Complexité

1. Introduction

2. Opérations à coût constant
   - Présentation
   - Un mot sur la racine carrée

3. Exemples de complexités de fonctions récursives
   - Récurrences $C_n = C_{n-1} + 1$ et $C_n = C_{n-1} + n$
   - Stratégie « diviser pour régner »
   - Tri fusion

Crédits

1. « Option informatique MPSI, MP/MP* », Roger Mansuy, paru chez Vuibert.

Crédits

1. « Option informatique MPSI, MP/MP* », Roger Mansuy, paru chez Vuibert.
2. Cours « diviser pour régner », Becirspahic

Crédits

1. « Option informatique MPSI, MP/MP* », Roger Mansuy, paru chez Vuibert.
2. Cours « diviser pour régner », Becirspahic
3. Wikipédia : Analyse de la compléxité des algorithmes.

Définition

L'*analyse de la complexité* d'un algorithme consiste en l'étude formelle de la quantité de ressources (par exemple de temps ou d'espace) nécessaire à l'exécution de cet algorithme.

- **En temps** Il s'agit de savoir si une fonction termine dans un temps raisonnable.

Définition

L'*analyse de la complexité* d'un algorithme consiste en l'étude formelle de la quantité de ressources (par exemple de temps ou d'espace) nécessaire à l'exécution de cet algorithme.

- **En temps** Il s'agit de savoir si une fonction termine dans un temps raisonnable.
- **En mémoire** Il s'agit de déterminer si la fonction dispose bien de toutes les ressources physiques pour s'exécuter.

Temps et espace

Soit f un programme prenant en paramètre une donnée d

- Complexité en espace pour une donnée d : déterminer la place en mémoire *en plus* de d occupée par le programme.

Temps et espace

Soit f un programme prenant en paramètre une donnée d

- Complexité en espace pour une donnée d : déterminer la place en mémoire *en plus* de d occupée par le programme.
- Complexité en temps pour une donnée d : dénombrer toutes les opérations *à coût constant* effectuées lors de l'appel $f(d)$ et faire la somme de la durée de chacune.

Temps et espace

Soit f un programme prenant en paramètre une donnée d

- Complexité en espace pour une donnée d : déterminer la place en mémoire *en plus* de d occupée par le programme.
- Complexité en temps pour une donnée d : dénombrer toutes les opérations *à coût constant* effectuées lors de l'appel $f(d)$ et faire la somme de la durée de chacune.
- Parfois, on ne s'intéresse qu'à certaines opérations bien définies (ça dépend de ce qu'on veut étudier).

Définition

Définition

Notons D_n l'ensemble des données de taille n et $C(d)$ la complexité d'un certain programme pour une donnée d.

1. La complexité dans le pire cas est $C_{\max}(n) = \max_{d \in D_n} C(d)$

Définition

Définition

Notons D_n l'ensemble des données de taille n et $C(d)$ la complexité d'un certain programme pour une donnée d.

1. La complexité dans le pire cas est $C_{\max}(n) = \max_{d \in D_n} C(d)$
2. La complexité dans le meilleur cas est $C_{\min}(n) = \min_{d \in D_n} C(d)$

Définition

Définition

Notons D_n l'ensemble des données de taille n et $C(d)$ la complexité d'un certain programme pour une donnée d.

1. La complexité dans le pire cas est $C_{\max}(n) = \max_{d \in D_n} C(d)$
2. La complexité dans le meilleur cas est $C_{\min}(n) = \min_{d \in D_n} C(d)$
3. La complexité en moyenne est $C_{\text{moy}}(n) = \sum_{d \in D_n} P(d)C(d)$ où P désigne la loi de probabilité associée à l'apparition des données de taille n.

Landau

Définition

Soient $U = (u_n)$ et $V = (v_n)$ deux suites réelles positives. On dit que :

1. *U est dominée* par V si il existe $\lambda \in \mathbb{R}_+$ et $N \in \mathbb{N}$ tels que pour tout $n \in \mathbb{N}$, si $n \geq N$ alors $u_n \leq \lambda v_n$. On le note $u_n = O(v_n)$.
   On dit aussi que V *domine* U.

Landau

Définition

Soient $U = (u_n)$ et $V = (v_n)$ deux suites réelles positives. On dit que :

1. U *est dominée* par V si il existe $\lambda \in \mathbb{R}_+$ et $N \in \mathbb{N}$ tels que pour tout $n \in \mathbb{N}$, si $n \geq N$ alors $u_n \leq \lambda v_n$. On le note $u_n = O(v_n)$.
   On dit aussi que V *domine* U.
2. U et V sont *de même ordre (de grandeur)* si U domine V et V domine U (*i.e.* $u_n = O(v_n)$ et $v_n = O(u_n)$). On le note $u_n = \Theta(v_n)$ ($\ll u_n$ est en grand theta de $v_n \gg$).

Landau

Définition

Soient $U = (u_n)$ et $V = (v_n)$ deux suites réelles positives. On dit que :

1. On écrit $u_n = \Omega(v_n)$ ($\ll u_n$ est en grand Omega de $v_n \gg$) si il existe $\lambda \in \mathbb{R}_+$ et $N \in \mathbb{N}$ tels que pour tout $n \in \mathbb{N}$, si $n \geq N$ alors $u_n \geq \lambda v_n$. Observons que dans ce cas $\frac{1}{\lambda} u_n \geq v_n$ et donc $u_n = \Omega(v_n) \iff v_n = O(u_n)$.

Remarque

Si $u_n \sim v_n$ alors $u_n = \Theta(v_n)$.

Landau

Définition

Soient $U = (u_n)$ et $V = (v_n)$ deux suites réelles positives. On dit que :

1. On écrit $u_n = \Omega(v_n)$ ($\ll u_n$ est en grand Omega de $v_n \gg$) si il existe $\lambda \in \mathbb{R}_+$ et $N \in \mathbb{N}$ tels que pour tout $n \in \mathbb{N}$, si $n \geq N$ alors $u_n \geq \lambda v_n$.
   Observons que dans ce cas $\frac{1}{\lambda} u_n \geq v_n$ et donc
   $u_n = \Omega(v_n) \iff v_n = O(u_n)$.
2. U et V sont dites *équivalentes* si et seulement si $v_n \neq 0$ APCR et $\frac{u_n}{v_n} \underset{n \to +\infty}{\longrightarrow} 1$.
   Cette définition (qui nous suffit) est plus restrictive que celle du cours de maths.

Remarque

Si $u_n \sim v_n$ alors $u_n = \Theta(v_n)$.

Que compter ?

- La plupart du temps, on se contente de donner une majoration « à constante multiplicative près » du temps de calcul. On ne compte pas précisément le nombre d'opérations à coût constant
- Dans ce cas, la notation en $O(f(n))$ (si n est l'entrée) nous suffit.
- Parfois, on s'intéresse à une opération spécifique, et dans ce cas un décompte précis est favorisé. Exemple :
  - nombre d'échanges d'éléments dans un tri de tableau ;
  - nombre de produits de flottants dans un produit matriciel « optimisé ».

Ordres de grandeur (complexité temporelle au pire)

Pour une donnée de taille n :

Dénomination	Définition	Exemple
temps constant	$O(1)$	Accès au 1er d'un tableau
logarithmique	$O(\log(n))$	Recherche dans une liste triée
linéaire	$O(n)$	Inversion d'un tableau
quasi-linéaire	$O(n\log(n))$	Tri fusion d'un tableau
quadratique	$O(n^2)$	Tri par insertion d'un tableau
polynomiale	$O(n^k)$ pour un $k > 1$	produit matriciel naïf en $\Theta(n^3)$
exponentielle	$O(2^{P(n)})$ $P \in \mathbb{R}[X]$ $\deg(P) \geq 1$	Satisfiabilité naïve d'une formule

Opérations à coûts constants

- Pour mesurer le temps d'exécution d'un programme, on utilise le nombre d'*opérations à coût constant* à effectuer plutôt qu'une durée en seconde. En général on en cherche une majoration, plus rarement une minoration.

Opérations à coûts constants

- Pour mesurer le temps d'exécution d'un programme, on utilise le nombre d'*opérations à coût constant* à effectuer plutôt qu'une durée en seconde. En général on en cherche une majoration, plus rarement une minoration.
- La raison est que le même programme s'exécutera plus ou moins vite selon la machine mais que le nombre d'opérations ne changera pas.

Opérations à coûts constants

- Pour mesurer le temps d'exécution d'un programme, on utilise le nombre d'*opérations à coût constant* à effectuer plutôt qu'une durée en seconde. En général on en cherche une majoration, plus rarement une minoration.
- La raison est que le même programme s'exécutera plus ou moins vite selon la machine mais que le nombre d'opérations ne changera pas.
- Il reste à définir ce qu'est une *opération à coût constant*. Ce sont des opérations qui nécessitent un nombre borné d'actions du processeur (comme une addition de `int64_t`).
  Exemples :

```
1        + - / * % //opérations sur les nombres
2        t[i] # accès en lecture / écriture
3
```

Il y en a d'autres que nous dévoilerons au fur et à mesure.

Opérations à coûts constants en C

Pour le calcul de la complexité temporelle :

```
1        + - / * % // opérations arithmétiques
2        & << >> ^ | // opérations bitwise
3        a =...// écrire une valeur à l'adresse de a
4        return // revient à écrire à une adresse spéciale
5        t[i] # accès en lecture écriture
6        // comparaisons de nombres déjà calculés
7        ... < ...; ... == ... ;
8        free(t)// libération
9
```

Opérations à coûts constants en C

Pour le calcul de la complexité temporelle :

```
1        + - / * % // opérations arithmétiques
2        & << >> ^ | // opérations bitwise
3        a =...// écrire une valeur à l'adresse de a
4        return // revient à écrire à une adresse spéciale
5        t[i] # accès en lecture écriture
6        // comparaisons de nombres déjà calculés
7        ... < ...; ... == ... ;
8        free(t)// libération
9
```

- Dans `x < y`, la comparaison elle-même est en $O(1)$ mais l'évaluation de `x` et `y` peut être coûteuse.

Opérations à coûts constants

- Bien que la syntaxe soit différente, les opérations en OCaml équivalentes à celles ci-dessus (lorsqu'elles existent) sont aussi considérées comme à coût constants. Le *filtrage* est considéré comme de coût constant.

Opérations à coûts constants

- Bien que la syntaxe soit différente, les opérations en OCaml équivalentes à celles ci-dessus (lorsqu'elles existent) sont aussi considérées comme à coût constants. Le *filtrage* est considéré comme de coût constant.
- En Python, on a coutume de considérer l'addition comme à coût constant bien que ce soit faux : en effet les entiers étant non bornés, la somme de deux nombres s'effectue sur des parties de ces nombres, Python rajoutant un traitement logiciel pour recoller les morceaux.

Opérations à coûts constants

- Bien que la syntaxe soit différente, les opérations en OCaml équivalentes à celles ci-dessus (lorsqu'elles existent) sont aussi considérées comme à coût constants. Le *filtrage* est considéré comme de coût constant.
- En Python, on a coutume de considérer l'addition comme à coût constant bien que ce soit faux : en effet les entiers étant non bornés, la somme de deux nombres s'effectue sur des parties de ces nombres, Python rajoutant un traitement logiciel pour recoller les morceaux.
- En MP2I, les allocations sur le tas `malloc`, `calloc` et `realloc` sont considérées comme à coût constant même si c'est en fait bien difficile à déterminer.

Opérations à coûts constants

- Bien que la syntaxe soit différente, les opérations en OCaml équivalentes à celles ci-dessus (lorsqu'elles existent) sont aussi considérées comme à coût constants. Le *filtrage* est considéré comme de coût constant.
- En Python, on a coutume de considérer l'addition comme à coût constant bien que ce soit faux : en effet les entiers étant non bornés, la somme de deux nombres s'effectue sur des parties de ces nombres, Python rajoutant un traitement logiciel pour recoller les morceaux.
- En MP2I, les allocations sur le tas `malloc`, `calloc` et `realloc` sont considérées comme à coût constant même si c'est en fait bien difficile à déterminer.
- Les opérations d'affichage et de saisie `printf, scanf` peuvent prendre du temps. Il est préférable d'indiquer qu'elles ne sont pas comptabilisées dans le calcul de la complexité temporelle.

Opérations élémentaires

- Dans certains ouvrages, la notion de complexité correspond au décompte des *opérations élémentaires* effectuées par l'algorithme.

Opérations élémentaires

- Dans certains ouvrages, la notion de complexité correspond au décompte des *opérations élémentaires* effectuées par l'algorithme.
- Le plus souvent ces opérations élémentaires sont simplement les opérations de coût constant (addition, décalage, oépration bouléenne...).

Opérations élémentaires

- Dans certains ouvrages, la notion de complexité correspond au décompte des *opérations élémentaires* effectuées par l'algorithme.
- Le plus souvent ces opérations élémentaires sont simplement les opérations de coût constant (addition, décalage, oépration bouléenne...).
- Mais elles ne sont pas nécessairement de coût constant. Il peut s'agir par exemple de compter toutes les comparaisons de chaînes de caractères dans un algorithme de jointure. L'opération élémentaire est ici la comparaison de chaînes : elle est linéaire.

Exemple introductif

- On cherche les diviseurs d'un entier n :

```
1  void diviseurs1(int n):
2      for (int i=2; i<n; i++){
3          if (n % i == 0){
4          printf("%d; "i);// compté ici comme O(1)
5       }
6
```

Exemple introductif

- On cherche les diviseurs d'un entier n :

```
1  void diviseurs1(int n):
2      for (int i=2; i<n; i++){
3          if (n % i == 0){
4          printf("%d; " i);// compté ici comme O(1)
5       }
6
```

- Ce programme effectue exactement une initialisation, $n-2$ calculs de restes de divisions euclidiennes, $n-1$ comparaisons $i < n$, $n-2$ comparaisons $n\%i == 0$, $n-2$ incrémentations $i++$ et un nombre d'affichages inférieur ou égal à $n-2$.
  Au maximum, il y a donc $5(n-2)+2$ *opérations à coûts constants*.

Exemple introductif

- On cherche les diviseurs d'un entier n :

```
1  void diviseurs1(int n):
2      for (int i=2; i<n; i++){
3          if (n % i == 0){
4          printf("%d; "i);// compté ici comme O(1)
5       }
6
```

- Ce programme effectue exactement une initialisation, $n-2$ calculs de restes de divisions euclidiennes, $n-1$ comparaisons $i < n$, $n-2$ comparaisons $n\%i == 0$, $n-2$ incrémentations $i++$ et un nombre d'affichages inférieur ou égal à $n-2$.
  Au maximum, il y a donc $5(n-2)+2$ *opérations à coûts constants*.
- L'ordre de grandeur est donc donné par une application affine : On dit que « la complexité temporelle est en $O(n)$ »

Exemple introductif (2)

- Si $n = pq$ avec $p \geq \sqrt{n}$, alors $q \leq \sqrt{n}$.

Exemple introductif (2)

- Si $n = pq$ avec $p \geq \sqrt{n}$, alors $q \leq \sqrt{n}$.
- Nouveau principe : on cherche tous les diviseurs q qui sont plus petits que $\sqrt{n}$; les autres valent $\frac{n}{q}$.

Exemple introductif (2)

- Si $n = pq$ avec $p \geq \sqrt{n}$, alors $q \leq \sqrt{n}$.
- Nouveau principe : on cherche tous les diviseurs q qui sont plus petits que $\sqrt{n}$; les autres valent $\frac{n}{q}$.
- On entre :

```
1   void diviseurs2(int n):
2      int q; int p;// 2 op
3      q=1; //1  op
4      while (q*q <=n){//2 op
5       if (n % q == 0){ // 2 ops
6          printf("%d",q);// 1 op
7          p=n/q; // 2op
8          if (p != q) //(ne pas afficher q 2 fois si n=p^2)
       :1op
9                printf(p); //1 op
10         q=q+1;//2 op
11      }
12     }
```

Exemple introductif (2)

- $\sqrt{n}$ itérations au plus, moins de 20 opérations à coûts constants par passage.

Exemple introductif (2)

- $\sqrt{n}$ itérations au plus, moins de 20 opérations à coûts constants par passage.
- En tout pas plus de $3 + 20\sqrt{n}$ opérations à coûts constants. $O(\sqrt{n})$.

Taille du problème

- Déterminer le temps d'exécution en fonction de la *taille* du problème.

Taille du problème

- Déterminer le temps d'exécution en fonction de la *taille* du problème.
- Exemple : recherche de diviseur de n. La taille est n, le temps d'exécution est proportionnel à n ou $\sqrt{n}$.

Taille du problème

- Déterminer le temps d'exécution en fonction de la *taille* du problème.
- Exemple : recherche de diviseur de n. La taille est n, le temps d'exécution est proportionnel à n ou $\sqrt{n}$.
- Exemple : manipulation d'une liste. Taille du problème : nombre d'éléments.

Taille du problème

- Déterminer le temps d'exécution en fonction de la *taille* du problème.
- Exemple : recherche de diviseur de n. La taille est n, le temps d'exécution est proportionnel à n ou $\sqrt{n}$.
- Exemple : manipulation d'une liste. Taille du problème : nombre d'éléments.
- Exemple : traitement d'un fichier texte. Taille du problème : nombre de caractères.

Taille du problème

- Déterminer le temps d'exécution en fonction de la *taille* du problème.
- Exemple : recherche de diviseur de n. La taille est n, le temps d'exécution est proportionnel à n ou $\sqrt{n}$.
- Exemple : manipulation d'une liste. Taille du problème : nombre d'éléments.
- Exemple : traitement d'un fichier texte. Taille du problème : nombre de caractères.
- Addition de matrices $n \times m$. Taille du problème le couple (n, m) ou nm

Taille du problème

- Déterminer le temps d'exécution en fonction de la *taille* du problème.
- Exemple : recherche de diviseur de n. La taille est n, le temps d'exécution est proportionnel à n ou $\sqrt{n}$.
- Exemple : manipulation d'une liste. Taille du problème : nombre d'éléments.
- Exemple : traitement d'un fichier texte. Taille du problème : nombre de caractères.
- Addition de matrices $n \times m$. Taille du problème le couple (n, m) ou nm
- Recherche d'un mot dans un fichier. Taille du problème le couple (n, m) où n est le nombre de caractères du texte et m celui du mot.

Taille du problème

- Déterminer le temps d'exécution en fonction de la *taille* du problème.
- Exemple : recherche de diviseur de n. La taille est n, le temps d'exécution est proportionnel à n ou $\sqrt{n}$.
- Exemple : manipulation d'une liste. Taille du problème : nombre d'éléments.
- Exemple : traitement d'un fichier texte. Taille du problème : nombre de caractères.
- Addition de matrices $n \times m$. Taille du problème le couple (n, m) ou nm
- Recherche d'un mot dans un fichier. Taille du problème le couple (n, m) où n est le nombre de caractères du texte et m celui du mot.
- Compression d'images : une image n'est rien de plus qu'une matrice $w \times h$ de pixels. La taille du problème est (w, h).

Taille du problème

- Déterminer le temps d'exécution en fonction de la *taille* du problème.
- Exemple : recherche de diviseur de n. La taille est n, le temps d'exécution est proportionnel à n ou $\sqrt{n}$.
- Exemple : manipulation d'une liste. Taille du problème : nombre d'éléments.
- Exemple : traitement d'un fichier texte. Taille du problème : nombre de caractères.
- Addition de matrices $n \times m$. Taille du problème le couple (n, m) ou nm
- Recherche d'un mot dans un fichier. Taille du problème le couple (n, m) où n est le nombre de caractères du texte et m celui du mot.
- Compression d'images : une image n'est rien de plus qu'une matrice $w \times h$ de pixels. La taille du problème est (w, h).
- Souvent, la taille du problème est indiquée dans l'énoncé.

Modèle de complexité temporelle impérative

La complexité d'un algorithme est une fonction C définie inductivement.

- On ne mesure pas le temps d'exécution d'un programme en secondes mais *le nombre d'opérations à coûts constants*. Parfois aussi, on mesure le nombre de fois où une certaine opérations est effectuée même si elle n'est pas à coût constant.

Modèle de complexité temporelle impérative

La complexité d'un algorithme est une fonction C définie inductivement.

- On ne mesure pas le temps d'exécution d'un programme en secondes mais *le nombre d'opérations à coûts constants*. Parfois aussi, on mesure le nombre de fois où une certaine opérations est effectuée même si elle n'est pas à coût constant.
- Opérations « élémentaires ». On considère qu'elles ont toutes le même coût.

Modèle de complexité temporelle impérative

La complexité d'un algorithme est une fonction C définie inductivement.

- On ne mesure pas le temps d'exécution d'un programme en secondes mais *le nombre d'opérations à coûts constants*. Parfois aussi, on mesure le nombre de fois où une certaine opérations est effectuée même si elle n'est pas à coût constant.
- Opérations « élémentaires ». On considère qu'elles ont toutes le même coût.
- Séquences : Le coût des instructions `p; q` en séquence est la somme des coûts de l'instruction `p` et de l'instruction `q` : $C(p) + C(q)$

Modèle de complexité temporelle impérative

La complexité d'un algorithme est une fonction C définie inductivement.

- On ne mesure pas le temps d'exécution d'un programme en secondes mais *le nombre d'opérations à coûts constants*. Parfois aussi, on mesure le nombre de fois où une certaine opérations est effectuée même si elle n'est pas à coût constant.
- Opérations « élémentaires ». On considère qu'elles ont toutes le même coût.
- Séquences : Le coût des instructions `p; q` en séquence est la somme des coûts de l'instruction `p` et de l'instruction `q` : $C(p) + C(q)$
- Tests : Le coût d'une expression conditionnelle `if (b) {p;} else {q;}` est inférieur ou égal au maximum des coûts des instructions `p` et `q`, plus un test plus ou moins compliqué (temps d'évaluation de l'expression `b`). $\max(C(p), C(q)) + C(b)$. En général, on a $C(b) = O(1)$ et on le néglige. Mais ce n'est pas toujours vrai.

Modèle pour la complexité temporelle

- Le coût d'une boucle `for(i=0;i<m;i++) p;` est `m` fois le coût de l'instruction `p` si ce coût ne dépend pas de la valeur de `i` : $1 + m(C(p) + 2) + 1 = O(mC(p))$ ($\ll +2 \gg$ pour l'incrémentation et comparaison, 1 : init et last compar.).

Modèle pour la complexité temporelle

- Le coût d'une boucle `for(i=0;i<m;i++) p;` est `m` fois le coût de l'instruction `p` si ce coût ne dépend pas de la valeur de `i` : $1 + m(C(p) + 2) + 1 = O(mC(p))$ ($\ll +2 \gg$ pour l'incrémentation et comparaison, 1 : init et last compar.).
- Quand le coût du corps de la boucle dépend de la valeur du compteur `i`, le coût total de la boucle est la somme des coûts du corps de la boucle pour chaque valeur de `i`, plus l'incrémentation : $3 + \sum_{i=0}^{m-1}(C(p_i) + 2) = O(\sum_{i=0}^{m-1} C(p_i))$. On peut donc négliger l'initialisation de `i`, ses incrémentations et comparaisons.
  Le coût d'une boucle `for` ne peut pas être inférieur à $O(m)$ car il y a toujours l'incrémentation du compteur (sauf si dans le corps de boucle se trouve une instruction d'arrêt ou de retour).

Modèle pour la complexité temporelle

- Le coût d'une boucle `for(i=0;i<m;i++) p;` est `m` fois le coût de l'instruction `p` si ce coût ne dépend pas de la valeur de `i` : $1 + m(C(p) + 2) + 1 = O(mC(p))$ ($\ll +2 \gg$ pour l'incrémentation et comparaison, 1 : init et last compar.).
- Quand le coût du corps de la boucle dépend de la valeur du compteur `i`, le coût total de la boucle est la somme des coûts du corps de la boucle pour chaque valeur de `i`, plus l'incrémentation : $3 + \sum_{i=0}^{m-1}(C(p_i) + 2) = O(\sum_{i=0}^{m-1} C(p_i))$. On peut donc négliger l'initialisation de `i`, ses incrémentations et comparaisons.
  Le coût d'une boucle `for` ne peut pas être inférieur à $O(m)$ car il y a toujours l'incrémentation du compteur (sauf si dans le corps de boucle se trouve une instruction d'arrêt ou de retour).
- Coût d'une instruction `while` : le nombre d'itérations n'est pas connu à priori. On évalue le nombre d'itérations. Puis on applique la même règle que pour une boucle `for` en y ajoutant la complexité du test.

Complexité temporelle (cas d'une récursion)

- On essaye d'établir une relation de récurrence entre $C(n)$ (la complexité pour une donnée de taille n) et la complexité des appels internes plus la complexité hors appel récursif.

Complexité temporelle (cas d'une récursion)

- On essaye d'établir une relation de récurrence entre $C(n)$ (la complexité pour une donnée de taille n) et la complexité des appels internes plus la complexité hors appel récursif.
- Complexité $C(n)$ en nombre de multiplications pour

```
1  int fact(int n){
2      assert(n>=0);//n supposé positif
3      if (n<2)
4          return 1;
5      else
6          return n * fact(n-1);
7   }
8
```

La version présentée n'est pas en *récursion terminale* (terme expliqué plus tard dans l'année).

Complexité temporelle (Récursion)

- Complexité $C(n)$ en nombre d'opérations à coût constant pour

```
1  int fact(int n){//n supposé positif
2        if (n<2) return 1; else return n * fact(n-1);
3
```

Complexité temporelle (Récursion)

- Complexité $C(n)$ en nombre d'opérations à coût constant pour

```
1  int fact(int n){//n supposé positif
2       if (n<2) return 1; else return n * fact(n-1);
3
```

  - $C(0) = C(1) = O(1)$ (deux comparaisons `n<2` et `n>=0`). En fait, on pourrait aussi compter l'opération d'écriture à l'adresse de retour et d'autres opérations cachées, mais ce n'est pas si important car on veut un comportement assymptotique. On simplifie $C(0) = 1 = C(1)$.

Complexité temporelle (Récursion)

- Complexité $C(n)$ en nombre d'opérations à coût constant pour

```
1  int fact(int n){//n supposé positif
2       if (n<2) return 1; else return n * fact(n-1);
3
```

  - $C(0) = C(1) = O(1)$ (deux comparaisons `n<2` et `n>=0`). En fait, on pourrait aussi compter l'opération d'écriture à l'adresse de retour et d'autres opérations cachées, mais ce n'est pas si important car on veut un comportement assymptotique. On simplifie $C(0) = 1 = C(1)$.
  - $C(n) = C(n-1) + 3$ (on compte `n<2` ; `n-1` ; `n * ...`).

Complexité temporelle (Récursion)

- Complexité $C(n)$ en nombre d'opérations à coût constant pour

```
1  int fact(int n){//n supposé positif
2       if (n<2) return 1; else return n * fact(n-1);
3
```

  - $C(0) = C(1) = O(1)$ (deux comparaisons `n<2` et `n>=0`). En fait, on pourrait aussi compter l'opération d'écriture à l'adresse de retour et d'autres opérations cachées, mais ce n'est pas si important car on veut un comportement assymptotique. On simplifie $C(0) = 1 = C(1)$.
  - $C(n) = C(n-1) + 3$ (on compte `n<2` ; `n-1` ; `n * ...`).
- Pour cette suite *arithmétique*, on obtient :

$$C(n) = C(n-1)+3 = (C(n-2)+3)+3 = \cdots = \underbrace{C(1)}_{\text{cte}} + 3(n-1) = O(n)$$

Il est plus simple de prendre $C(n) = C(n-1) + 1$ (même ODG).

Complexité en mémoire (cas d'une récursion)

```
1  int fact(int
2      if (n<2) return 1; else return n * fact(n-1);
3   }
```

Complexité en mémoire (cas d'une récursion)

```
1  int fact(int
2      if (n<2) return 1; else return n * fact(n-1);
3   }
```

- Dans cette version non récursive terminale, pour renvoyer $n!$ il est nécessaire de d'abord calculer $(n-1)!$ puis de multiplier par n. Ainsi il faut garder n en mémoire.

Complexité en mémoire (cas d'une récursion)

```
1  int fact(int
2      if (n<2) return 1; else return n * fact(n-1);
3   }
```

- Dans cette version non récursive terminale, pour renvoyer $n!$ il est nécessaire de d'abord calculer $(n-1)!$ puis de multiplier par n. Ainsi il faut garder n en mémoire.
- Pour renvoyer $(n-1)!$ il est nécessaire de d'abord calculer $(n-2)!$ puis de multiplier par $n-1$. Ainsi il faut garder $n-1$ en mémoire.

Complexité en mémoire (cas d'une récursion)

```
1  int fact(int
2      if (n<2) return 1; else return n * fact(n-1);
3   }
```

- Dans cette version non récursive terminale, pour renvoyer $n!$ il est nécessaire de d'abord calculer $(n-1)!$ puis de multiplier par n. Ainsi il faut garder n en mémoire.
- Pour renvoyer $(n-1)!$ il est nécessaire de d'abord calculer $(n-2)!$ puis de multiplier par $n-1$. Ainsi il faut garder $n-1$ en mémoire.
- Au bout de la chaîne, lorsque enfin le test $n < 2$ est positif, les valeurs $n, n-1, \ldots, 2$ sont stockées quelque part (en fait *dans la pile*) en vue de leur utilisation future. La complexité en mémoire est donc linéaire.

Complexité en mémoire (cas d'une récursion)

```
1  int fact(int
2      if (n<2) return 1; else return n * fact(n-1);
3   }
```

- Dans cette version non récursive terminale, pour renvoyer $n!$ il est nécessaire de d'abord calculer $(n-1)!$ puis de multiplier par n. Ainsi il faut garder n en mémoire.
- Pour renvoyer $(n-1)!$ il est nécessaire de d'abord calculer $(n-2)!$ puis de multiplier par $n-1$. Ainsi il faut garder $n-1$ en mémoire.
- Au bout de la chaîne, lorsque enfin le test $n < 2$ est positif, les valeurs $n, n-1, \ldots, 2$ sont stockées quelque part (en fait *dans la pile*) en vue de leur utilisation future. La complexité en mémoire est donc linéaire.
- Or, une version impérative avec boucle `while` permet une complexité spatiale en $O(1)$. On peut aussi écrire une *récursion terminale* pour le même résultat.

Calculs simplifiés

- En général, on ne s'encombre pas avec une trop grande précision.

Calculs simplifiés

- En général, on ne s'encombre pas avec une trop grande précision.
- Si $C(n) = C(n-1) + f(n)$ on choisit l'expression la plus simple ayant même ODG que f. Par exemple :

Calculs simplifiés

- En général, on ne s'encombre pas avec une trop grande précision.
- Si $C(n) = C(n-1) + f(n)$ on choisit l'expression la plus simple ayant même ODG que f. Par exemple :
  - n^2 si $f(n) = 3n^2 + 5n + 15$.

Calculs simplifiés

- En général, on ne s'encombre pas avec une trop grande précision.
- Si $C(n) = C(n-1) + f(n)$ on choisit l'expression la plus simple ayant même ODG que f. Par exemple :
  - n^2 si $f(n) = 3n^2 + 5n + 15$.
  - $n2^n$ si $f(n) = 6n2^n + 30n^5 + \log n$

Calculs simplifiés

- En général, on ne s'encombre pas avec une trop grande précision.
- Si $C(n) = C(n-1) + f(n)$ on choisit l'expression la plus simple ayant même ODG que f. Par exemple :
  - n^2 si $f(n) = 3n^2 + 5n + 15$.
  - $n2^n$ si $f(n) = 6n2^n + 30n^5 + \log n$
- Seuls cas où il faut être précis : lorsqu'on s'intéresse à une opération en particulier (Exemples : multiplications d'entiers ; écritures dans un tableau...).

Limites du modèle

- Le modèle de complexité présenté est une approximation de la réalité.

Limites du modèle

- Le modèle de complexité présenté est une approximation de la réalité.
- Le modèle ne tient pas compte du fait que la taille des entiers peut être grande (surtout vrai en Python). La multiplication de deux très grands nombres est bien entendu plus coûteuse que celle de $2 * 3$.
  Il faudra par exemple d'abord lire tous les chiffres des opérandes et écrire ceux du résultat. Plus toutes les autres opérations logicielles que Python ajoute au traitement processeur.

Limites du modèle

- Le modèle de complexité présenté est une approximation de la réalité.
- Le modèle ne tient pas compte du fait que la taille des entiers peut être grande (surtout vrai en Python). La multiplication de deux très grands nombres est bien entendu plus coûteuse que celle de $2 * 3$.
  Il faudra par exemple d'abord lire tous les chiffres des opérandes et écrire ceux du résultat. Plus toutes les autres opérations logicielles que Python ajoute au traitement processeur.
- Autre exemple, une très grande liste (ou matrice) ne tiendra pas en mémoire vive : il faudra faire des accès disques pour parcourir les éléments.

Limites du modèle

- Le modèle de complexité présenté est une approximation de la réalité.
- Le modèle ne tient pas compte du fait que la taille des entiers peut être grande (surtout vrai en Python). La multiplication de deux très grands nombres est bien entendu plus coûteuse que celle de $2 * 3$. Il faudra par exemple d'abord lire tous les chiffres des opérandes et écrire ceux du résultat. Plus toutes les autres opérations logicielles que Python ajoute au traitement processeur.
- Autre exemple, une très grande liste (ou matrice) ne tiendra pas en mémoire vive : il faudra faire des accès disques pour parcourir les éléments.
- En CPGE, on convient que `malloc(..)` est une opération à coût constant (ce qui est faux). En revanche `realloc(..)` peut-être éventuellement en $O(1)$ dans des cas où il n'y a pas de copie à faire.

Limites du modèle

- Le modèle de complexité présenté est une approximation de la réalité.
- Le modèle ne tient pas compte du fait que la taille des entiers peut être grande (surtout vrai en Python). La multiplication de deux très grands nombres est bien entendu plus coûteuse que celle de $2 * 3$. Il faudra par exemple d'abord lire tous les chiffres des opérandes et écrire ceux du résultat. Plus toutes les autres opérations logicielles que Python ajoute au traitement processeur.
- Autre exemple, une très grande liste (ou matrice) ne tiendra pas en mémoire vive : il faudra faire des accès disques pour parcourir les éléments.
- En CPGE, on convient que `malloc(..)` est une opération à coût constant (ce qui est faux). En revanche `realloc(..)` peut-être éventuellement en $O(1)$ dans des cas où il n'y a pas de copie à faire.
- L'évaluation du temps mis par un algorithme pour s'exécuter est un domaine de recherche à part entière, car elle se révèle parfois très difficile.

Meilleur des cas

- La complexité au pire est la plus significative, mais il peut être utile de connaître aussi la complexité dans le meilleur des cas, pour avoir une borne inférieure du temps d'exécution d'un algorithme.

Meilleur des cas

- La complexité au pire est la plus significative, mais il peut être utile de connaître aussi la complexité dans le meilleur des cas, pour avoir une borne inférieure du temps d'exécution d'un algorithme.
- En particulier, si la complexité dans le meilleur et dans le pire des cas sont du même ordre, cela signifie que le temps d'exécution de l'algorithme est relativement indépendant des données et ne dépend que de la taille du problème.

Complexité en moyenne

- Parler de moyenne des temps d'exécution n'a de sens que si l'on a une idée de la fréquence des différentes données possibles pour un même problème de taille n.

Complexité en moyenne

- Parler de moyenne des temps d'exécution n'a de sens que si l'on a une idée de la fréquence des différentes données possibles pour un même problème de taille n.
- Les calculs de complexité moyenne recourent aux notions définies en mathématiques dans le cadre de la théorie des probabilités et des statistiques.

Complexité en moyenne

- Parler de moyenne des temps d'exécution n'a de sens que si l'on a une idée de la fréquence des différentes données possibles pour un même problème de taille n.
- Les calculs de complexité moyenne recourent aux notions définies en mathématiques dans le cadre de la théorie des probabilités et des statistiques.
- La complexité en moyenne n'est, en général, pas attendue dans les sujets mais on en verra quelques exemples.

Complexité en espace

- Il n'y a pas que le temps d'exécution des algorithmes qui intéresse les informaticiens.

Complexité en espace

- Il n'y a pas que le temps d'exécution des algorithmes qui intéresse les informaticiens.
- Une autre ressource importante en informatique est la mémoire.

Complexité en espace

- Il n'y a pas que le temps d'exécution des algorithmes qui intéresse les informaticiens.
- Une autre ressource importante en informatique est la mémoire.
- On appelle *complexité en espace* d'un algorithme la place nécessaire en mémoire pour faire fonctionner cet algorithme *en dehors des paramètres du programme*.

Complexité en espace

- Il n'y a pas que le temps d'exécution des algorithmes qui intéresse les informaticiens.
- Une autre ressource importante en informatique est la mémoire.
- On appelle *complexité en espace* d'un algorithme la place nécessaire en mémoire pour faire fonctionner cet algorithme *en dehors des paramètres du programme*.
- Elle s'exprime également sous la forme d'un $O(f(n))$ où n est la taille du problème.

Complexité en mémoire

- Pour les codes impératifs, il suffit de faire le total des tailles en mémoire des différentes variables utilisées. Avec ce principe, la complexité en mémoire est inférieure à la complexité temporelle. On suppose qu'une opération d'écriture en mémoire prend un temps $O(1)$

Complexité en mémoire

- Pour les codes impératifs, il suffit de faire le total des tailles en mémoire des différentes variables utilisées. Avec ce principe, la complexité en mémoire est inférieure à la complexité temporelle.
  On suppose qu'une opération d'écriture en mémoire prend un temps $O(1)$
- La seule vraie exception à la règle est le cas des fonctions récursives, qui cachent souvent une complexité en espace élevée du fait de l'empilement des stack frames.
  La récursion terminale est une bonne solution à ce problème.

Objectif

Les pages qui suivent sont données pour la culture.

- Dans cette sous-section, on présente une astuce très utilisée par les cartes graphiques 3D pour calculer la racine carrée.

Objectif

Les pages qui suivent sont données pour la culture.

- Dans cette sous-section, on présente une astuce très utilisée par les cartes graphiques 3D pour calculer la racine carrée.
- Les cartes graphiques font en effet beaucoup de calculs géométriques (par exemple pour déterminer si un projectile rencontre une surface comme un mur : intersection droite plan).

Objectif

Les pages qui suivent sont données pour la culture.

- Dans cette sous-section, on présente une astuce très utilisée par les cartes graphiques 3D pour calculer la racine carrée.
- Les cartes graphiques font en effet beaucoup de calculs géométriques (par exemple pour déterminer si un projectile rencontre une surface comme un mur : intersection droite plan).
- Ces calculs font intervenir des notions comme le vecteur normal à une surface (il faut diviser un produit vectoriel par une norme, donc la racine carrée de la somme des carrés des coordonnées du vecteur.

Objectif

Les pages qui suivent sont données pour la culture.

- Dans cette sous-section, on présente une astuce très utilisée par les cartes graphiques 3D pour calculer la racine carrée.
- Les cartes graphiques font en effet beaucoup de calculs géométriques (par exemple pour déterminer si un projectile rencontre une surface comme un mur : intersection droite plan).
- Ces calculs font intervenir des notions comme le vecteur normal à une surface (il faut diviser un produit vectoriel par une norme, donc la racine carrée de la somme des carrés des coordonnées du vecteur.
- D'où l'importance d'un calcul efficace de la racine carrée.

Méthode de Héron

Calcul de $\sqrt{A}$

- Par dichotomie avec une précision ε en résolvant $x^2 - A = 0$ sur $[0; A+1]$. Calcul en $O(\log(\frac{A+1}{\varepsilon}))$. Ce n'est déjà pas si mal.

Méthode de Héron

Calcul de $\sqrt{A}$

- Par dichotomie avec une précision ε en résolvant $x^2 - A = 0$ sur $[0; A+1]$. Calcul en $O(\log(\frac{A+1}{\varepsilon}))$. Ce n'est déjà pas si mal.
- Méthode de Héron (une spécialisation de la méthode de Newton à cet objectif particulier) :

Méthode de Héron

Calcul de $\sqrt{A}$

- Par dichotomie avec une précision ε en résolvant $x^2 - A = 0$ sur $[0; A+1]$. Calcul en $O(\log(\frac{A+1}{\varepsilon}))$. Ce n'est déjà pas si mal.
- Méthode de Héron (une spécialisation de la méthode de Newton à cet objectif particulier) :
  - Choisir x_0 initial puis $x_{k+1} = \frac{1}{2}(x_k + \frac{A}{x_k})$.

Méthode de Héron

Calcul de $\sqrt{A}$

- Par dichotomie avec une précision ε en résolvant $x^2 - A = 0$ sur $[0; A+1]$. Calcul en $O(\log(\frac{A+1}{\varepsilon}))$. Ce n'est déjà pas si mal.
- Méthode de Héron (une spécialisation de la méthode de Newton à cet objectif particulier) :
  - Choisir x_0 initial puis $x_{k+1} = \frac{1}{2}(x_k + \frac{A}{x_k})$.
  - Convergence quadratique : le nombre de chiffres significatifs double à chaque itération. Au bout de quelques itérations (15 ou 20), on atteint la précision maximale possible sur un ordinateur de bureau. On peut alors considérer que le calcul de la racine carrée est quasiment en temps constant (pas tout à fait).

John Carmack et $\frac{1}{\sqrt{A}}$

John Carmack est un informaticien américain à l'origine de nombreux jeux vidéos, dont Wolfenstein 3D, Doom et Quake.

- Les cartes graphiques 3D avec processeur 32 bits (la majorité des cartes du marché en 2020) utilisent l'*astuce de John Carmack* et son *nombre magique* pour calculer, non pas $\sqrt{A}$ mais $\frac{1}{\sqrt{A}}$.

John Carmack et $\frac{1}{\sqrt{A}}$

John Carmack est un informaticien américain à l'origine de nombreux jeux vidéos, dont Wolfenstein 3D, Doom et Quake.

- Les cartes graphiques 3D avec processeur 32 bits (la majorité des cartes du marché en 2020) utilisent l'*astuce de John Carmack* et son *nombre magique* pour calculer, non pas $\sqrt{A}$ mais $\frac{1}{\sqrt{A}}$.
- Le flottant A codé sur 32 bits en $A = (-1)^0 2^{e_1}(1 + m_1)$ est considéré comme l'entier $I_1 = 2^{23}(127 + e_1 + m_1)$: `int i1 = *(int *) &a;` .

John Carmack et $\frac{1}{\sqrt{A}}$

John Carmack est un informaticien américain à l'origine de nombreux jeux vidéos, dont Wolfenstein 3D, Doom et Quake.

- Les cartes graphiques 3D avec processeur 32 bits (la majorité des cartes du marché en 2020) utilisent l'*astuce de John Carmack* et son *nombre magique* pour calculer, non pas $\sqrt{A}$ mais $\frac{1}{\sqrt{A}}$.
- Le flottant A codé sur 32 bits en $A = (-1)^0 2^{e_1}(1 + m_1)$ est considéré comme l'entier $I_1 = 2^{23}(127 + e_1 + m_1)$: `int i1 = *(int *) &a;` .
- Carmack utilise son nombre magique, l'entier codé en hexadécimal $0x5f3759df$ (on a trouvé mieux depuis) et lui soustrait $\lfloor \frac{I_1}{2} \rfloor$. Il convertit ensuite cet entier en flottant.

John Carmack et $\frac{1}{\sqrt{A}}$

John Carmack est un informaticien américain à l'origine de nombreux jeux vidéos, dont Wolfenstein 3D, Doom et Quake.

- Les cartes graphiques 3D avec processeur 32 bits (la majorité des cartes du marché en 2020) utilisent l'*astuce de John Carmack* et son *nombre magique* pour calculer, non pas $\sqrt{A}$ mais $\frac{1}{\sqrt{A}}$.
- Le flottant A codé sur 32 bits en $A = (-1)^0 2^{e_1}(1 + m_1)$ est considéré comme l'entier $I_1 = 2^{23}(127 + e_1 + m_1)$: `int i1 = *(int *) &a;` .
- Carmack utilise son nombre magique, l'entier codé en hexadécimal $0x5f3759df$ (on a trouvé mieux depuis) et lui soustrait $\lfloor \frac{I_1}{2} \rfloor$. Il convertit ensuite cet entier en flottant.
- Quelle que soit la valeur de A, on peut montrer que le flottant retourné est proche de $\frac{1}{\sqrt{A}}$ avec une erreur au pire de $2/1000$! ! Complexité temporelle en $O(1)$.

Exponentiation naïve

```
let rec pow x n = match n with
  | 0 -> 1
  | _ -> x * pow x (n-1);;
```

- La complexité (dans tous les cas) est de la forme $C_n = C_{n-1} + 1$ (déjà étudié pour la factorielle).

Exponentiation naïve

```
1 let rec pow x n = match n with
2   | 0 -> 1
3   | _ -> x * pow x (n-1);;
```

- La complexité (dans tous les cas) est de la forme $C_n = C_{n-1} + 1$ (déjà étudié pour la factorielle).
- C'est une suite arithmétique de raison 1.

Exponentiation naïve

```
1 let rec pow x n = match n with
2   | 0 -> 1
3   | _ -> x * pow x (n-1);;
```

- La complexité (dans tous les cas) est de la forme $C_n = C_{n-1} + 1$ (déjà étudié pour la factorielle).
- C'est une suite arithmétique de raison 1.
- On a donc $C_n = \Theta(n)$

Exponentiation naïve

```
1 let rec pow x n = match n with
2   | 0 -> 1
3   | _ -> x * pow x (n-1);;
```

- La complexité (dans tous les cas) est de la forme $C_n = C_{n-1} + 1$ (déjà étudié pour la factorielle).
- C'est une suite arithmétique de raison 1.
- On a donc $C_n = \Theta(n)$
- A noter que le second membre de la relation de récurrence est 1 puisqu'il y a un nombre d'opérations à coût constants en $O(1)$ (mais pas égal à 1 : il y a au moins le filtrage, une soustraction, une multiplication). On se simplifie la vie en prenant le terme le plus simple possible dans la classe de complexité

Insertion dans une liste triée

```
let rec insert l x =  match l with
  | [] -> [x]
  | y::q when y>x -> x::l
  | y::q -> y::(insert q x);;
```

- Hors appel récursif, il faut compter l'ajout en tête de liste (règle 3) et le filtrage lui-même (il n'a pas un coût nul, bien que constant).

Insertion dans une liste triée

```
let rec insert l x =  match l with
  | [] -> [x]
  | y::q when y>x -> x::l
  | y::q -> y::(insert q x);;
```

- Hors appel récursif, il faut compter l'ajout en tête de liste (règle 3) et le filtrage lui-même (il n'a pas un coût nul, bien que constant).
- La relation de récurrence pour la complexité au pire est donc $C_n = C_{n-1} + O(1)$ ce que l'on simplifie en $C_n = C_{n-1} + 1$.

Insertion dans une liste triée

```
let rec insert l x =  match l with
  | [] -> [x]
  | y::q when y>x -> x::l
  | y::q -> y::(insert q x);;
```

- Hors appel récursif, il faut compter l'ajout en tête de liste (règle 3) et le filtrage lui-même (il n'a pas un coût nul, bien que constant).
- La relation de récurrence pour la complexité au pire est donc $C_n = C_{n-1} + O(1)$ ce que l'on simplifie en $C_n = C_{n-1} + 1$.
- Au mieux $C_n = 1$ (règles de filtrage 1 et 2).

Insertion dans une liste triée

```
let rec insert l x =  match l with
  | [] -> [x]
  | y::q when y>x -> x::l
  | y::q -> y::(insert q x);;
```

- Hors appel récursif, il faut compter l'ajout en tête de liste (règle 3) et le filtrage lui-même (il n'a pas un coût nul, bien que constant).
- La relation de récurrence pour la complexité au pire est donc $C_n = C_{n-1} + O(1)$ ce que l'on simplifie en $C_n = C_{n-1} + 1$.
- Au mieux $C_n = 1$ (règles de filtrage 1 et 2).
- Complexité au pire $O(n)$; au mieux $O(1)$.

Tri par insertion

```
let rec tri l =
    match l with
    | [] -> []
    | x::q -> insert x (tri q);;
```

- Insertion dans `tri q` au pire en $O(n)$ où $n = |\ell|$.
  Complexité au pire $C_n = C_{n-1} + n$ (n pour l'insertion au pire).

Tri par insertion

```
let rec tri l =
    match l with
    | [] -> []
    | x::q -> insert x (tri q);;
```

- Insertion dans `tri q` au pire en $O(n)$ où $n = |\ell|$.
  Complexité au pire $C_n = C_{n-1} + n$ (n pour l'insertion au pire).
- On a donc $C_n - C_{n-1} = n$. Il vient par télescopage

$$C_n - \underbrace{C_0}_{\text{cte}} = \sum_{i=1}^{n} C_i - C_{i-1} = \sum_{i=0}^{n} i = O(n^2)$$

Donc $C_n = O(n^2)$

Tri par insertion

```
let rec tri l =
    match l with
    | [] -> []
    | x::q -> insert x (tri q);;
```

- Insertion dans `tri q` au pire en $O(n)$ où $n = |\ell|$.
  Complexité au pire $C_n = C_{n-1} + n$ (n pour l'insertion au pire).
- On a donc $C_n - C_{n-1} = n$. Il vient par télescopage

$$C_n - \underbrace{C_0}_{\text{cte}} = \sum_{i=1}^{n} C_i - C_{i-1} = \sum_{i=0}^{n} i = O(n^2)$$

  Donc $C_n = O(n^2)$
- Complexité au mieux $C_n = C_{n-1} + 1$. Donc $C_n = O(n)$.

Diviser pour régner

Méthode

- découper la donnée que l'on doit traiter en deux parts (ou plus) de tailles proches ;

Diviser pour régner

Méthode

- découper la donnée que l'on doit traiter en deux parts (ou plus) de tailles proches ;
- Résoudre le problème sur les parties plus petites ;

Diviser pour régner

Méthode

- découper la donnée que l'on doit traiter en deux parts (ou plus) de tailles proches ;
- Résoudre le problème sur les parties plus petites ;
- Combiner les résultats obtenus pour construire le résultat correspondant à la donnée initiale.

Diviser pour régner

Méthode

- découper la donnée que l'on doit traiter en deux parts (ou plus) de tailles proches ;
- Résoudre le problème sur les parties plus petites ;
- Combiner les résultats obtenus pour construire le résultat correspondant à la donnée initiale.

- Une donnée de taille n est découpée en deux autres données de tailles $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$

Diviser pour régner

Méthode

- découper la donnée que l'on doit traiter en deux parts (ou plus) de tailles proches ;
- Résoudre le problème sur les parties plus petites ;
- Combiner les résultats obtenus pour construire le résultat correspondant à la donnée initiale.

- Une donnée de taille n est découpée en deux autres données de tailles $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$
- La complexité obéit à une relation de la forme :

$$C_n = aC_{\lfloor \frac{n}{2} \rfloor} + bC_{\lceil \frac{n}{2} \rceil} + f(n)$$

où $(a, b) \neq (0, 0)$ sont des constantes associées au problème et $f(n)$ correspond au coût total du partage et de la recombinaison.

Diviser pour régner

Méthode

- découper la donnée que l'on doit traiter en deux parts (ou plus) de tailles proches ;
- Résoudre le problème sur les parties plus petites ;
- Combiner les résultats obtenus pour construire le résultat correspondant à la donnée initiale.

- Une donnée de taille n est découpée en deux autres données de tailles $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$
- La complexité obéit à une relation de la forme :

$$C_n = aC_{\lfloor \frac{n}{2} \rfloor} + bC_{\lceil \frac{n}{2} \rceil} + f(n)$$

où $(a, b) \neq (0, 0)$ sont des constantes associées au problème et $f(n)$ correspond au coût total du partage et de la recombinaison.

- Une telle récurrence est dite de type ≪ diviser pour régner ≫.

Choix du dernier terme

Proposition

Soient a,b deux entiers positifs tels que $ab \neq 0$ *et* f, g *deux fonctions croissantes de même ordre de grandeur. Alors les suites* (U_n) *et* (V_n) *telles que* $U_0 = V_0$ *et pour tout* $n \in \mathbb{N}^*$ *:*

$$\begin{aligned} U_n &= aU_{\lfloor n/2 \rfloor} + bU_{\lceil n/2 \rceil} + f(n) \\ V_n &= aV_{\lfloor n/2 \rfloor} + bV_{\lceil n/2 \rceil} + g(n) \end{aligned}$$

sont du même ordre de grandeur.

Cette proposition (admise) justifie qu'on remplace le dernier terme par un terme plus simple de même ordre de grandeur (donc 1 pour $O(1)$, n^2 pour $O(n^2)$ etc.)

Croissance de la complexité en fonction de la taille n de la donnée.

On considère $U_n = f(n) + aU_{\lfloor n/2 \rfloor} + bU_{\lceil n/2 \rceil}$ avec f croissante et a, b positifs. On pose $U_0 = U_1 = 1$ parce que ça nous arrange et qu'on étudie un comportement asymptotique.

On montre que $U_{n-1} \leq U_n$ pour $n \geq 1$

Croissance de la complexité en fonction de la taille n de la donnée.

On considère $U_n = f(n) + aU_{\lfloor n/2 \rfloor} + bU_{\lceil n/2 \rceil}$ avec f croissante et a, b positifs. On pose $U_0 = U_1 = 1$ parce que ça nous arrange et qu'on étudie un comportement asymptotique.

On montre que $U_{n-1} \leq U_n$ pour $n \geq 1$

Cas de base $1 = U_0 \leq U_1 = 1$

Croissance de la complexité en fonction de la taille n de la donnée.

On considère $U_n = f(n) + aU_{\lfloor n/2 \rfloor} + bU_{\lceil n/2 \rceil}$ avec f croissante et a, b positifs. On pose $U_0 = U_1 = 1$ parce que ça nous arrange et qu'on étudie un comportement asymptotique.

On montre que $U_{n-1} \leq U_n$ pour $n \geq 1$

Cas de base $1 = U_0 \leq U_1 = 1$

Hérédité Si $n > 1$ et $U_k \geq U_q$ pour tout $n > k \geq q$.

Croissance de la complexité en fonction de la taille n de la donnée.

On considère $U_n = f(n) + aU_{\lfloor n/2 \rfloor} + bU_{\lceil n/2 \rceil}$ avec f croissante et a, b positifs. On pose $U_0 = U_1 = 1$ parce que ça nous arrange et qu'on étudie un comportement asymptotique.

On montre que $U_{n-1} \leq U_n$ pour $n \geq 1$

Cas de base $1 = U_0 \leq U_1 = 1$

Hérédité Si $n > 1$ et $U_k \geq U_q$ pour tout $n > k \geq q$.

- $U_n = f(n) + aU_{\lfloor n/2 \rfloor} + bU_{\lceil n/2 \rceil}$ et

Croissance de la complexité en fonction de la taille n de la donnée.

On considère $U_n = f(n) + aU_{\lfloor n/2 \rfloor} + bU_{\lceil n/2 \rceil}$ avec f croissante et a, b positifs. On pose $U_0 = U_1 = 1$ parce que ça nous arrange et qu'on étudie un comportement asymptotique.

On montre que $U_{n-1} \leq U_n$ pour $n \geq 1$

Cas de base $1 = U_0 \leq U_1 = 1$

Hérédité Si $n > 1$ et $U_k \geq U_q$ pour tout $n > k \geq q$.

- $U_n = f(n) + aU_{\lfloor n/2 \rfloor} + bU_{\lceil n/2 \rceil}$ et
- $U_{n-1} = f(n-1) + aU_{\lfloor (n-1)/2 \rfloor} + bU_{\lceil (n-1)/2 \rceil}$

Croissance de la complexité en fonction de la taille n de la donnée.

On considère $U_n = f(n) + aU_{\lfloor n/2 \rfloor} + bU_{\lceil n/2 \rceil}$ avec f croissante et a, b positifs. On pose $U_0 = U_1 = 1$ parce que ça nous arrange et qu'on étudie un comportement asymptotique.

On montre que $U_{n-1} \leq U_n$ pour $n \geq 1$

Cas de base $1 = U_0 \leq U_1 = 1$

Hérédité Si $n > 1$ et $U_k \geq U_q$ pour tout $n > k \geq q$.

- $U_n = f(n) + aU_{\lfloor n/2 \rfloor} + bU_{\lceil n/2 \rceil}$ et
- $U_{n-1} = f(n-1) + aU_{\lfloor (n-1)/2 \rfloor} + bU_{\lceil (n-1)/2 \rceil}$
- Or, les fonctions parties entières sont croissantes, donc $aU_{\lfloor (n-1)/2 \rfloor} \leq aU_{\lfloor n/2 \rfloor}$ et $bU_{\lceil (n-1)/2 \rceil} \leq bU_{\lceil n/2 \rceil}$ par HR. Et $f(n-1) \leq f(n)$ par hypothèse.

Croissance de la complexité en fonction de la taille n de la donnée.

On considère $U_n = f(n) + aU_{\lfloor n/2 \rfloor} + bU_{\lceil n/2 \rceil}$ avec f croissante et a, b positifs. On pose $U_0 = U_1 = 1$ parce que ça nous arrange et qu'on étudie un comportement asymptotique.

On montre que $U_{n-1} \leq U_n$ pour $n \geq 1$

Cas de base $1 = U_0 \leq U_1 = 1$

Hérédité Si $n > 1$ et $U_k \geq U_q$ pour tout $n > k \geq q$.

- $U_n = f(n) + aU_{\lfloor n/2 \rfloor} + bU_{\lceil n/2 \rceil}$ et
- $U_{n-1} = f(n-1) + aU_{\lfloor (n-1)/2 \rfloor} + bU_{\lceil (n-1)/2 \rceil}$
- Or, les fonctions parties entières sont croissantes, donc $aU_{\lfloor (n-1)/2 \rfloor} \leq aU_{\lfloor n/2 \rfloor}$ et $bU_{\lceil (n-1)/2 \rceil} \leq bU_{\lceil n/2 \rceil}$ par HR. Et $f(n-1) \leq f(n)$ par hypothèse.
- Donc $U_n \geq U_{n-1}$. Il vient que $U_k \geq U_q$ pour tout $n \geq k \geq q$.

Croissance de la complexité en fonction de la taille n de la donnée.

On considère $U_n = f(n) + aU_{\lfloor n/2 \rfloor} + bU_{\lceil n/2 \rceil}$ avec f croissante et a, b positifs. On pose $U_0 = U_1 = 1$ parce que ça nous arrange et qu'on étudie un comportement asymptotique.

On montre que $U_{n-1} \leq U_n$ pour $n \geq 1$

Cas de base $1 = U_0 \leq U_1 = 1$

Hérédité Si $n > 1$ et $U_k \geq U_q$ pour tout $n > k \geq q$.

- $U_n = f(n) + aU_{\lfloor n/2 \rfloor} + bU_{\lceil n/2 \rceil}$ et
- $U_{n-1} = f(n-1) + aU_{\lfloor (n-1)/2 \rfloor} + bU_{\lceil (n-1)/2 \rceil}$
- Or, les fonctions parties entières sont croissantes, donc $aU_{\lfloor (n-1)/2 \rfloor} \leq aU_{\lfloor n/2 \rfloor}$ et $bU_{\lceil (n-1)/2 \rceil} \leq bU_{\lceil n/2 \rceil}$ par HR. Et $f(n-1) \leq f(n)$ par hypothèse.
- Donc $U_n \geq U_{n-1}$. Il vient que $U_k \geq U_q$ pour tout $n \geq k \geq q$.

Donc $(U_n) \uparrow$. En pratique, on ne refait pas cette preuve et on admet que la complexité est croissante avec la taille des

Quelques observations

Dans ce qui suit, toutes les fonctions sont positives et $n \in \mathbb{N}$

- Pour $a > 0$, $\ln n = \Theta(\log_a)$ car $\log_a x = \dfrac{\ln x}{\ln a}$
- $n = \Theta(\lfloor n \rfloor)$ car pour $n > 2$

$$\frac{n}{2} < n - 1 < \lfloor n \rfloor \leq n$$

(de même $n = \Theta(\lceil n \rceil)$)

Recherche dichotomique dans un tableau trié

- On cherche un élément x dans un tableau trié ;

Pour varier les plaisirs, on propose une version récursive OCaml de cet algorithme (on a déjà vu une version impérative en C).
De plus, on considère que la position droite `d` est le dernier indice où on cherche x (et non pas, comme déjà étudié, le 1er indice où on ne le cherche pas).

Recherche dichotomique dans un tableau trié

- On cherche un élément x dans un tableau trié ;
- Si x est strictement plus grand que l'élément qui est au centre du tableau, on cherche x dans la moitié droite ;

Pour varier les plaisirs, on propose une version récursive OCaml de cet algorithme (on a déjà vu une version impérative en C).
De plus, on considère que la position droite `d` est le dernier indice où on cherche x (et non pas, comme déjà étudié, le 1er indice où on ne le cherche pas).

Recherche dichotomique dans un tableau trié

- On cherche un élément x dans un tableau trié ;
- Si x est strictement plus grand que l'élément qui est au centre du tableau, on cherche x dans la moitié droite ;
- S'il est plus petit, on cherche x dans la moitié gauche.

Pour varier les plaisirs, on propose une version récursive OCaml de cet algorithme (on a déjà vu une version impérative en C).
De plus, on considère que la position droite `d` est le dernier indice où on cherche x (et non pas, comme déjà étudié, le 1er indice où on ne le cherche pas).

Recherche dichotomique dans un tableau trié

- On cherche un élément x dans un tableau trié ;
- Si x est strictement plus grand que l'élément qui est au centre du tableau, on cherche x dans la moitié droite ;
- S'il est plus petit, on cherche x dans la moitié gauche.
- Lorsque la portion du tableau étudiée est de taille 1, on compare x avec l'unique élément de cette portion.

Pour varier les plaisirs, on propose une version récursive OCaml de cet algorithme (on a déjà vu une version impérative en C).
De plus, on considère que la position droite `d` est le dernier indice où on cherche x (et non pas, comme déjà étudié, le 1er indice où on ne le cherche pas).

Recherche dichotomique dans un tableau trié ↑

```
let dicho t x = (*chercher x dans le tableau trié t*)
  let rec cherche_entre g d =
    (*chercher entre g et d inclus*)
    if g = d then (x=t.(g))
    else let m = (g+d)/2 in
      if (x>t.(m)) then cherche_entre (m+1) d
      else cherche_entre g m
  in cherche_entre 0 (Array.length t -1);;
```

Recherche dichotomique dans un tableau trié ↑

```
1  let dicho t x = (*chercher x dans le tableau trié t*)
2    let rec cherche_entre g d =
3      (*chercher entre g et d inclus*)
4      if g = d then (x=t.(g))
5      else let m = (g+d)/2 in
6        if (x>t.(m)) then cherche_entre (m+1) d
7        else cherche_entre g m
8    in cherche_entre 0 (Array.length t -1);;
```

- A chaque étape, la zone de recherche est divisée par deux (on note $n/2$ la division euclidienne) :

```
x x x x x : tableau de taille impaire n
    m
Sous-tableaux de taille n/2 et n/2+1
x x x x : tableau de taille paire n
  m
2 sous-tableaux de taille n/2
```

Dichotomie

- Hors appels récursifs, il y a un nombre borné d'autres opérations, toutes en $O(1)$. Le coût total à chaque tour hors appels récursifs est donc $O(1)$.

Dichotomie

- Hors appels récursifs, il y a un nombre borné d'autres opérations, toutes en $O(1)$. Le coût total à chaque tour hors appels récursifs est donc $O(1)$.
- Pour un tableau de taille n, la complexité au pire est du type $C_n = C_{\lceil \frac{n}{2} \rceil} + O(1)$; simplifiée en $C_n = C_{\lceil \frac{n}{2} \rceil} + 1$ (on cherche dans le plus grand sous-tableau pour que ce soit le pire cas)

Dichotomie

- Hors appels récursifs, il y a un nombre borné d'autres opérations, toutes en $O(1)$. Le coût total à chaque tour hors appels récursifs est donc $O(1)$.
- Pour un tableau de taille n, la complexité au pire est du type $C_n = C_{\lceil \frac{n}{2} \rceil} + O(1)$; simplifiée en $C_n = C_{\lceil \frac{n}{2} \rceil} + 1$ (on cherche dans le plus grand sous-tableau pour que ce soit le pire cas)
- Si $n = 2^p$ alors $p = \log_2 n$ et

$$C_{2^p} = C_{2^{p-1}} + 1 = C_{2^{p-2}} + \underbrace{1+1}_{2} = \cdots = \underbrace{C_{2^0}}_{\text{cte}} + p = \Theta(\log_2 n)$$

Dichotomie

- Hors appels récursifs, il y a un nombre borné d'autres opérations, toutes en $O(1)$. Le coût total à chaque tour hors appels récursifs est donc $O(1)$.
- Pour un tableau de taille n, la complexité au pire est du type $C_n = C_{\lceil \frac{n}{2} \rceil} + O(1)$; simplifiée en $C_n = C_{\lceil \frac{n}{2} \rceil} + 1$ (on cherche dans le plus grand sous-tableau pour que ce soit le pire cas)
- Si $n = 2^p$ alors $p = \log_2 n$ et

$$C_{2^p} = C_{2^{p-1}} + 1 = C_{2^{p-2}} + \underbrace{1+1}_{2} = \cdots = \underbrace{C_{2^0}}_{\text{cte}} + p = \Theta(\log_2 n)$$

- Pour n quelconque, $2^{\lfloor \log_2 n \rfloor} \leq n < 2^{\lfloor \log_2 n \rfloor + 1}$. Comme la complexité est croissante avec la taille du tableau $C_n \leq C_{2^{\lfloor \log_2 n \rfloor + 1}} = O(\lfloor \log_2 n \rfloor + 1)$. Il vient $C_n = O(\log_2 n)$.

Dichotomie

- Hors appels récursifs, il y a un nombre borné d'autres opérations, toutes en $O(1)$. Le coût total à chaque tour hors appels récursifs est donc $O(1)$.
- Pour un tableau de taille n, la complexité au pire est du type $C_n = C_{\lceil \frac{n}{2} \rceil} + O(1)$; simplifiée en $C_n = C_{\lceil \frac{n}{2} \rceil} + 1$ (on cherche dans le plus grand sous-tableau pour que ce soit le pire cas)
- Si $n = 2^p$ alors $p = \log_2 n$ et

$$C_{2^p} = C_{2^{p-1}} + 1 = C_{2^{p-2}} + \underbrace{1+1}_{2} = \cdots = \underbrace{C_{2^0}}_{\text{cte}} + p = \Theta(\log_2 n)$$

- Pour n quelconque, $2^{\lfloor \log_2 n \rfloor} \leq n < 2^{\lfloor \log_2 n \rfloor + 1}$. Comme la complexité est croissante avec la taille du tableau $C_n \leq C_{2^{\lfloor \log_2 n \rfloor + 1}} = O(\lfloor \log_2 n \rfloor + 1)$. Il vient $C_n = O(\log_2 n)$.
- Idem minoration :

$$C_n \geq C_{2^{\lfloor \log_2 n \rfloor}} = \Omega(\lfloor \log_2 n \rfloor) : \text{Conclusion, } C_n = \Theta(\log_2 n)$$

Exponentiation rapide

```
1 let rec puissance_rapide x n = match n with
2   | 0 -> 1
3   | n when n mod 2 = 0 -> puissance_rapide (x*x) (n/2)
4   | _ -> x*puissance_rapide x (n-1) ;;
```

Exponentiation rapide

```
1 let rec puissance_rapide x n = match n with
2   | 0 -> 1
3   | n when n mod 2 = 0 -> puissance_rapide (x*x) (n/2)
4   | _ -> x*puissance_rapide x (n-1) ;;
```

- Principe $x^{2k} = (x^2)^k$ et $x^{2k+1} = x \cdot x^{2k}$

Exponentiation rapide

```
let rec puissance_rapide x n = match n with
  | 0 -> 1
  | n when n mod 2 = 0 -> puissance_rapide (x*x) (n/2)
  | _ -> x*puissance_rapide x (n-1) ;;
```

- Principe $x^{2k} = (x^2)^k$ et $x^{2k+1} = x \cdot x^{2k}$
- Complexité en nombre de multiplications

Exponentiation rapide

```
let rec puissance_rapide x n = match n with
  | 0 -> 1
  | n when n mod 2 = 0 -> puissance_rapide (x*x) (n/2)
  | _ -> x*puissance_rapide x (n-1) ;;
```

- Principe $x^{2k} = (x^2)^k$ et $x^{2k+1} = x \cdot x^{2k}$
- Complexité en nombre de multiplications
  - Lorsque n est paire $C_n = C_{n/2} + 1 = C_{\lfloor \frac{n}{2} \rfloor} + 1$.

Exponentiation rapide

```
let rec puissance_rapide x n = match n with
  | 0 -> 1
  | n when n mod 2 = 0 -> puissance_rapide (x*x) (n/2)
  | _ -> x*puissance_rapide x (n-1) ;;
```

- Principe $x^{2k} = (x^2)^k$ et $x^{2k+1} = x \cdot x^{2k}$
- Complexité en nombre de multiplications
  - Lorsque n est paire $C_n = C_{n/2} + 1 = C_{\lfloor \frac{n}{2} \rfloor} + 1$.
  - Si n est impaire (et > 1), $C_n = 1 + C_{n-1} = 1 + C_{(n-1)/2} + 1 = C_{\lfloor \frac{n}{2} \rfloor} + 2$.

Exponentiation rapide

```
let rec puissance_rapide x n = match n with
  | 0 -> 1
  | n when n mod 2 = 0 -> puissance_rapide (x*x) (n/2)
  | _ -> x*puissance_rapide x (n-1) ;;
```

- Principe $x^{2k} = (x^2)^k$ et $x^{2k+1} = x \cdot x^{2k}$
- Complexité en nombre de multiplications
  - Lorsque n est paire $C_n = C_{n/2} + 1 = C_{\lfloor \frac{n}{2} \rfloor} + 1$.
  - Si n est impaire (et > 1), $C_n = 1 + C_{n-1} = 1 + C_{(n-1)/2} + 1 = C_{\lfloor \frac{n}{2} \rfloor} + 2$.
- Donc récurrence de la forme $C_n = C_{\lfloor \frac{n}{2} \rfloor} + O(1)$. On étudie toujours la forme la plus simple $C_n = C_{\lfloor \frac{n}{2} \rfloor} + 1$. On a vu que $C_n = \Theta(\log_2 n)$

Exponentiation rapide

```
let rec puissance_rapide x n = match n with
  | 0 -> 1
  | n when n mod 2 = 0 -> puissance_rapide (x*x) (n/2)
  | _ -> x*puissance_rapide x (n-1) ;;
```

- Principe $x^{2k} = (x^2)^k$ et $x^{2k+1} = x \cdot x^{2k}$
- Complexité en nombre de multiplications
  - Lorsque n est paire $C_n = C_{n/2} + 1 = C_{\lfloor \frac{n}{2} \rfloor} + 1$.
  - Si n est impaire (et > 1), $C_n = 1 + C_{n-1} = 1 + C_{(n-1)/2} + 1 = C_{\lfloor \frac{n}{2} \rfloor} + 2$.
- Donc récurrence de la forme $C_n = C_{\lfloor \frac{n}{2} \rfloor} + O(1)$. On étudie toujours la forme la plus simple $C_n = C_{\lfloor \frac{n}{2} \rfloor} + 1$. On a vu que $C_n = \Theta(\log_2 n)$
- Si la taille des entiers est non bornée, on exprime la complexité de l'algorithme en fonction du nombre de bits de l'écriture binaire de n : $\lfloor \log_2 n \rfloor + 1 = \Theta(\log_2 n)$. Donc complexité linéaire en le nombre de bits de l'exposant.

Tri fusion

Division

On veut trier une liste. On sépare l'entrée en deux listes/deux tableaux (approximativement) de même longueur, qu'on trie séparément, puis on fusionne les deux résultats en comparant à chaque fois les plus petits éléments.

```
let rec separer l = match l with
  | [] -> ([],[])
  | [e] -> ([e],[])
  | a::b::r -> let (m1,m2)=separer r in (a::m1,b::m2) ;;
```

Tri fusion

Division

On veut trier une liste. On sépare l'entrée en deux listes/deux tableaux (approximativement) de même longueur, qu'on trie séparément, puis on fusionne les deux résultats en comparant à chaque fois les plus petits éléments.

```
let rec separer l = match l with
  | [] -> ([],[])
  | [e] -> ([e],[])
  | a::b::r -> let (m1,m2)=separer r in (a::m1,b::m2) ;;
```

- Complexité temporelle de la forme $C_n = C_{n-2} + 1$. Ainsi, pour $n = 2k$:

$$C_{2k} = C_{2k-2} + 1 = C_{2k-4} + 1 + 1 = \cdots = C_0 + k$$

On trouve de même $C_{2k+1} = C_1 + k$ pour $n = 2k + 1$.

Tri fusion

Division

On veut trier une liste. On sépare l'entrée en deux listes/deux tableaux (approximativement) de même longueur, qu'on trie séparément, puis on fusionne les deux résultats en comparant à chaque fois les plus petits éléments.

```
let rec separer l = match l with
  | [] -> ([],[])
  | [e] -> ([e],[])
  | a::b::r -> let (m1,m2)=separer r in (a::m1,b::m2) ;;
```

- Complexité temporelle de la forme $C_n = C_{n-2} + 1$. Ainsi, pour $n = 2k$:

$$C_{2k} = C_{2k-2} + 1 = C_{2k-4} + 1 + 1 = \cdots = C_0 + k$$

  On trouve de même $C_{2k+1} = C_1 + k$ pour $n = 2k + 1$.
- Comme dans les 2 hypothèses de parité et d'imparité, $k = \lfloor \frac{n}{2} \rfloor$, et puisque $n = \Theta(\lfloor \frac{n}{2} \rfloor)$, on obtient $C_n = \Theta(n)$

Tri fusion

fusion

On fusionne deux listes triées (par ordre croissant) en ajoutant prioritairement le plus petit élément des deux listes.

```
1  let rec fusion l1 l2 = match (l1,l2) with
2    | (l,[]) | ([],l) -> l
3    | (a::r,b::s) -> if a<=b
4      then a::(fusion r l2)
5      else b::(fusion l1 s);;
```

Tri fusion

fusion

On fusionne deux listes triées (par ordre croissant) en ajoutant prioritairement le plus petit élément des deux listes.

```
let rec fusion l1 l2 = match (l1,l2) with
  | (l,[]) | ([],l) -> l
  | (a::r,b::s) -> if a<=b
    then a::(fusion r l2)
    else b::(fusion l1 s);;
```

- Hors appels récursifs internes, les opérations sont en $O(1)$ et en nombre borné.

Tri fusion

fusion

On fusionne deux listes triées (par ordre croissant) en ajoutant prioritairement le plus petit élément des deux listes.

```
let rec fusion l1 l2 = match (l1,l2) with
  | (l,[]) | ([],l) -> l
  | (a::r,b::s) -> if a<=b
    then a::(fusion r l2)
    else b::(fusion l1 s);;
```

- Hors appels récursifs internes, les opérations sont en $O(1)$ et en nombre borné.
- La complexité dépend de la façon dont sont réparties les données dans les listes. Meilleur cas en $O(\min(|\ell_1|, |\ell_2|))$: lorsque tous les éléments de la liste la plus courte sont plus petits que ceux de la plus longue.

Tri fusion

fusion

On fusionne deux listes triées (par ordre croissant) en ajoutant prioritairement le plus petit élément des deux listes.

```
let rec fusion l1 l2 = match (l1,l2) with
  | (l,[]) | ([],l) -> l
  | (a::r,b::s) -> if a<=b
    then a::(fusion r l2)
    else b::(fusion l1 s);;
```

- Hors appels récursifs internes, les opérations sont en $O(1)$ et en nombre borné.
- La complexité dépend de la façon dont sont réparties les données dans les listes. Meilleur cas en $O(\min(|\ell_1|, |\ell_2|))$: lorsque tous les éléments de la liste la plus courte sont plus petits que ceux de la plus longue.
- Pour éviter un raisonnement qui tiendrait compte de la répartition des données, on peut étudier la complexité d'une fonction manifestement plus coûteuse : `fusion_couteuse` du transparent suivant.

Tri fusion

fusion

- La fonction suivante réalise aussi la fusion. Elle est plus coûteuse que la précédente mais plus facile à étudier :

```
let rec fusion_couteuse l1 l2 = match (l1,l2) with
| ([],[]) -> []
| (a::r,[]) | ([],a::r) -> a::(fusion_couteuse [] r)
| (a::r,b::s) -> if a<=b
  then a::(fusion_couteuse r l2)
  else b::(fusion_couteuse l1 s);;
```

Tri fusion

fusion

- La fonction suivante réalise aussi la fusion. Elle est plus coûteuse que la précédente mais plus facile à étudier :

```
let rec fusion_couteuse l1 l2 = match (l1,l2) with
| ([],[]) -> []
| (a::r,[]) | ([],a::r) -> a::(fusion_couteuse [] r)
| (a::r,b::s) -> if a<=b
  then a::(fusion_couteuse r l2)
  else b::(fusion_couteuse l1 s);;
```

- La complexité dépend seulement de la somme des tailles des listes. En notant n, m les tailles de `l1,l2` on obtient la relation suivante $C(n+m) = C(n+m-1) + O(1)$ et $C(0) = 1$.

Tri fusion

fusion

- La fonction suivante réalise aussi la fusion. Elle est plus coûteuse que la précédente mais plus facile à étudier :

```
let rec fusion_couteuse l1 l2 = match (l1,l2) with
| ([],[]) -> []
| (a::r,[]) | ([],a::r) -> a::(fusion_couteuse [] r)
| (a::r,b::s) -> if a<=b
  then a::(fusion_couteuse r l2)
  else b::(fusion_couteuse l1 s);;
```

- La complexité dépend seulement de la somme des tailles des listes. En notant n, m les tailles de `l1,l2` on obtient la relation suivante $C(n+m) = C(n+m-1) + O(1)$ et $C(0) = 1$.
- On étudie donc $C(n+m) = C(n+m-1) + 1$. Il s'agit d'une suite arithmétique. On a donc $C(n+m) = \Theta(n+m)$ (ordre de grandeur).

Tri fusion

fusion

- La fonction suivante réalise aussi la fusion. Elle est plus coûteuse que la précédente mais plus facile à étudier :

```
let rec fusion_couteuse l1 l2 = match (l1,l2) with
| ([],[]) -> []
| (a::r,[]) | ([],a::r) -> a::(fusion_couteuse [] r)
| (a::r,b::s) -> if a<=b
  then a::(fusion_couteuse r l2)
  else b::(fusion_couteuse l1 s);;
```

- La complexité dépend seulement de la somme des tailles des listes. En notant n, m les tailles de `l1,l2` on obtient la relation suivante $C(n+m) = C(n+m-1) + O(1)$ et $C(0) = 1$.
- On étudie donc $C(n+m) = C(n+m-1) + 1$. Il s'agit d'une suite arithmétique. On a donc $C(n+m) = \Theta(n+m)$ (ordre de grandeur).
- Donc la complexité de `fusion`, toujours meilleure, est aussi en $O(n+m)$ (majoration, ça nous suffit pour la suite).

Tri fusion

Tri

Code du tri fusion

```
let rec tri_fusion l = match l with
    | [] -> []
    | [e] -> [e]
    | l -> let (m1,m2) = separer l in
    fusion (tri_fusion m1) (tri_fusion m2);;
```

Tri fusion

Complexité du tri

- On note n la longueur de `l`. La division est en $\Theta(n)$. La fusion s'applique aux deux moitiés triées ; elle a ici un coût en $O(n)$ (et même en $\Theta(n)$ -en exo-). Bref : séparation + fusion en $\Theta(n)$.

Tri fusion

Complexité du tri

- On note n la longueur de `l`. La division est en $\Theta(n)$. La fusion s'applique aux deux moitiés triées ; elle a ici un coût en $O(n)$ (et même en $\Theta(n)$ -en exo-). Bref : séparation + fusion en $\Theta(n)$.
- La complexité obéit à une relation diviser-pour-régner de la forme

$$C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + \Theta(n) \text{ ; simplifié en } C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + n$$

Tri fusion

Complexité du tri

- On note n la longueur de `l`. La division est en $\Theta(n)$. La fusion s'applique aux deux moitiés triées ; elle a ici un coût en $O(n)$ (et même en $\Theta(n)$ -en exo-). Bref : séparation + fusion en $\Theta(n)$.
- La complexité obéit à une relation diviser-pour-régner de la forme

$$C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + \Theta(n) \text{ ; simplifié en } C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + n$$

- Si $n = 2^k$ alors $\lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil = 2^{k-1}$ (et $k = \log_2 n$) :

$$C_{2^k} = 2^k + 2C_{2^{k-1}} = 2 \cdot 2^k + 2^2 C_{2^{k-2}} = \cdots = k2^k + 2^k \cdot C_{2^{k-k}} = \Theta(n \log_2 n)$$

Tri fusion

Complexité du tri

- On note n la longueur de `l`. La division est en $\Theta(n)$. La fusion s'applique aux deux moitiés triées ; elle a ici un coût en $O(n)$ (et même en $\Theta(n)$ -en exo-). Bref : séparation + fusion en $\Theta(n)$.
- La complexité obéit à une relation diviser-pour-régner de la forme

$$C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + \Theta(n) \text{ ; simplifié en } C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + n$$

- Si $n = 2^k$ alors $\lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil = 2^{k-1}$ (et $k = \log_2 n$) :

$$C_{2^k} = 2^k + 2C_{2^{k-1}} = 2 \cdot 2^k + 2^2 C_{2^{k-2}} = \cdots = k2^k + 2^k \cdot C_{2^{k-k}} = \Theta(n \log_2 n)$$

- En général, $2^{\lfloor \log_2 n \rfloor} \leq n < 2^{\lfloor \log_2 n \rfloor + 1}$. Comme la complexité est $\uparrow$:

$$C_n \leq C_{2^{\lfloor \log_2 n \rfloor + 1}} = O\left((\overbrace{\lfloor \log_2 n \rfloor + 1}^{O(\log_2 n)}) \underbrace{2^{\lfloor \log_2 n \rfloor + 1}}_{O(n)} \right) = O(n \log n)$$

Idem pour une minoration. Conclusion : $C_n = \Theta(n \log_2 n)$

Tri fusion impératif

- Vocabulaire : le tri d'un tableau est dit *en place* si l'algorithme travaille directement sur le tableau en entrée et n'utilise pas de tableau auxiliaire.
  Un tri est *stable* s'il conserve les positions relatives des éléments de même valeur.

Tri fusion impératif

- Vocabulaire : le tri d'un tableau est dit *en place* si l'algorithme travaille directement sur le tableau en entrée et n'utilise pas de tableau auxiliaire.
  Un tri est *stable* s'il conserve les positions relatives des éléments de même valeur.
- Il existe une version du tri fusion pour les tableaux.
  Une version délicate (mais possible) existe pour un maintien *en place* mais non *stable*.

Tri fusion impératif

- Vocabulaire : le tri d'un tableau est dit *en place* si l'algorithme travaille directement sur le tableau en entrée et n'utilise pas de tableau auxiliaire.
  Un tri est *stable* s'il conserve les positions relatives des éléments de même valeur.
- Il existe une version du tri fusion pour les tableaux.
  Une version délicate (mais possible) existe pour un maintien *en place* mais non *stable*.
- Le tri fusion est en $O(n \log n)$ au pire, au meilleur et en moyenne.

Tri fusion impératif

- Vocabulaire : le tri d'un tableau est dit *en place* si l'algorithme travaille directement sur le tableau en entrée et n'utilise pas de tableau auxiliaire.
  Un tri est *stable* s'il conserve les positions relatives des éléments de même valeur.
- Il existe une version du tri fusion pour les tableaux.
  Une version délicate (mais possible) existe pour un maintien *en place* mais non *stable*.
- Le tri fusion est en $O(n \log n)$ au pire, au meilleur et en moyenne.
- On ne peut pas faire mieux en moyenne (voir cours sur les arbres) pour un tri *par comparaison* (algorithmes de tri procédant par comparaisons successives entre éléments).