Algorithmique et complexité

Polytech Paris-Saclay, PEIP 2, Informatique 3

Thibaut Benjamin

26 Novembre 2025

Amphi 4

Séance du jour : Sujets avancés de complexité

- Un dernier algorithme de tri
- Mémoisation
- Vecteurs C++ et complexité amortie
- Classes P et NP

Complexité en moyenne et tri rapide

Visualisation

https://mszula.github.io/visual-sorting/?algorithm=quick-sort

Principe du tri rapide

- Le tri rapide est un tri récursif en deux étapes
- **1. Partitionnement :** on commence par arranger les éléments du tableau à trier dans deux sous-tableaux tels que :
  Tous les éléments du sous-tableau de gauche sont plus petits que tous les éléments du sous-tableau de droite
- **2. Appels récursifs :** on procède au tri rapide sur chacun des deux sous-tableaux.

Principe du partitionnement

- On choisit un élément particulier dans le tableau de longueur n que l'on appelle le **pivot**, et on crée un tableau auxiliaire `aux` temporaire de longueur n.

- On parcourt le tableau, en maintenant deux indices `debut` et `fin` initialisés respectivement à 0 et $n-1$.
  - Lorsque l'on rencontre un élément plus petit que le pivot, on l'ajoute dans `aux[debut]` et on incrémente `debut`
  - Lorsque l'on rencontre un élément plus grand que le pivot, on l'ajoute dans `aux[fin]` et on décrémente `fin`.

Complexité du partitionnement

- 1 parcourt du tableau initial $\rightarrow \Theta(n)$
- 1 copie $\rightarrow \Theta(n)$
- Résultat global : $\Theta(n)$

Complexité du tri rapide

- Problème on ne connaît pas a priori la taille des appels récursifs.

- Notons $T(n)$ la complexité du tri rapide sur un tableau de taille n.
  - **Partitionnement :** $\Theta(n)$
  - **Appels réccursifs :** T(longueur tableau gauche) $+$ T(longueur tableau droit) ????

- La taille des appels récursifs dépend du choix du pivot. Plus les tableaux sont équilibrés, plus l'algorithme sera rapide.

Complexité du tri rapide, pire cas

- Pire des cas : lorsque les tableaux sont maximalement déséquilibrés.
  l'un des deux sous-tableaux est de taille 1 et l'autre de taille $n-1$
- La complexité est alors donnée par :
  - **Partitionnement :** $\Theta(n)$
  - **Appels réccursifs :** $T(1) + T(n-1)$
- Complexité dans le pire des cas $\Theta(n^2)$

Complexité du tri rapide, cas moyen

- En moyenne, si on suppose que les tableaux sont bien mélangés, les deux sous-tableaux seront en général équilibrés.

- La complexité en moyenne est alors donnée par :
  - **Partitionnement :** $\Theta(n)$
  - **Appels réccursifs :** $T(\frac{n}{2}) + T(\frac{n}{2})$

- Complexité dans le pire des cas $\Theta(n \log(n))$

Le choix du pivot

- Si on connaît le cas typique d'utilisation de notre fonction, faire un choix de pivot astucieux
  - Si en moyenne les tableaux seront aléatoires, on peut choisir n'importe quel élément du tableau.
  - Si dans le cas typique, les tableaux sont déjà presque triés, on peut choisir le milieu du tableau
  - Si le cas typique on sait que certaines zones vont contenir des petites/grandes cases, on essaiera de les éviter
- Si on n'a aucune idée et que notre fonction pourrait être utilisée par d'autres personnes, on peut commencer par mélanger le tableau avant de le trier.

Mémoisation

Calcul de la suite de Fibonacci

- La suite de Fibonacci est une suite mathématique définie par récurrence de la manière suivante :

$$\begin{cases} F(0) = 0 \\ F(1) = 1 \\ F(n) = F(n-1) + F(n-2) \quad \text{pour } n \geq 2 \end{cases}$$

- Notre objectif ici est d'écrire un algorithme pour calculer la valeur de la suite de Fibonacci

Algorithme naïf

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

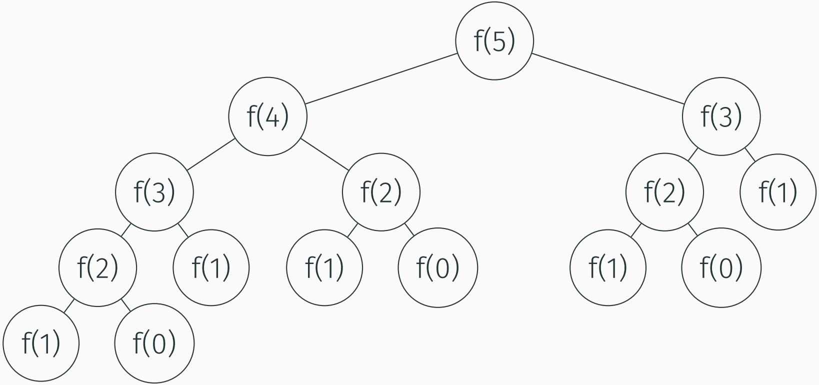
Complexité de l'algorithme naïf

- On note $T(n)$ la complexité de `fibonacci(n)`

- Relation de récurrence :
  - On a $T(0) = \Theta(1)$ et $T(1) = \Theta(1)$
  - Et on a $T(n) = T(n-1) + T(n-2) + \Theta(1)$

- La résolution de cette suite (formule de Binet) donne $T(n) = \Theta(2^n)$
  Complexité exponentielle

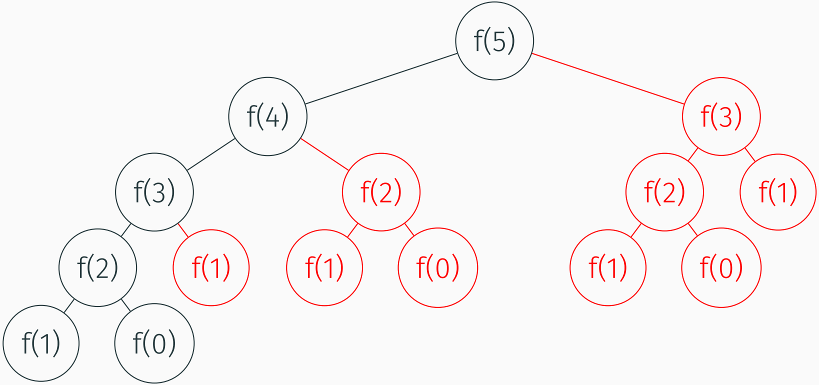
Liste des appels récursifs pour fibonacci(5)



Remarque

- Il y a beaucoup d'appels récursif redondants à la fonction fibonacci.
- Par exemple, on appelle `fibonacci(3)` à deux moments dans le calcul de `fibonacci(5)`. Ces deux appels renvoient le même résultat, pas la peine de le recalculer !

Liste des appels récursifs pour fibonacci(5), appels redondants en rouge



Mémoisation

- Principe de la mémoisation : stocker dans un tableau le résultat de l'appel à `fibonacci(n)`.
- La fonction fibonacci va maintenant d'abord vérifier si le nombre a déjà été calculé dans le tableau de mémoisation.
  - Si le résultat est déjà stocké dans le tableau, on le renvoie
  - Sinon, on le calcule, on l'ajoute au tableau, puis on le renvoie

Fibonacci avec mémoisation : fonction auxiliaire

```c
int fibonacci_memo_aux (int n, int memo[]){
  if (n == 0){return 0;}
  if (n == 1){return 1;}
  if (memo[n] == -1){
    int resultat = fibonacci (n-1) + fibonacci (n-2);
    memo[n] = resultat;
  }
  return memo[n];
}
```

Fibonacci avec mémoisation : fonction finale

```c
int fibonacci_memo(int n){
  int memo[n];
  for (int i = 0; i<n; i++){
    memo[i] = -1;
  }
  return fibonacci_memo_aux (n, memo);
}
```

Complexité de Fibonacci mémoisé - pire cas naïf

- La complexité de `fibonacci_memo_aux` dépend de si la valeur calculée est déjà dans le tableau.

- On pourrait dire : au pire des cas, la valeur n'est jamais dans le tableau.
  - Il faut donc supposer que l'on fait toujours le maximum d'appels récursifs

$$\begin{cases} T(0) = \Theta(1) \\ T(1) = \Theta(1) \\ T(n) = \mathcal{O}(T(n-1) + T(n-2)) \end{cases}$$

  - On trouve alors a nouveau $T(n) = \mathcal{O}(2^n)$.
    Complexité exponentielle. C'est correct, mais on peut être bien plus précis.

Complexité de Fibonacci mémoisé - analyse fine

- Notre erreur dans le raisonnement précédent : comme on ajoute le résultat au tableau, on ne peut pas être toujours dans le pire cas.

- Lors du calcul de `fibonacci(n-1)+fibonacci(n-2)`, on aura forcément déjà calculé `fibonacci(n-2)` dans l'appel correspondant à `fibonacci(n-1)`.
  - On peut donc simplifier la suite en

$$\begin{cases} T(0) = \Theta(1) \\ T(1) = \Theta(1) \\ T(n) = \mathcal{O}(T(n-1)) + \mathcal{O}(1) \end{cases}$$

  - Complexité globale : $\mathcal{O}(n)$
    Cet algorithme est linéaire

Mémoisation à grande échelle

- Dans notre exemple, on a mémoisé le résultat de la fonction de fibonacci de manière locale.
- Il peut être intéressant de plutôt mémoiser le résultat d'un calcul globalement, de cette manière à pouvoir réutiliser les calcul à l'échelle de notre programme.
- Avec `fibonacci`, cela permettrait à ce qu'un appel à `fibonacci(6)` à un moment dans le programme soit rendu plus rapide par le fait qu'on a déjà calculé `fibonacci(5)` à un autre moment.

- Attention : La mémoisation est une technique utile pour stocker le résultat de fonctions « pures ».

- Par exemple : Ce serait une très mauvaise idée de mémoiser une fonction qui renvoie la somme des éléments d'un tableau.

Mémoisation et effet de bords

- Attention : La mémoisation est une technique utile pour stocker le résultat de fonctions « pures ».
- Par exemple : Ce serait une très mauvaise idée de mémoiser une fonction qui renvoie la somme des éléments d'un tableau.
- En effet, le tableau peut être modifié, et la somme mémoisée ne retournera pas la bonne valeur.

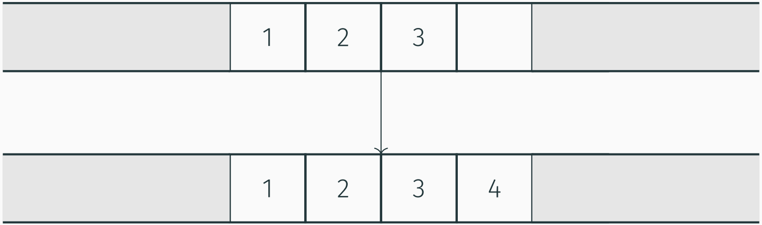
Vecteurs et complexité amortie

Vecteurs et tableaux

- Dans ce cours nous avons utilisé les tableaux de C++, et non les vecteurs.
- Les vecteurs sont des tableaux dynamiques, c'est à dire dont la taille peut changer.
- La complexité des opérations sur les vecteurs est plus subtile à évaluer.

Ajouter un élément à un vecteur, cas rapide

- Si il reste de la place dans la zone mémoire allouée pour le vecteur, on ajoute l'élément voulue à cette place :



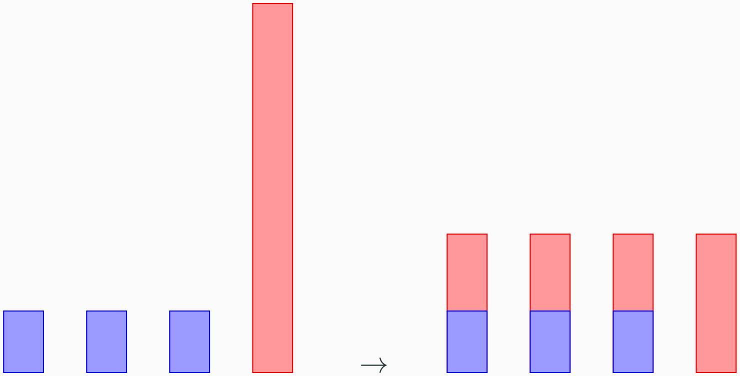
- Complexité : $\Theta(1)$
  Temps constant

Ajouter un élément à un vecteur, cas lent

- Si il n'y a plus de place dans la zone mémoire allouée pour le vecteur, on commence par recopier tout le vecteur à dans une zone mémoire allouée avec plus d'espace, et on ajoute l'élément ensuite.

- Complexité : $\Theta(n)$
  Linéaire

Le principe de la complexité amortie



Principe théorique

- Si entre deux fois où l'opération coûte cher, on s'assure d'avoir toujours assez d'opérations pas cher pour répartir les coûts de manière homogène, on peut parler de complexité amortie.
- On peut imaginer que chaque opération peu cher achète une part de l'opération chère, et que l'on peut faire l'opération chère que lorsqu'on a assez de part. $\rightarrow$ Mathématiquement difficile à modéliser et à analyser.
- En pratique, l'ajout d'un élément dans un vecteur est en temps constant amorti

Tableaux et vecteurs, une dernière comparaison

- Les tableaux sont moins pratiques à utiliser, mais ils sont plus simples à analyser.
- Les vecteurs permettent bien plus d'opérations, mais ils font beaucoup de choses de manière cachées. Les opérations sont en moyenne peu chère, mais peuvent de temps en temps être lentes.

Les classes P et NP

La classe P

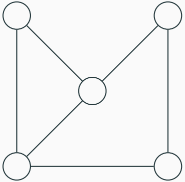
- Une classe de complexité importante en informatique est la classe P contenant tous problèmes de décision dont la complexité est $\mathcal{O}$ d'un polynôme.
- Elle contient par exemple tous les algorithmes de tri qu'on a vu, et la grande majorité des exemples de ce cours.

La classe NP

- La classe de complexité NP contient les problèmes de décisions, pour lesquels on sait vérifier qu'une solution est valide en un temps polynômial.
- Pour autant, on ne connaît pas nécessairement d'algorithme en temps polynômial pour résoudre le problème

Un exemple : 3-coloriabilité d’un graphe

- Etant donné un graphe, existe-t-il un moyen de colorier ses sommets avec 4 couleurs, de manière à ce que deux sommets adjacents n’aient pas la même couleur ?



- Si on me donne un coloriage, je peux vérifier qu’il est effectivement valide, en temps polynomial. Mais tester tous les coloriages possibles jusqu’à en trouver un est en temps exponentiel.

NP = Non-deterministically polynomial

- Imaginons que je cherche à construire un coloriage, et à chaque nouveau noeud, je choisis une couleur.

- Si j'ai beaucoup de chance, et que je fais tous les bons choix du premier coup, alors j'ai construit une solution au problème.

- Comme je sais vérifier en temps polynomial, ma solution est en temps polynomial.
  NP = en temps polynomial avec beaucoup de chance

Une question à 1 million de dollars

- A ce jour, on ne sait pas si les classes P et NP
- Si c'était le cas, cela signifierai que pour chacun des problèmes NP – que l'on peut résoudre en temps polynomial avec beaucoup de chance – il existe un algorithme en temps polynomial, sans avoir besoin de chance.
- C'est l'un des problème du millénaire, mis à pris à 1 million de dollars par le Clay Mathematical Institute.