Algorithmique et complexité

Polytech Paris-Saclay, PEIP 2, Informatique 3

Thibaut Benjamin

19 Novembre 2025

Amphi 3

Séance du jour

- Rappel sur les calculs de complexité
- Récursivité
- Complexité des fonctions récursives

Rappel sur les calculs de complexité

Premier exemple

```
// Diviser par 2 les nombres pairs et renvoyer le minimum
int diviserParDeux (int n, int tab[]){
  int min = tab[0];
  for (int i = 0; i < n; i++){
    if (tab[i]%2 == 0){
      tab[i] = tab[i]/2;
    }
    if (min > tab[i]){
      min = tab[i];
    }
  }
  return min;
}
```

Analyse de complexité

- Les opérations d'initialisations sont négligeables par rapport à celles de la boucle.

Analyse de complexité

- Les opérations d'initialisations sont négligeables par rapport à celles de la boucle.

- A chaque passage dans la boucle, on fait entre 2 et 4 opérations $\rightarrow \Theta(1)$.

Analyse de complexité

- Les opérations d'initialisations sont négligeables par rapport à celles de la boucle.
- A chaque passage dans la boucle, on fait entre 2 et 4 opérations $\rightarrow \Theta(1)$.
- On fait n passages dans la boucle $\rightarrow \Theta(n)$

Analyse de complexité

- Les opérations d'initialisations sont négligeables par rapport à celles de la boucle.
- A chaque passage dans la boucle, on fait entre 2 et 4 opérations $\rightarrow \Theta(1)$.
- On fait n passages dans la boucle $\rightarrow \Theta(n)$
- Total : $\Theta(1) \times \Theta(n) = \Theta(n)$
  Attention, c'est un abus de notation !

Deuxième exemple

```
// Somme de tous les elements d'une matrice
int sommeMatrice (int n, int m, int** mat){
  int s = 0;
  for (int i = 0; i < n; i++){
    for (int j = 0; j < m; j++){
      s = s + mat[i][j];
    }
  }
  return s;
}
```

Analyse de complexité

- Les opérations d'initialisations sont négligeables par rapport à celles des boucles.

Analyse de complexité

- Les opérations d'initialisations sont négligeables par rapport à celles des boucles.

- A chaque passage dans la boucle interne, on fait 1 opérations $\rightarrow \Theta(1)$.

Analyse de complexité

- Les opérations d'initialisations sont négligeables par rapport à celles des boucles.
- A chaque passage dans la boucle interne, on fait 1 opérations $\rightarrow \Theta(1)$.
- On fait m passages dans la boucle $\rightarrow \Theta(m)$, ce qui fait $\Theta(1) \times \Theta(m) = \Theta(m)$ à chaque passage dans la boucle externe.

Analyse de complexité

- Les opérations d'initialisations sont négligeables par rapport à celles des boucles.
- A chaque passage dans la boucle interne, on fait 1 opérations $\rightarrow \Theta(1)$.
- On fait m passages dans la boucle $\rightarrow \Theta(m)$, ce qui fait $\Theta(1) \times \Theta(m) = \Theta(m)$ à chaque passage dans la boucle externe.
- On fait n passages dans la boucle externe $\rightarrow \Theta(n)$, ce qui fait un total de $\Theta(m) \times \Theta(n) = \Theta(m \times n)$.
  Attention, c'est un abus de notation !

La récursivité

Situation typique

La récursivité est utile dans la situation suivante :

- On cherche à obtenir résoudre un problème sur une entrée de taille n
- A partir de la réponse à ce même problème sur une entrée plus petite taille, on cherche sait résoudre le problème de taille n

Un exemple

- On cherche à calculer la valeur de la factorielle $n! = 1 \times 2 \times \ldots \times n$.

- A on sait calculer cette valeur en fonction d'une factorielle plus petite :

$$n! = (n-1)! \times n$$

Stratégie des fonctions récursives

On applique la formule, et pour calculer le résultat sur le plus petit sous-problème, on appelle la fonction que l'on définit.

Exemple : la fonction factorielle

$$n! = (n-1)! \times n$$

```
// Fonction factorielle
int factorielle (int n){
  return factorielle (n-1) * n;
}
```

Exemple : la fonction factorielle

$$n! = (n-1)! \times n$$

```
// Fonction factorielle
int factorielle (int n){
  return factorielle (n-1) * n;
}
```

Cette fonction n'est pas tout à fait correcte

Pile d'appels récursifs

```
factorielle(3)
```

Pile d'appels récursifs

```
factorielle(3)
    ↳ factorielle(2)*3
```

Pile d'appels récursifs

```
factorielle(3)
    ↳ factorielle(2)*3
        ↳ factorielle(1) * 2
```

Pile d'appels récursifs

```
factorielle(3)
    ↳ factorielle(2)*3
        ↳ factorielle(1) * 2
            ↳ factorielle(0) * 1
```

Pile d'appels récursifs

```
factorielle(3)
    ↳ factorielle(2)*3
        ↳ factorielle(1) * 2
            ↳ factorielle(0) * 1
                ↳ factorielle(-1) * 0
```

Pile d'appels récursifs

```
factorielle(3)
    ↳ factorielle(2)*3
        ↳ factorielle(1) * 2
            ↳ factorielle(0) * 1
                ↳ factorielle(-1) * 0
                    ↳ factorielle(-2) * -1
```

Pile d'appels récursifs

```
factorielle(3)
    ↳ factorielle(2)*3
        ↳ factorielle(1) * 2
            ↳ factorielle(0) * 1
                ↳ factorielle(-1) * 0
                    ↳ factorielle(-2) * -1
                        ⋮
```

Pile d'appels récursifs

```
factorielle(3)
    ↳ factorielle(2)*3
        ↳ factorielle(1) * 2
            ↳ factorielle(0) * 1
                ↳ factorielle(-1) * 0
                    ↳ factorielle(-2) * -1
                        ⋮
```

La fonction ne s'arrête jamais

(Et en plus dans cet exemple le calcul devient faux à l'appel de `factorielle(0)`).

Stratégie (corrigée) des fonctions récursives

La fonction s'appelle elle-même pour donner les résultat sur des plus petits sous-problèmes, jusqu'à tomber sur un cas simple appelé cas de base, pour lequel on donne le résultat directement.

Exemple : la fonction factorielle

$$n! = (n-1)! \times n$$

```
// Fonction factorielle
int factorielle (int n){
  if (n == 0){
    return 1;
  }
  return factorielle (n-1) * n;
}
// appel a factorielle(3)
```

Exemple : la fonction factorielle

$$n! = (n-1)! \times n$$

```
// Fonction factorielle
int factorielle (int n){
  if (n == 0){
    return 1;
  }
  return factorielle (n-1) * n;
}
// appel a factorielle(3)
```

```
6
```

Pile d'appels récursifs

```
factorielle(3)
```

Pile d'appels récursifs

```
factorielle(3)
    ↳ factorielle(2)*3
```

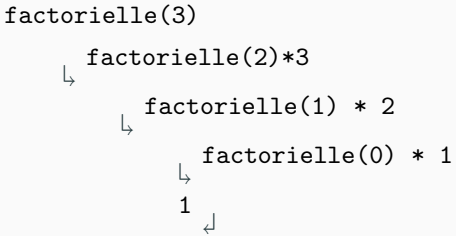
Pile d'appels récursifs

```
factorielle(3)
    ↳ factorielle(2)*3
        ↳ factorielle(1) * 2
```

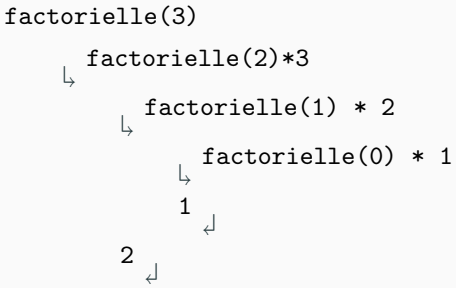
Pile d'appels récursifs

```
factorielle(3)
    ↳ factorielle(2)*3
        ↳ factorielle(1) * 2
            ↳ factorielle(0) * 1
```

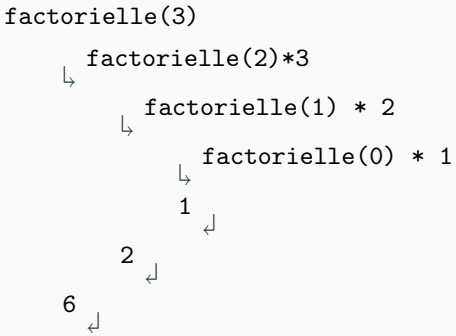
Pile d'appels récursifs



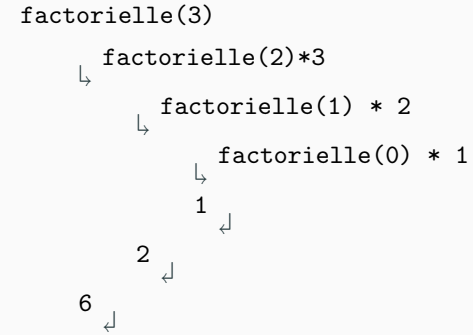
Pile d'appels récursifs



Pile d'appels récursifs



Pile d'appels récursifs



On a bien calculé $3! = 6$

Analyse de complexité

- On note $T(n)$ la complexité de la fonction `factorielle(n)`
- On a $T(0) = 1$
- On a une relation de récurrence $T(n) = T(n) + 2$
- En résolvant la récurrence (formule pour les suites arithmétiques) :
  $T(n) = 2 \times n + 1$
  Complexité $\Theta(n)$

Le tri par insertion revisité avec la récursion

Rappel

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

Stratégie récursive du tri par insertion

- On veut trier les n premières cases du tableau

- On commence par trier les $n-1$ premières cases du tableau

- On obtient le résultat en insérant la n^{e} case du tableau au bon endroit et en décalant
  on suppose que l'on a déjà la fonction `inserer(int n, int tab[])` que vous avez écrit lors du TP 1.

Tri par insertion : version récursive

```
// Tri par insertion, version recursive
void triInsertion (int n, int tab[]){
  if (n == 0){
    return;
  }
  triInsertion(n-1, tab);
  inserer(n, tab);
}
```

Analyse de complexité

- On rappelle que la fonction `inserer(int n, int tab[])` est en $\mathcal{O}(n)$.

- On dénote $T(n)$ le nombre d'opérations fait par le tri par insertion.

- Cela définit une suite récurrente

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

- Résoudre cette suite donne $T(n) = \mathcal{O}(1 + 2 + \ldots + (n+1)) = \mathcal{O}(\frac{(n+1)(n+2)}{2})$
  $\rightarrow \mathcal{O}(n^2)$

Contributions à la complexité

- Dans le tri par insertion, on déduit un tableau trié de taille n en faisant un appel récursif sur un tableau de taille $n-1$.
- Á chaque appel récursif, la taille du tableau en entrée décroît de 1, ce n'est pas très rapide. $\rightarrow \mathcal{O}(n)$ appels récursifs.
- Pour assembler les solutions des appels récursifs en une solution globale, on fait $\mathcal{O}(n)$ opérations
- D'où une complexité en $\mathcal{O}(n) \times \mathcal{O}(n) = \mathcal{O}(n^2)$.

Diviser pour régner : Le tri fusion

Rappel

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

Principe du tri fusion

- Idée clé : Pour trier un tableau, on sépare le tableau en 2 moitié, que l'on trie.
- On reconstitue le tableau trié en faisant une fusion des deux sous-tableaux triés.

La fusion

- On a un tableau t séparé en 2 sous-tableau trié, on veut les fusionner pour que le tableau global soit trié.

- On maintient deux positions i la première moitié et j dans la seconde moitié.
  - Si $t[i] < t[j]$, alors l'élément $t[i]$ est à sa bonne place, on incrémente donc i.
  - Sinon, il faut insérer $t[j]$ en position j et donc décaler tous éléments contenus dans les positions $i, \ldots j-1$. On incrémente ensuite i et j.

- On continue ainsi jusqu'à ce que l'un des deux tableaux ait été parcouru entièrement.

Complexité de la fusion

- Dans le pire des cas, toute la deuxième moitié du tableau contient des éléments plus grands que tout ceux de la première moitié du tableau.
- Dans ce cas, on fera à chaque étape 1 comparaison, et un décalage de $\frac{n}{2}$ éléments, soit $\mathcal{O}(n)$ opérations
- On fait en tout une étape par élément du deuxième tableau, soit $\mathcal{O}(n)$ étapes
- Soit une complexité totale de $\mathcal{O}(n^2)$

Algorithme de tri fusion

On part d'un tableau que l'on cherche à trié

- Cas de base : si le tableau est de longueur 1, il est déjà trié

- Sinon, on sépare le tableau en deux moitié et on fait les opérations suivantes :
  - On appelle récursivement le tri fusion sur chacune des deux moitiés
  - On fusionne le résultat

Analyse de complexité

Soit $T(n)$ la complexité du tri fusion sur un tableau de longueur n

- Cas de base : $T(1) = 1$

- Formule de récurrence :
  - On appelle récursivement le tri fusion sur chacune des deux moitiés $\rightarrow 2T(\frac{n}{2})$
  - On fusionne le résultat $\rightarrow \mathcal{O}(n^2)$

  $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n^2)$

- Complexité totale : $\mathcal{O}(n^2 \log(n))$

Contributions à la complexité

- Cette fois-ci, à chaque appel récursif, la taille des entrées est divisée par 2, ce qui fait $\mathcal{O}(\log(n))$ appels récursifs

- Mais pour assembler la solution à partir des appels récursifs, on paye un coût de $\mathcal{O}(n^2)$

- On a réussi à réduire la contribution liée au nombre d'appels récursifs, mais au prix d'un coût d'assemblage plus élevé $\rightarrow$ au global, cet algorithme est plus lent que le précédent

La fusion améliorée

- On peut améliorer la complexité de la fusion, en utilisant plus d'espace mémoire ! Etant donné un tableau t composé de deux sous-tableaux triés, on commence par créer un tableau t' de même taille. On maintient les positions i dans la première moitié et j dans la deuxième moitié du tableau.

- A chaque étape , on fait les opérations suivantes
  - si $t[i] < t[j]$ on copie $t[i]$ dans $t'[i+j]$ et on incrémente i
  - sinon on copie $t[j]$ dans $t'[i+j]$ et on incrémente j

- On répète jusqu'à avoir parcouru une des moitiés en entier, et alors on rempli t' avec ce qui reste de l'autre moitié, puis on copie t' dans t.

Complexité de la fusion améliorée

- Dans une première phase, on parcourt le tableau t en entier pour construire t'. A chaque étape on fait 3 opérations. $\rightarrow \Theta(n)$.

- Dans la deuxième phase, on copie le tableau t' dans le tableau, pour cela, il faut parcourir le tableau entier et faire une opération par étape. $\rightarrow \Theta(n)$.

- Résultat : $\Theta(n)$.

Complexité du tri fusion avec fusion améliorée

Soit $T(n)$ la complexité du tri fusion sur un tableau de longueur n

- Cas de base : $T(1) = 1$

- Formule de récurrence :
  - On appelle récursivement le tri fusion sur chacune des deux moitiés $\rightarrow 2T(\frac{n}{2})$
  - On fusionne le résultat $\rightarrow \mathcal{O}(n)$

  $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n)$

- Complexité totale : $\mathcal{O}(n \log(n))$

Contributions à la complexité

- Chaque appel récursif divise la taille des entrées par 2, ce qui fait $\mathcal{O}(\log n)$ appels récursifs.
- Pour reconstituer la solution au problème initial a partir du résultat des appels récursifs, on fait $\mathcal{O}(n)$ opérations
- D'où la complexité en $\mathcal{O}(n \log(n))$. C'est le mieux que l'on puisse faire pour un algorithme qui procède avec des comparaisons.

Conclusion

Résumé de la séance

- Complexité des boucles : Θ(nb d'itérations $\times$ complexité du corps)
- Fonction récursive = fonction qui s'appelle elle-même avec des plus petites entrées.
- La complexité d'une fonction récursive est obtenue en résolvant une récurrence.
- Contributions à la complexité d'une fonction récursive : nombre d'appels récursifs, et coût d'assemblage de la solution globale.