ROBERT MOGOS

ALGORITHMIQUE AVANCÉE

SYLLABUS

- Introduction
  - Itération et récursivité
  - Complexité en temps et en espace
- Tableaux
  - Opérations sur les tableaux
  - Tri : tri a bulles, tri rapide, tri fusion, tri par paquets
  - Recherche dichotomique

SYLLABUS

- Listes chainées
  - Implémentation
  - Opérations sur les listes (inverser, recherche ..)
- Piles et files
- Table de hachage

SYLLABUS

- Arbres
  - Arbres génériques
  - Arbres binaires
- Arbres binaires de recherche
- Opérations sur les arbres (recherche, parcours en profondeur et en largeur)

QU'EST-CE QUE L'ALGORITHMIQUE ?

Un algorithme est suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème.

- *Trouver une méthode* de résolution du problème
- Trouver une méthode *efficace*
  - *en temps*
  - *en espace*

ITÉRATION

- Un algorithme est dit itératif lorsque on répète un bloc d'instructions.

```
int my_pow_iterative(int n, int e) {
  if (0 == e) return 0;
  int number = n;
  for (int i = 1; i < e; i ++) {
    number = number * n;
  }
  return number;
}
```

RÉCURSIVITÉ

Un algorithme est dit récursif lorsqu'il est défini en fonction de lui-même.

```c
int my_pow_recursive(int n, int e) {
  if (0 == e) return 0;
  if (1 == e) return n;
  return n * my_pow_recursive(n, e - 1);
}
```

RÉCURSIVITÉ

- Méthode
  - Décomposer en sous-problèmes
  - Avoir des cas d'arrêt: un certain nombre de cas dont la résolution est connu
    - un appel récursif peut produire lui-même un autre appel récursif, etc, ce qui peut mener à une suite infinie d'appels (StackOverflow)

FACTORIELLE DE N (N!)

N! = 1 * 2 * 3 * .... * N

Exemple: 3! = 1 * 2 * 3 = 6

A vous de jouer ;)

N! – ITÉRATIF

```
int factorial_iterative(int n) {
  if (0 == n) return 0;
  int factorial = 1;
  for (int i = 1; i <= n; i ++) {
    factorial = factorial * i;
  }
  return factorial;
}
```

N! – RÉCURSIF

```c
int factorial_recursive(int n) {
  if (0 == n) return 0;
  if (1 == n) return 1;
  return n * factorial_recursive(n - 1);
}
```

SUITE DE FIBONACCI

Fb(0) = 0;

Fb(1) = 1;

Fb(n) = Fb(n - 1) + Fb(n - 2);

Exemple:

Fb(2) = Fb(1) + Fb(0) = 1

Fb(3) = Fb(2) + Fb(1) = 2

Fb(4) = Fb(3) + Fb(2) = 3

SUITE DE FIBONACCI – RÉCURSIF

```
int fibonacci_recursive(int n) {
  if (0 == n) return 0;
  if (1 == n) return 1;
  int fib = fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2);
  return fib;
}
```

SUITE DE FIBONACCI - ITÉRATIF

```
int fibonacci_iterative(int n) {
  if (0 == n) return 0;
  if (1 == n) return 1;
  int firstNumber = 0;
  int secondNumber = 1;
  int fib = 1;
  for (int i = 2; i <= n; i ++) {
    fib = firstNumber + secondNumber;
    firstNumber = secondNumber;
    secondNumber = fib;
  }
  return fib;
}
```

COMPLEXITÉ

La complexité d'un algorithme est la mesure du nombre d'opérations fondamentales qu'il effectue sur un jeu de données. La complexité est exprimée comme une fonction de la taille du jeu de données

- On calcule la complexité en terme de
- temps
  - le nombre d'étapes de calcul / le temps d'exécution d'un algorithme
- espace
  - évaluation de l'espace mémoire occupé par l'exécution de l'algorithme

COMPLEXITÉ

- Complexité au pire
  - temps d’exécution maximum, dans le cas le plus défavorable
- Complexité au mieux
  - temps d'exécution minimum, dans le cas le plus favorable
- Complexité moyenne
  - temps d'exécution dans un cas médian, ou moyenne des temps d’exécution

Le plus souvent, on utilise la complexité au pire, car on veut borner le temps d'exécution

COMPLEXITÉ

- Notation: O dite notation de Landau
- O(1) : complexité constante, pas d'augmentation du temps d'exécution quand le paramètre croit
- O(log(n)) : complexité logarithmique, augmentation très faible du temps d'exécution quand le paramètre croit. Exemple : algorithmes qui décomposent un problème en un ensemble de problèmes plus petits (dichotomie).
- O(n) : complexité linéaire, augmentation linéaire du temps d'exécution quand le paramètre croit (si le paramètre double, le temps double). Exemple : algorithmes qui parcourent séquentiellement des structures linéaires.
- O(nlog(n)) : complexité quasi-linéaire, augmentation un peu supérieure à O(n). Exemple : algorithmes qui décomposent un problème en d'autres plus simples, traités indépendamment et qui combinent les solutions partielles pour calculer la solution générale.
- $O(n^2)$: complexité quadratique, quand le paramètre double, le temps d'exécution est multiplié par 4. Exemple : algorithmes avec deux boucles imbriquées.
- $O(n^i)$: complexité polynomiale, quand le paramètre double, le temps d'exécution est multiplié par 2i. Exemple : algorithme utilisant i boucles imbriquées