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;
}
```

Exponentiation rapide ou indienne (1/4)

But: Calculer x^n

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

Exponentiation rapide ou indienne (1/4)

But: Calculer x^n

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

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

- Combien de multiplication et d'assignations en fonction de `n`?

Exponentiation rapide ou indienne (1/4)

But: Calculer x^n

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

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

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

Exponentiation rapide ou indienne (2/4)

- Proposez un algorithme naïf et récursif

Exponentiation rapide ou indienne (2/4)

- Proposez un algorithme naïf et récursif

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

Exponentiation rapide ou indienne (3/4)

Exponentiation rapide ou indienne de x^n

- Écrivons $n = \sum_{i=0}^{d-1} b_i 2^i,\ b_i = \{0, 1\}$ (écriture binaire sur d bits, avec $d \sim \log_2(n)$).
- $$x^n = x^{2^0 b_0} \cdot x^{2^1 b_1} \cdots x^{2^{d-1} b_{d-1}}.$$
- On a besoin de d calculs pour les x^{2^i}.
- On a besoin de d calculs pour évaluer les produits de tous les termes.

Combien de calculs en terme de n?

Exponentiation rapide ou indienne (3/4)

Exponentiation rapide ou indienne de x^n

- Écrivons $n = \sum_{i=0}^{d-1} b_i 2^i,\ b_i = \{0, 1\}$ (écriture binaire sur d bits, avec $d \sim \log_2(n)$).
- $$x^n = x^{2^0 b_0} \cdot x^{2^1 b_1} \cdots x^{2^{d-1} b_{d-1}}.$$
- On a besoin de d calculs pour les x^{2^i}.
- On a besoin de d calculs pour évaluer les produits de tous les termes.

Combien de calculs en terme de n?

- n est représenté en binaire avec d bits $\Rightarrow d \sim \log_2(n)$.
- il y a $2\log_2(n) \sim \log_2(n)$ calculs.

Exponentiation rapide ou indienne (4/4)

Le vrai algorithme

- Si n est pair: calculer $\left(x^{n/2}\right)^2$,
- Si n est impair: calculer $x \cdot \left(x^{(n-1)/2}\right)^2$.

Exercice: écrire l'algorithme récursif correspondant

Exponentiation rapide ou indienne (4/4)

Le vrai algorithme

- Si n est pair: calculer $\left(x^{n/2}\right)^2$,
- Si n est impair: calculer $x \cdot \left(x^{(n-1)/2}\right)^2$.

Exercice: écrire l'algorithme récursif correspondant

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

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

```c
#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 -O0
the computation took about 0.00836 seconds
RUN ONCE -O3
the computation took about 0.00203 seconds
RUN THOUSAND TIMES -O0
the computation took about 0.00363 seconds
RUN THOUSAND TIMES -O3
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 -O0
the computation took about 0.00836 seconds
RUN ONCE -O3
the computation took about 0.00203 seconds
RUN THOUSAND TIMES -O0
the computation took about 0.00363 seconds
RUN THOUSAND TIMES -O3
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

```c
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

```c
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

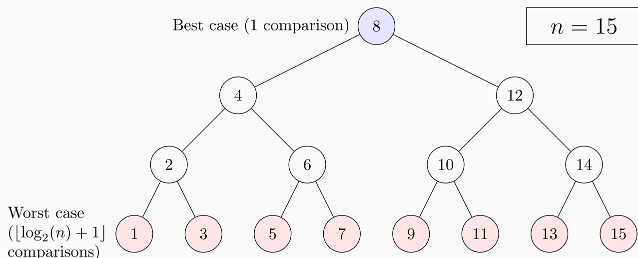

Figure 1: Source: Wikipédia

Analyse de complexité algorithmique (4/4)

Recherche dichotomique

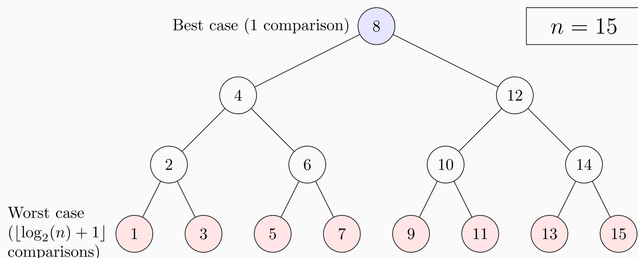

Figure 1: Source: Wikipédia

- 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 (4/4)

Recherche dichotomique

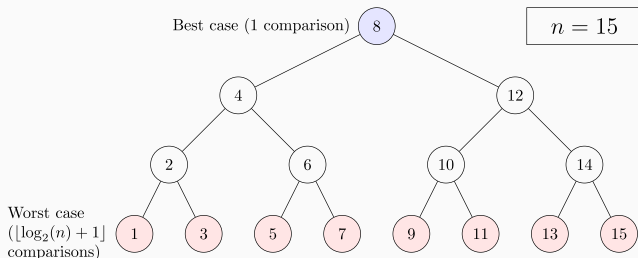

Figure 1: Source: Wikipédia

- 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 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`

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

Finalement

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