Cours 4
Diviser pour Régner

L2 Informatique

Université de Montpellier

Diviser pour régner

- Méthode algorithmique utilisée pour obtenir une meilleur complexité que des algorithmiques plus 'naïfs' (...).
- Le principe est de diviser le problème en des sous-problèmes et de recombiner les solutions obtenues ('régner').
- La base de ce type d'algorithmes est la récursivité.

1. Qlqs rappels sur la récursivité
2. Premier exemple : tri fusion
3. Qu'est-ce que « diviser pour régner » ?
4. Deuxième exemple : multiplication d'entiers
5. Exemple spécial : Calcul de rang

1. Qlqs rappels sur la récursivité

2. Premier exemple : tri fusion

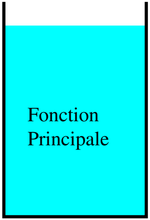
3. Qu'est-ce que « diviser pour régner » ?

4. Deuxième exemple : multiplication d'entiers

5. Exemple spécial : Calcul de rang

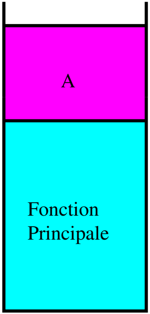
Appel de fonctions en mémoire

- La pile des appels de fonction en mémoire.
- Au début du lancement de tout programme, la `fonction principale` du programme est chargé en mémoire, dans cette pile.



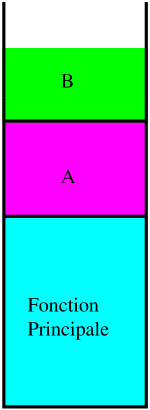
Appel de fonctions en mémoire

- Appel d'une fonction `A` par la fonction `principale` :
- La fonction `principale` se met en attente, et la fonction `A` est chargée dans la pile en mémoire.



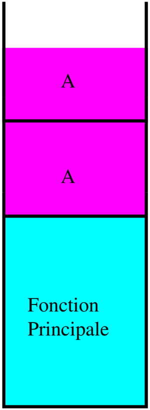
Appel de fonctions en mémoire

- Appel d'une fonction `B` par la fonction `A` :
- La fonction `A` se met en attente, et la fonction `B` est chargée dans la pile en mémoire.



Appel de fonctions en mémoire

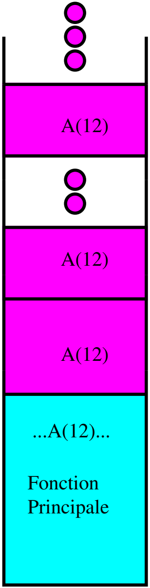
- La récursivité : La fonction `A` s'appelle elle-même (de la même manière qu'elle appelait la fonction `B`...)



Appel de fonctions en mémoire

- La récursivité : Attention aux appels récursifs à l'infini...
- Exemple :

```
def A(n):
if n==0:
        return 0
 return A(n)
```

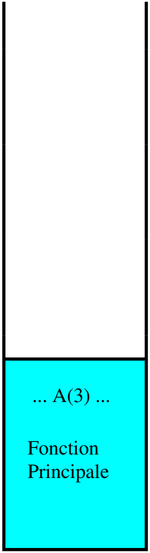


Appel de fonctions en mémoire

- Il faut assurer un nombre fini d'appels récursifs.
- Souvent, un paramètre décroît et un cas de base est prévu.
- Exemple :

```
def A(n):
if n==0:
       return 0
 return A(n-1)+1
```

- Le programme principal appelle `A(3)` et se met en attente.

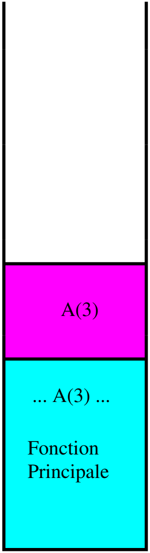


Appel de fonctions en mémoire

- Il faut assurer un nombre fini d'appels récursifs.
- Souvent, un paramètre décroît et un cas de base est prévu.
- Exemple :

```
def A(n):
if n==0:
        return 0
 return A(n-1)+1
```

- `A` se charge dans la pile avec 3 pour paramètre, appelle `A(2)` et se met en attente.

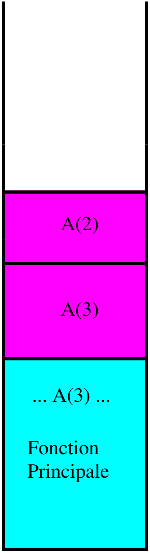


Appel de fonctions en mémoire

- Il faut assurer un nombre fini d'appels récursifs.
- Souvent, un paramètre décroît et un cas de base est prévu.
- Exemple :

```
def A(n):
if n==0:
        return 0
 return A(n-1)+1
```

- `A` se charge de nouveau dans la pile avec 2 pour paramètre, appelle `A(1)` et se met en attente.

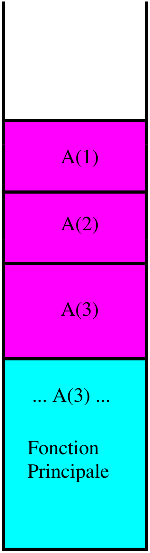


Appel de fonctions en mémoire

- Il faut assurer un nombre fini d'appels récursifs.
- Souvent, un paramètre décroît et un cas de base est prévu.
- Exemple :

```
def A(n):
if n==0:
        return 0
 return A(n-1)+1
```

- `A` se charge encore dans la pile avec 1 pour paramètre, appelle `A(0)` et se met en attente.

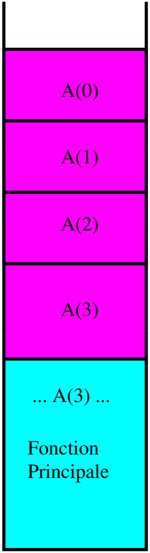


Appel de fonctions en mémoire

- Il faut assurer un nombre fini d'appels récursifs.
- Souvent, un paramètre décroît et un cas de base est prévu.
- Exemple :

```
def A(n):
if n==0:
       return 0
 return A(n-1)+1
```

- `A` se charge une dernière fois dans la pile avec 0 pour paramètre, puis retourne 0 à `A(1)`

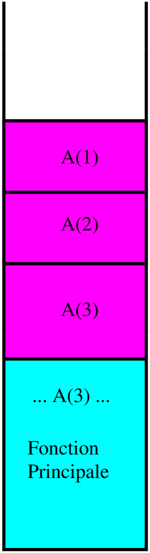


Appel de fonctions en mémoire

- Il faut assurer un nombre fini d'appels récursifs.
- Souvent, un paramètre décroît et un cas de base est prévu.
- Exemple :

```
def A(n):
if n==0:
        return 0
 return A(n-1)+1
```

- `A(1)` peut finir son calcul et renvoie 1 à `A(2)`

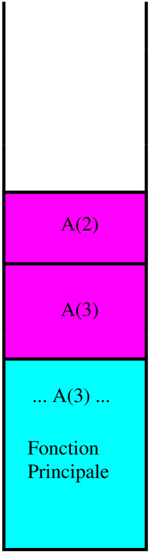


Appel de fonctions en mémoire

- Il faut assurer un nombre fini d'appels récursifs.
- Souvent, un paramètre décroît et un cas de base est prévu.
- Exemple :

```
def A(n):
if n==0:
       return 0
 return A(n-1)+1
```

- `A(2)` peut finir son calcul et renvoie 2 à `A(3)`

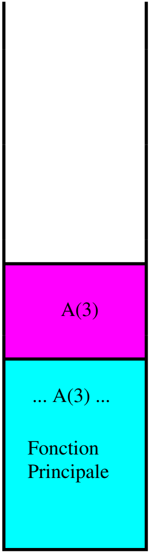


Appel de fonctions en mémoire

- Il faut assurer un nombre fini d'appels récursifs.
- Souvent, un paramètre décroît et un cas de base est prévu.
- Exemple :

```
def A(n):
if n==0:
       return 0
 return A(n-1)+1
```

- `A(3)` peut finir son calcul et renvoie 3 à la fonction principale

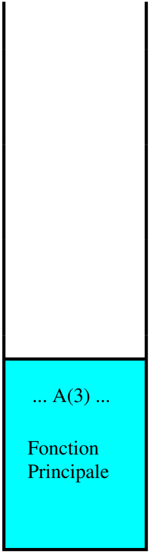


Appel de fonctions en mémoire

- Il faut assurer un nombre fini d'appels récursifs.
- Souvent, un paramètre décroît et un cas de base est prévu.
- Exemple :

```
def A(n):
if n==0:
       return 0
 return A(n-1)+1
```

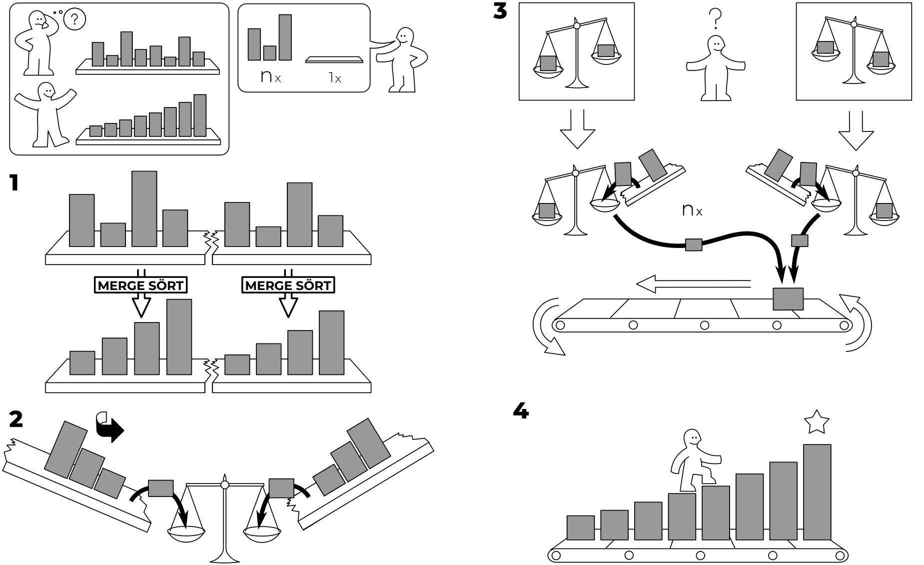
- La fonction principale a obtenu la valeur de `A(3)`, elle peut continuer sa petite vie de processus...



1. Qlqs rappels sur la récursivité
2. Premier exemple : tri fusion
3. Qu'est-ce que « diviser pour régner » ?
4. Deuxième exemple : multiplication d'entiers
5. Exemple spécial : Calcul de rang

MERGE SÖRT

idea-instructions.com/merge-sort/



Algorithme du TriFusion

Algorithme : TriFusion(T)

$n \leftarrow$ taille(T)
si $n = 1$ alors
    Retourner T
sinon
    $T_1 \leftarrow$ TriFusion($T[0, \lfloor n/2 \rfloor - 1]$)
    $T_2 \leftarrow$ TriFusion($T[\lfloor n/2 \rfloor, n-1]$)
    Retourner Fusion(T_1, T_2)

Algorithme du TriFusion

Algorithme : TriFusion(T)

$n \leftarrow$ taille(T)
si $n = 1$ alors
    Retourner T
sinon
    $T_1 \leftarrow$ TriFusion($T[0, \lfloor n/2 \rfloor - 1]$)
    $T_2 \leftarrow$ TriFusion($T[\lfloor n/2 \rfloor, n-1]$)
    Retourner Fusion(T_1, T_2)

Lemme

Soit $T(n)$ la complexité de TriFusion et $F(n)$ la complexité de Fusion. Alors

$$T(n) = \begin{cases} O(1) & \text{si } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + F(n) + O(1) & \text{sinon} \end{cases}$$

Algorithme de Fusion

Algorithme : Fusion(T_1, T_2)

$n_1 \leftarrow$ taille(T_1) ; $n_2 \leftarrow$ taille(T_2)
$S \leftarrow$ tableau de taille $n_1 + n_2$
$i_1 \leftarrow 0$; $i_2 \leftarrow 0$
pour $i_S = 0$ *à* $n_1 + n_2 - 1$ faire
    si $i_1 \geq n_1$ alors $S[i_S] \leftarrow T_2[i_2]$; $i_2 \leftarrow i_2 + 1$;
    sinon si $i_2 \geq n_2$ alors $S[i_S] \leftarrow T_1[i_1]$; $i_1 \leftarrow i_1 + 1$;
    sinon si $T_1[i_1] < T_2[i_2]$ alors $S[i_S] \leftarrow T_1[i_1]$; $i_1 \leftarrow i_1 + 1$;
    sinon $S[i_S] \leftarrow T_2[i_2]$; $i_2 \leftarrow i_2 + 1$;

retourner S

Algorithme de Fusion

Algorithme : Fusion(T_1, T_2)

$n_1 \leftarrow$ taille(T_1) ; $n_2 \leftarrow$ taille(T_2)
$S \leftarrow$ tableau de taille $n_1 + n_2$
$i_1 \leftarrow 0$; $i_2 \leftarrow 0$
pour $i_S = 0$ *à* $n_1 + n_2 - 1$ faire
  si $i_1 \geq n_1$ alors $S[i_S] \leftarrow T_2[i_2]$; $i_2 \leftarrow i_2 + 1$;
  sinon si $i_2 \geq n_2$ alors $S[i_S] \leftarrow T_1[i_1]$; $i_1 \leftarrow i_1 + 1$;
  sinon si $T_1[i_1] < T_2[i_2]$ alors $S[i_S] \leftarrow T_1[i_1]$; $i_1 \leftarrow i_1 + 1$;
  sinon $S[i_S] \leftarrow T_2[i_2]$; $i_2 \leftarrow i_2 + 1$;

retourner S

Exemple : tri de $[10, 5, 2, 4, 3, 7, 6, 4]$

Algorithme de Fusion

Algorithme : Fusion(T_1, T_2)

$n_1 \leftarrow$ taille(T_1) ; $n_2 \leftarrow$ taille(T_2)
$S \leftarrow$ tableau de taille $n_1 + n_2$
$i_1 \leftarrow 0$; $i_2 \leftarrow 0$
pour $i_S = 0$ *à* $n_1 + n_2 - 1$ faire
    si $i_1 \geq n_1$ alors $S[i_S] \leftarrow T_2[i_2]$; $i_2 \leftarrow i_2 + 1$;
    sinon si $i_2 \geq n_2$ alors $S[i_S] \leftarrow T_1[i_1]$; $i_1 \leftarrow i_1 + 1$;
    sinon si $T_1[i_1] < T_2[i_2]$ alors $S[i_S] \leftarrow T_1[i_1]$; $i_1 \leftarrow i_1 + 1$;
    sinon $S[i_S] \leftarrow T_2[i_2]$; $i_2 \leftarrow i_2 + 1$;
retourner S

Lemme

La complexité de Fusion est $O(n_1 + n_2)$.

Preuve facile...

Algorithme de Fusion

Algorithme : Fusion(T_1, T_2)

$n_1 \leftarrow$ taille(T_1) ; $n_2 \leftarrow$ taille(T_2)
$S \leftarrow$ tableau de taille $n_1 + n_2$
$i_1 \leftarrow 0$; $i_2 \leftarrow 0$
pour $i_S = 0$ *à* $n_1 + n_2 - 1$ faire
  si $i_1 \geq n_1$ alors $S[i_S] \leftarrow T_2[i_2]$; $i_2 \leftarrow i_2 + 1$;
  sinon si $i_2 \geq n_2$ alors $S[i_S] \leftarrow T_1[i_1]$; $i_1 \leftarrow i_1 + 1$;
  sinon si $T_1[i_1] < T_2[i_2]$ alors $S[i_S] \leftarrow T_1[i_1]$; $i_1 \leftarrow i_1 + 1$;
  sinon $S[i_S] \leftarrow T_2[i_2]$; $i_2 \leftarrow i_2 + 1$;
retourner S

Lemme

Si T_1 et T_2 sont deux tableaux triés (par ordre croissant), Fusion(T_1, T_2) renvoie un tableau trié contenant l'union des éléments de T_1 et T_2.

Algorithme de Fusion

Algorithme : Fusion(T_1, T_2)

$n_1 \leftarrow$ taille(T_1) ; $n_2 \leftarrow$ taille(T_2)
$S \leftarrow$ tableau de taille $n_1 + n_2$
$i_1 \leftarrow 0$; $i_2 \leftarrow 0$
pour $i_S = 0$ *à* $n_1 + n_2 - 1$ faire
  si $i_1 \geq n_1$ alors $S[i_S] \leftarrow T_2[i_2]$; $i_2 \leftarrow i_2 + 1$;
  sinon si $i_2 \geq n_2$ alors $S[i_S] \leftarrow T_1[i_1]$; $i_1 \leftarrow i_1 + 1$;
  sinon si $T_1[i_1] < T_2[i_2]$ alors $S[i_S] \leftarrow T_1[i_1]$; $i_1 \leftarrow i_1 + 1$;
  sinon $S[i_S] \leftarrow T_2[i_2]$; $i_2 \leftarrow i_2 + 1$;

retourner S

Preuve rapide

$\mathscr{P}_{i_S}$: à l'entrée de l'itération i_S de la boucle pour,

1. $S[0, i_S - 1]$ contient les i_S plus petits éléments de $T_1 \cup T_2$ en ordre croissant
2. i_1 est l'indice du plus petit élément de T_1 non présent dans S
3. i_2 est l'indice du plus petit élément de T_2 non présent dans S

Retour sur le TriFusion

Théorème

L'algorithme TriFusion trie tout tableau de taille n en temps $O(n \log n)$.

Algorithme : TriFusion(T)

$n \leftarrow$ taille(T)
si $n = 1$ alors
  Retourner T
sinon
  $T_1 \leftarrow$ TriFusion($T[0, \lfloor n/2 \rfloor - 1]$)
  $T_2 \leftarrow$ TriFusion($T[\lfloor n/2 \rfloor, n-1]$)
  Retourner Fusion(T_1, T_2)

Retour sur le TriFusion

Théorème

L'algorithme TriFusion trie tout tableau de taille n en temps $O(n \log n)$.

Algorithme : TriFusion(T)

$n \leftarrow$ taille(T)
si $n = 1$ alors
  Retourner T
sinon
  $T_1 \leftarrow$ TriFusion($T[0, \lfloor n/2 \rfloor - 1]$)
  $T_2 \leftarrow$ TriFusion($T[\lfloor n/2 \rfloor, n-1]$)
  Retourner Fusion(T_1, T_2)

Preuve de correction par récurrence (facile !)

- Si $n = 1$, OK
- Si $n > 1$, $\lfloor n/2 \rfloor \leq \lceil n/2 \rceil < \mathrm{n}$, donc T_1 et T_2 triés après appels récursifs. La correction de Fusion suffit à conclure.

Retour sur le TriFusion

Théorème
L'algorithme TriFusion trie tout tableau de taille n en temps $O(n \log n)$.

Algorithme : TriFusion(T)

$n \leftarrow$ taille(T)
si $n = 1$ alors
  Retourner T
sinon
  $T_1 \leftarrow$ TriFusion($T[0, \lfloor n/2 \rfloor - 1]$)
  $T_2 \leftarrow$ TriFusion($T[\lfloor n/2 \rfloor, n-1]$)
  Retourner Fusion(T_1, T_2)

Preuve de correction par récurrence (facile !)

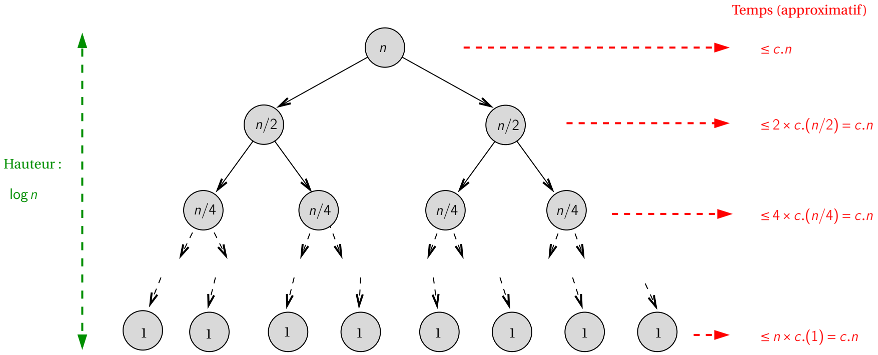
- Si $n = 1$, OK
- Si $n > 1$, $\lfloor n/2 \rfloor \leq \lceil n/2 \rceil <$ n, donc T_1 et T_2 triés après appels récursifs. La correction de Fusion suffit à conclure.

Preuve de complexité

- Équation de récurrence : $t(n) \leq 2t(\lceil n/2 \rceil) + O(n)$
- Arbre de récursion $\rightsquigarrow$ estimation du temps de calcul
- Preuve par récurrence de l'estimation

TRIFUSION : idée de la complexité, arbre de récursion

- On note c une constante majorant le nombre d'opérations élémentaires apparaissant dans TRIFUSION et FUSION (hors boucle et appels récursifs).



- En tout, l'algo a une complexité approximative de $c.n.\log n$, c'est-à-dire en $O(n \log n)$

TriFusion : preuve de la complexité, par récurrence

- On note toujours c une constante majorant le nombre d'opérations élémentaires apparaissant dans TriFusion et Fusion (hors boucle et appels récursifs).
- On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TriFusion appelé sur un tableau de taille n.

TRIFUSION : preuve de la complexité, par récurrence

- On note toujours c une constante majorant le nombre d'opérations élémentaires apparaissant dans TRIFUSION et FUSION (hors boucle et appels récursifs).
- On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TRIFUSION appelé sur un tableau de taille n.
- Etape 1 : Invariant : $\mathscr{P}_n =' t(n) \leq c.n.\log(2n)'$
  On montre que $\mathscr{P}_n$ est vraie pour tout $n = 2^k$ et $k \geq 0$

TriFusion : preuve de la complexité, par récurrence

- On note toujours c une constante majorant le nombre d'opérations élémentaires apparaissant dans TriFusion et Fusion (hors boucle et appels récursifs).
- On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TriFusion appelé sur un tableau de taille n.
- Etape 1 : Invariant : $\mathscr{P}_n =' t(n) \leq c.n.\log(2n)'$
  On montre que $\mathscr{P}_n$ est vraie pour tout $n = 2^k$ et $k \geq 0$
  - $\mathscr{P}_1 (= \mathscr{P}_{2^0})$ est vraie.
  - Supposons $\mathscr{P}_n$ vraie pour $n = 2^{k-1}$, alors pour $n = 2^k$ on a :
    - $t(n) = t(2^k) \leq c.2^k + 2.t(2^{k-1})$ (équation)

TriFusion : preuve de la complexité, par récurrence

- On note toujours c une constante majorant le nombre d'opérations élémentaires apparaissant dans TriFusion et Fusion (hors boucle et appels récursifs).
- On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TriFusion appelé sur un tableau de taille n.
- Etape 1 : Invariant : $\mathscr{P}_n =' t(n) \leq c.n.\log(2n)'$

  On montre que $\mathscr{P}_n$ est vraie pour tout $n = 2^k$ et $k \geq 0$
  - $\mathscr{P}_1 (= \mathscr{P}_{2^0})$ est vraie.
  - Supposons $\mathscr{P}_n$ vraie pour $n = 2^{k-1}$, alors pour $n = 2^k$ on a :
    - $t(n) = t(2^k) \leq c.2^k + 2.t(2^{k-1})$ (équation)
    - $t(n) \leq c.2^k + 2.(c.2^{k-1}\log(2.2^{k-1}))$ (hyp. de réc.)

TriFusion : preuve de la complexité, par récurrence

- On note toujours c une constante majorant le nombre d'opérations élémentaires apparaissant dans TriFusion et Fusion (hors boucle et appels récursifs).
- On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TriFusion appelé sur un tableau de taille n.
- Etape 1 : Invariant : $\mathscr{P}_n =' t(n) \leq c.n.\log(2n)'$
  On montre que $\mathscr{P}_n$ est vraie pour tout $n = 2^k$ et $k \geq 0$
  - $\mathscr{P}_1 (= \mathscr{P}_{2^0})$ est vraie.
  - Supposons $\mathscr{P}_n$ vraie pour $n = 2^{k-1}$, alors pour $n = 2^k$ on a :
    - $t(n) = t(2^k) \leq c.2^k + 2.t(2^{k-1})$ (équation)
    - $t(n) \leq c.2^k + 2.(c.2^{k-1}\log(2.2^{k-1}))$ (hyp. de réc.)
    - $t(n) \leq c.2^k + c.2^k(\log(2.2^k) - 1)$
      $t(n) \leq c.2^k.\log 2.2^k = c.n.\log(2n)$

TriFusion : preuve de la complexité, par récurrence

- On note toujours c une constante majorant le nombre d'opérations élémentaires apparaissant dans TriFusion et Fusion (hors boucle et appels récursifs).
- On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TriFusion appelé sur un tableau de taille n.
- Etape 1 : Invariant : $\mathscr{P}_n =' t(n) \leq c.n.\log(2n)'$
  On montre que $\mathscr{P}_n$ est vraie pour tout $n = 2^k$ et $k \geq 0$
- Etape 2 : Invariant : $\mathscr{P}_n =' t(n) \leq 4.c.n.\log(2n)'$
  On montre que $\mathscr{P}_n$ est vraie pour tout $n \geq 1$

TriFusion : preuve de la complexité, par récurrence

- On note toujours c une constante majorant le nombre d'opérations élémentaires apparaissant dans TriFusion et Fusion (hors boucle et appels récursifs).
- On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TriFusion appelé sur un tableau de taille n.
- Etape 1 : Invariant : $\mathscr{P}_n =' t(n) \leq c.n.\log(2n)'$
  On montre que $\mathscr{P}_n$ est vraie pour tout $n = 2^k$ et $k \geq 0$
- Etape 2 : Invariant : $\mathscr{P}_n =' t(n) \leq 4.c.n.\log(2n)'$
  On montre que $\mathscr{P}_n$ est vraie pour tout $n \geq 1$
  - Pour n quelconque, il existe k tel que $2^{k-1} < n \leq 2^k$
  - On a $t(n) \leq t(2^k) \leq c.2^k.\log(2.2^k)$ par l'étape 1

TriFusion : preuve de la complexité, par récurrence

- On note toujours c une constante majorant le nombre d'opérations élémentaires apparaissant dans TriFusion et Fusion (hors boucle et appels récursifs).
- On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TriFusion appelé sur un tableau de taille n.
- Etape 1 : Invariant : $\mathscr{P}_n =' t(n) \leq c.n.\log(2n)'$
  On montre que $\mathscr{P}_n$ est vraie pour tout $n = 2^k$ et $k \geq 0$
- Etape 2 : Invariant : $\mathscr{P}_n =' t(n) \leq 4.c.n.\log(2n)'$
  On montre que $\mathscr{P}_n$ est vraie pour tout $n \geq 1$
  - Pour n quelconque, il existe k tel que $2^{k-1} < n \leq 2^k$
  - On a $t(n) \leq t(2^k) \leq c.2^k.\log(2.2^k)$ par l'étape 1
  - On a aussi $2^k \leq 2.n$, donc $t(n) \leq 2c.n.\log(4.n) = 2c.n.\log(2.(2n)) = 2c.n.(log(2n)+1) \leq 4c.n.\log(2n)$ $\blacksquare$

TRIFUSION : preuve de la complexité, par récurrence

- On note toujours c une constante majorant le nombre d'opérations élémentaires apparaissant dans TRIFUSION et FUSION (hors boucle et appels récursifs).
- On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TRIFUSION appelé sur un tableau de taille n.
- Etape 1 : Invariant : $\mathscr{P}_n =' t(n) \leq c.n.\log(2n)'$
  On montre que $\mathscr{P}_n$ est vraie pour tout $n = 2^k$ et $k \geq 0$
- Etape 2 : Invariant : $\mathscr{P}_n =' t(n) \leq 4.c.n.\log(2n)'$
  On montre que $\mathscr{P}_n$ est vraie pour tout $n \geq 1$
  - Pour n quelconque, il existe k tel que $2^{k-1} < n \leq 2^k$
  - On a $t(n) \leq t(2^k) \leq c.2^k.\log(2.2^k)$ par l'étape 1
  - On a aussi $2^k \leq 2.n$, donc $t(n) \leq 2c.n.\log(4.n) = 2c.n.\log(2.(2n)) = 2c.n.(log(2n)+1) \leq 4c.n.\log(2n)$ ∎
- Arg... C'est un peu fastidieux...

1. Qlqs rappels sur la récursivité

2. Premier exemple : tri fusion

3. Qu'est-ce que « diviser pour régner » ?

4. Deuxième exemple : multiplication d'entiers

5. Exemple spécial : Calcul de rang

La stratégie « diviser pour régner »

1. Diviser le problème en sous-problèmes
2. Résoudre récursivement ces sous-problèmes
3. Combiner les solutions pour reconstruire la solution du problème original.

La stratégie « diviser pour régner »

1. Diviser le problème en sous-problèmes
2. Résoudre récursivement ces sous-problèmes
3. Combiner les solutions pour reconstruire la solution du problème original.

- Stratégie principalement utilisée pour obtenir de meilleures complexités que celles données par un algorithme moins évolué.

La stratégie « diviser pour régner »

1. Diviser le problème en sous-problèmes
2. Résoudre récursivement ces sous-problèmes
3. Combiner les solutions pour reconstruire la solution du problème original.

- Stratégie principalement utilisée pour obtenir de meilleures complexités que celles données par un algorithme moins évolué.
- Exemple : la recherche dichotomique

La stratégie « diviser pour régner »

1. Diviser le problème en sous-problèmes
2. Résoudre récursivement ces sous-problèmes
3. Combiner les solutions pour reconstruire la solution du problème original.

- Stratégie principalement utilisée pour obtenir de meilleures complexités que celles données par un algorithme moins évolué.
- Exemple : la recherche dichotomique

Exemple du tri fusion

1. Diviser le tableau en 2 sous-tableaux de tailles environ égales
2. Trier récursivement chaque sous-tableau
3. Fusionner les sous-tableaux triés

Analyse d'un algorithme « diviser pour régner »

Récurrence(s) sur la taille du problème

Analyse d'un algorithme « diviser pour régner »

Récurrence(s) sur la taille du problème

- Preuve de correction

Analyse d'un algorithme « diviser pour régner »

Récurrence(s) sur la taille du problème

- Preuve de correction
- Complexité :
  1. Établir l'équation de récurrence
  2. Estimer le résultat (arbre de récursion, déroulement de la récurrence, habitude, ...)
  3. Preuve par récurrence

Analyse d'un algorithme « diviser pour régner »

Récurrence(s) sur la taille du problème

- Preuve de correction
- Complexité :
  1. Établir l'équation de récurrence
  2. Estimer le résultat (arbre de récursion, déroulement de la récurrence, habitude, ...)
  3. Preuve par récurrence

  ou

  2-3. Utiliser le « *master theorem* » !

Une version du « *master theorem* »

Théorème

S'il existe trois entiers $a \geq 0$, $b > 1$, $d \geq 0$ *et* $n_0 > 0$ *tels que pour tout* $n \geq n_0$,

$$T(n) \leq aT(\lceil n/b \rceil) + O(n^d)$$

Alors

$$T(n) = \begin{cases} O(n^d) & \textit{si } b^d > a \\ O(n^d \log n) & \textit{si } b^d = a \\ O(n^{\frac{\log a}{\log b}}) & \textit{si } b^d < a \end{cases}$$

Une version du « *master theorem* »

Théorème

S'il existe trois entiers $a \geq 0$, $b > 1$, $d \geq 0$ et $n_0 > 0$ tels que pour tout $n \geq n_0$,

$$\mathrm{T}(\mathrm{n}) \leq \mathrm{a}\mathrm{T}(\lceil \mathrm{n}/\mathrm{b} \rceil) + \mathrm{O}(\mathrm{n}^{\mathrm{d}})$$

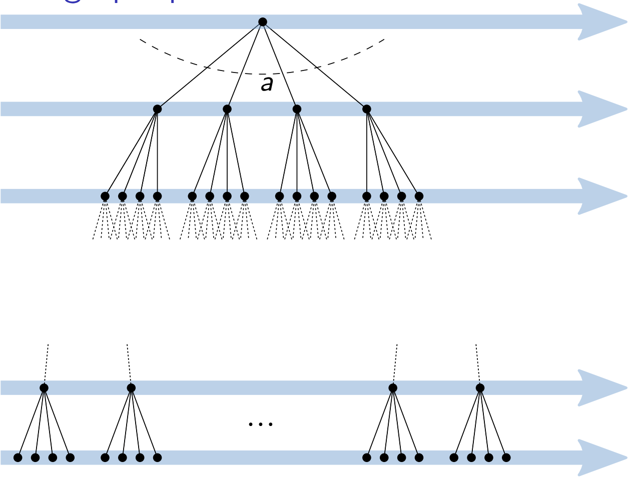
Alors

$$\mathrm{T}(\mathrm{n}) = \begin{cases} \mathrm{O}(\mathrm{n}^{\mathrm{d}}) & \textit{si } \mathrm{b}^{\mathrm{d}} > \mathrm{a} \\ \mathrm{O}(\mathrm{n}^{\mathrm{d}} \log \mathrm{n}) & \textit{si } \mathrm{b}^{\mathrm{d}} = \mathrm{a} \\ \mathrm{O}(\mathrm{n}^{\frac{\log \mathrm{a}}{\log \mathrm{b}}}) & \textit{si } \mathrm{b}^{\mathrm{d}} < \mathrm{a} \end{cases}$$

Exemple du tri fusion

- $T(n) \leq 2T(\lceil n/2 \rceil]) + O(n)$: $a = 2$, $b = 2$, $d = 1$
- $b^d = a \rightsquigarrow T(n) = O(n^d \log n) = O(n \log n)$

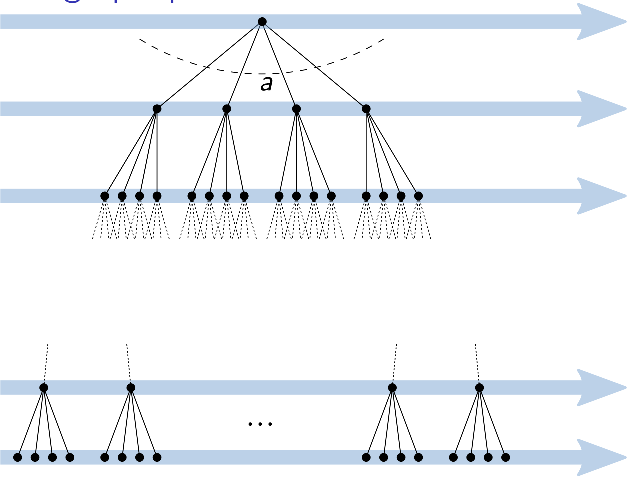
Preuve graphique



$$
\begin{array}{c}
1 \times n^d \\
+ \\
a \times (n/b)^d \\
+ \\
a^2 \times (n/b^2)^d \\
+ \\
\vdots \\
+ \\
a^{\ell-1} \times (n/b^{\ell-1})^d \\
+ \\
a^{\ell} \times (n/b^{\ell})^d
\end{array}
$$

modifié d'après *Algorithms* de Dasgupta, Papadimitriou, Vazirani
inspiré aussi de *Algorithms* de J. Erickson

Preuve graphique



$1 \times n^d$

$+$

$a \times (n/b)^d$

$+$

$a^2 \times (n/b^2)^d$

$+$

$\vdots$

$+$

$a^{\ell-1} \times (n/b^{\ell-1})^d$

$+$

$a^{\ell} \times (n/b^{\ell})^d$

$(\ell = \log_b n)$

$$= \sum_{i=0}^{\ell} a^i \left(\frac{n}{b^i}\right)^d = n^d \sum_{i=0}^{\ell} \left(\frac{a}{b^d}\right)^i = \begin{cases} O(n^d) & \text{si } b^d > a \\ O(n^d \log n) & \text{si } b^d = a \\ O(n^{\log b/\log a}) & \text{si } b^d < a \end{cases}$$

modifié d'après *Algorithms* de Dasgupta, Papadimitriou, Vazirani

inspiré aussi de *Algorithms* de J. Erickson

1. Qlqs rappels sur la récursivité

2. Premier exemple : tri fusion

3. Qu'est-ce que « diviser pour régner » ?

4. Deuxième exemple : multiplication d'entiers

5. Exemple spécial : Calcul de rang

Retour à l'école primaire

⚠ On se place dans le modèle RAM
(en WORD-RAM, le problème n'aurait pas vraiment d'intérêt...)

Retour à l'école primaire

On se place dans le modèle RAM
(en WORD-RAM, le problème n'aurait pas vraiment d'intérêt...)

Multiplication d'entiers

Entrée Deux entiers A et B écrits en base 10

Sortie L'entier $C = A \times B$, en base 10

$$
\begin{array}{lr}
 & 1382 \\
\times & 7634 \\
\hline
 & 5528 \\
+ & 4146 \\
+ & 8292 \\
+ & 9674 \\
\hline
 & 10550188
\end{array}
$$

Retour à l'école primaire

On se place dans le modèle RAM
(en WORD-RAM, le problème n'aurait pas vraiment d'intérêt...)

$$
\begin{array}{rr}
 & 1382 \\
\times & 7634 \\
\hline
 & 5528 \\
+ & 4146 \\
+ & 8292 \\
+ & 9674 \\
\hline
 & 10550188
\end{array}
$$

Multiplication d'entiers

Entrée Deux entiers A et B écrits en base 10

Sortie L'entier $C = A \times B$, en base 10

Question

- *Combien de* multiplications chiffre à chiffre *sont effectuées ?*
- *Combien d'*additions chiffre à chiffre *sont effectuées ?*

Retour à l'école primaire

⚠ On se place dans le modèle RAM
(en WORD-RAM, le problème n'aurait pas vraiment d'intérêt...)

Multiplication d'entiers

Entrée Deux entiers A et B écrits en base 10

Sortie L'entier $C = A \times B$, en base 10

Question

- *Combien de* multiplications chiffre à chiffre *sont effectuées ?*
- *Combien d'*additions chiffre à chiffre *sont effectuées ?*

$\rightsquigarrow$ $O(n^2)$ *multiplications (et additions)*

$$
\begin{array}{lr}
 & 1382 \\
\times & 7634 \\
\hline
 & 5528 \\
+ & 4146 \\
+ & 8292 \\
+ & 9674 \\
\hline
 & 10550188
\end{array}
$$

Retour à l'école primaire

On se place dans le modèle RAM
(en WORD-RAM, le problème n'aurait pas vraiment d'intérêt...)

$$\begin{array}{lr} & 1382 \\ \times & 7634 \\ \hline & 5528 \\ + & 4146 \\ + & 8292 \\ + & 9674 \\ \hline & 10550188 \end{array}$$

Multiplication d'entiers

Entrée Deux entiers A et B écrits en base 10

Sortie L'entier $C = A \times B$, en base 10

Question

- *Combien de* multiplications chiffre à chiffre *sont effectuées ?*
- *Combien d'*additions chiffre à chiffre *sont effectuées ?*

$\rightsquigarrow O(n^2)$ *multiplications (et additions)*

Peut-on faire mieux ?

Première tentative

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

$A = 1382, \; B = 7634$

Première tentative

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$

$B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

$A = 1382, \ B = 7634$
$A_1 = 13, \ A_0 = 82$
$B_1 = 76, \ B_0 = 34$

Première tentative

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$

$B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

Récursion $C_{00} = A_0 \times B_0,\ C_{01} = A_0 \times B_1$

$C_{10} = A_1 \times B_0,\ C_{11} = A_1 \times B_1$

$A = 1382,\ B = 7634$
$A_1 = 13,\ A_0 = 82$
$B_1 = 76,\ B_0 = 34$
$C_{00} = 2788,\ C_{01} = 6232$
$C_{10} = 442,\ C_{11} = 988$

Première tentative

Entrée	$A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$
Diviser	$A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$
	$B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$
Récursion	$C_{00} = A_0 \times B_0,\ C_{01} = A_0 \times B_1$
	$C_{10} = A_1 \times B_0,\ C_{11} = A_1 \times B_1$
Combiner	$C = C_{00} + 10^{\lfloor n/2 \rfloor}(C_{01} + C_{10}) + 10^{2\lfloor n/2 \rfloor} C_{11}$

$$
\begin{aligned}
&A = 1382,\ B = 7634 \\
&A_1 = 13,\ A_0 = 82 \\
&B_1 = 76,\ B_0 = 34 \\
&C_{00} = 2788,\ C_{01} = 6232 \\
&C_{10} = \ \ 442,\ C_{11} = \ \ 988 \\
&C = 2788 + 100 \cdot (6232 \\
&\qquad + 442) + 10000 \cdot 988 \\
&\quad = 10550188
\end{aligned}
$$

Première tentative

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$
$B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

Récursion $C_{00} = A_0 \times B_0,\ C_{01} = A_0 \times B_1$
$C_{10} = A_1 \times B_0,\ C_{11} = A_1 \times B_1$

Combiner $C = C_{00} + 10^{\lfloor n/2 \rfloor}(C_{01} + C_{10}) + 10^{2\lfloor n/2 \rfloor} C_{11}$

Preuve de correction :
$$AB = A_0 B_0 + 10^{\lfloor n/2 \rfloor}(A_0 B_1 + A_1 B_0) + 10^{2\lfloor n/2 \rfloor} A_1 B_1$$

$A = 1382,\ B = 7634$
$A_1 = 13,\ A_0 = 82$
$B_1 = 76,\ B_0 = 34$
$C_{00} = 2788,\ C_{01} = 6232$
$C_{10} = 442,\ C_{11} = 988$
$C = 2788 + 100 \cdot (6232 + 442) + 10000 \cdot 988 = 10550188$

Première tentative

$A = 1382,\ B = 7634$
$A_1 = 13,\ A_0 = 82$
$B_1 = 76,\ B_0 = 34$
$C_{00} = 2788,\ C_{01} = 6232$
$C_{10} = 442,\ C_{11} = 988$
$C = 2788 + 100 \cdot (6232 + 442) + 10000 \cdot 988 = 10550188$

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$
$B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

Récursion $C_{00} = A_0 \times B_0,\ C_{01} = A_0 \times B_1$
$C_{10} = A_1 \times B_0,\ C_{11} = A_1 \times B_1$

Combiner $C = C_{00} + 10^{\lfloor n/2 \rfloor}(C_{01} + C_{10}) + 10^{2\lfloor n/2 \rfloor} C_{11}$

Preuve de correction :
$AB = A_0 B_0 + 10^{\lfloor n/2 \rfloor}(A_0 B_1 + A_1 B_0) + 10^{2\lfloor n/2 \rfloor} A_1 B_1$

Preuve de complexité : $T(n) \leq 4T(\lceil n/2 \rceil) + O(n)$

- $a = 4,\ b = 2,\ d = 1\ :\ b^d < a$
- $T(n) = O(n^{\log a / \log b}) = O(n^{\log 4 / \log 2}) = O(n^2) \ldots$

Idée de Karatsuba[1]

$$\boxed{A_0B_1 + A_1B_0 = A_0B_0 + A_1B_1 - (A_0 - A_1)(B_0 - B_1)}$$

1. Voir : `https://fr.wikipedia.org/wiki/Algorithme_de_Karatsuba`

Idée de Karatsuba [1]

$$\mathrm{A_0B_1 + A_1B_0 = A_0B_0 + A_1B_1 - (A_0 - A_1)(B_0 - B_1)}$$

- A_0B_0 et A_1B_1 sont calculés de toute façon
- un seul produit en plus !

1. Voir : https://fr.wikipedia.org/wiki/Algorithme_de_Karatsuba

Algorithme de Karatsuba (1962)

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

$A = 1382, \; B = 7634$

Algorithme de Karatsuba (1962)

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$

$B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

$A = 1382,\ B = 7634$
$A_1 = 13,\ A_0 = 82$
$B_1 = 76,\ B_0 = 34$

Algorithme de Karatsuba (1962)

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$

$B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

Récursion $C_{00} = A_0 \times B_0,\ C_{11} = A_1 \times B_1$

$D = (A_0 - A_1) \times (B_0 - B_1)$

$A = 1382,\ B = 7634$

$A_1 = 13,\ A_0 = 82$

$B_1 = 76,\ B_0 = 34$

$C_{00} = 2788,\ C_{11} = 988$

$D = 69 \times (-42) = -2898$

Algorithme de Karatsuba (1962)

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$

$B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

Récursion $C_{00} = A_0 \times B_0, \; C_{11} = A_1 \times B_1$

$D = (A_0 - A_1) \times (B_0 - B_1)$

Combiner $C = C_{00} + 10^{\lfloor n/2 \rfloor}(C_{00} + C_{11} - D) + 10^{2\lfloor n/2 \rfloor} C_{11}$

$A = 1382, \; B = 7634$
$A_1 = 13, \; A_0 = 82$
$B_1 = 76, \; B_0 = 34$
$C_{00} = 2788, \; C_{11} = 988$
$D = 69 \times (-42) = -2898$
$C = 2788 + 100 \cdot (2788 + 988 + 2898) + 10000 \cdot 988 = 10550188$

Algorithme de Karatsuba (1962)

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$

$B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

Récursion $C_{00} = A_0 \times B_0, \; C_{11} = A_1 \times B_1$

$D = (A_0 - A_1) \times (B_0 - B_1)$

Combiner $C = C_{00} + 10^{\lfloor n/2 \rfloor}(C_{00} + C_{11} - D) + 10^{2\lfloor n/2 \rfloor} C_{11}$

$A = 1382, \; B = 7634$
$A_1 = 13, \; A_0 = 82$
$B_1 = 76, \; B_0 = 34$
$C_{00} = 2788, \; C_{11} = 988$
$D = 69 \times (-42) = -2898$
$C = 2788 + 100 \cdot (2788 + 988 + 2898) + 10000 \cdot 988 = 10550188$

Algorithme : $\textsc{Karatsuba}(A, B)$

si A *et* B *n'ont qu'un chiffre* alors retourner $a_0 b_0$

Écrire A sous la forme $A_0 + 10^{\lfloor n/2 \rfloor} A_1$
Écrire B sous la forme $B_0 + 10^{\lfloor n/2 \rfloor} B_1$
$C_{00} \leftarrow \textsc{Karatsuba}(A_0, B_0)$
$C_{11} \leftarrow \textsc{Karatsuba}(A_1, B_1)$
$D \leftarrow \textsc{Karatsuba}(A_0 - A_1, B_0 - B_1)$
retourner $C_{00} + 10^{\lfloor n/2 \rfloor}(C_{00} + C_{11} - D) + 10^{2\lfloor n/2 \rfloor} C_{11}$

Analyse de l'algorithme de Karatsuba

Algorithme : $\textsc{Karatsuba}(A, B)$

si *A et B n'ont qu'un chiffre* alors retourner $a_0 b_0$;

Écrire A sous la forme $A_0 + 10^{\lfloor n/2 \rfloor} A_1$

Écrire B sous la forme $B_0 + 10^{\lfloor n/2 \rfloor} B_1$

$C_{00} \leftarrow \textsc{Karatsuba}(A_0, B_0)$

$C_{11} \leftarrow \textsc{Karatsuba}(A_1, B_1)$

$D \leftarrow \textsc{Karatsuba}(A_0 - A_1, B_0 - B_1)$

retourner $C_{00} + 10^{\lfloor n/2 \rfloor}(C_{00} + C_{11} - D) + 10^{2\lfloor n/2 \rfloor} C_{11}$

Lemme

L'algorithme de Karatsuba retourne le produit de A et B et si $K(n)$ dénote le temps de calcul de $\textsc{Karatsuba}$ pour des entrées de taille n, on a $K(n) \leq 3K(\lceil n/2 \rceil) + O(n)$

Analyse de l'algorithme de Karatsuba

Algorithme : $\textsc{Karatsuba}(A,B)$

si *A et B n'ont qu'un chiffre* alors retourner $a_0 b_0$;

Écrire A sous la forme $A_0 + 10^{\lfloor n/2 \rfloor} A_1$

Écrire B sous la forme $B_0 + 10^{\lfloor n/2 \rfloor} B_1$

$C_{00} \leftarrow \textsc{Karatsuba}(A_0, B_0)$

$C_{11} \leftarrow \textsc{Karatsuba}(A_1, B_1)$

$D \leftarrow \textsc{Karatsuba}(A_0 - A_1, B_0 - B_1)$

retourner $C_{00} + 10^{\lfloor n/2 \rfloor}(C_{00} + C_{11} - D) + 10^{2\lfloor n/2 \rfloor} C_{11}$

Lemme

L'algorithme de Karatsuba retourne le produit de A et B et si $K(n)$ dénote le temps de calcul de Karatsuba pour des entrées de taille n, on a $K(n) \leq 3K(\lceil n/2 \rceil) + O(n)$

Corollaire (*master theorem*)

$(a = 3,\ b = 2$ et $d = 1,\ b^d < a)$

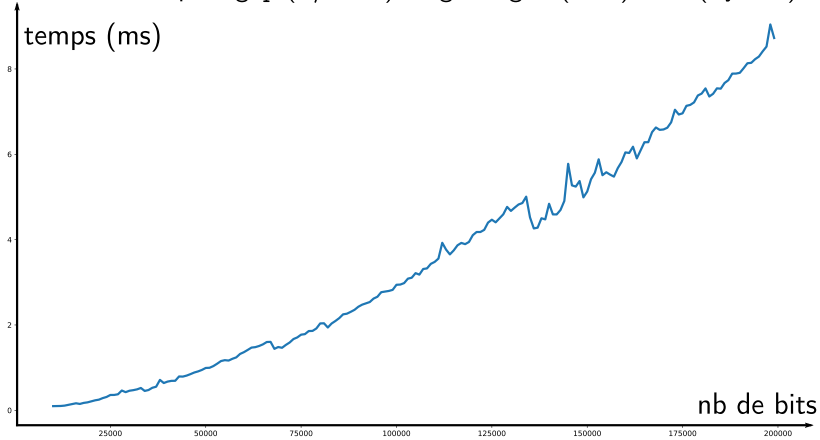
$K(n) = O(n^{\log 3/\log 2}) = \mathrm{O}(\mathrm{n}^{\log 3}) \simeq O(n^{1,58})$

Dans la *vraie vie*

- Base 10 $\rightsquigarrow$ bases 2^{32}, 2^{64}, ...
  - Grands entiers représentés comme liste d'entiers de taille k bits $\iff$ entiers écrits en base 2^k !
  - Exemple : `gmp` (C/C++), `BigInteger` (Java), `int` (Python)

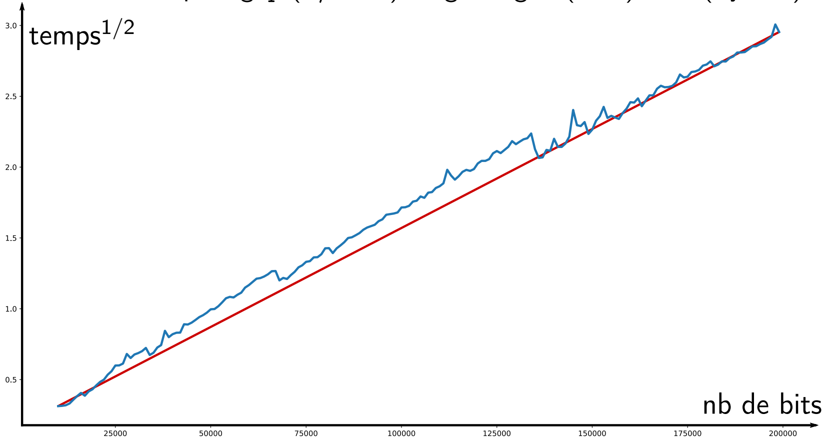
Dans la *vraie vie*

- Base 10 $\rightsquigarrow$ bases 2^{32}, 2^{64}, ...
  - Grands entiers représentés comme liste d'entiers de taille k bits $\iff$ entiers écrits en base 2^k !
  - Exemple : `gmp` (C/C++), `BigInteger` (Java), `int` (Python)



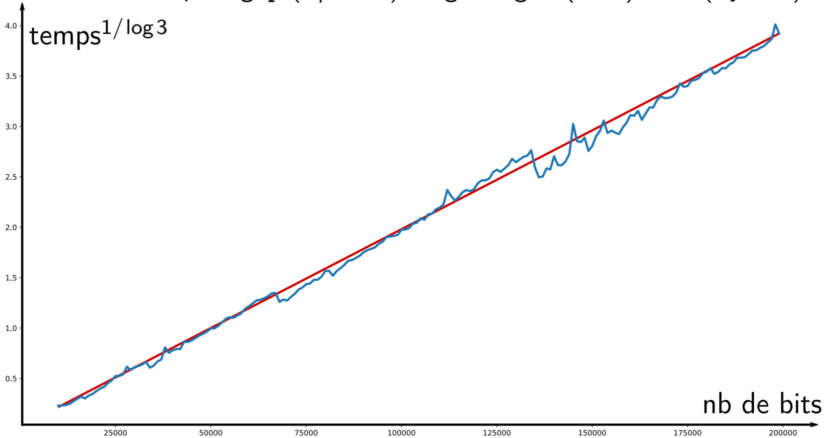
Dans la *vraie vie*

- Base 10 $\rightsquigarrow$ bases 2^{32}, 2^{64}, ...
  - Grands entiers représentés comme liste d'entiers de taille k bits $\iff$ entiers écrits en base 2^k !
  - Exemple : `gmp` (C/C++), `BigInteger` (Java), `int` (Python)



Dans la *vraie vie*

- Base 10 $\rightsquigarrow$ bases 2^{32}, 2^{64}, ...
  - Grands entiers représentés comme liste d'entiers de taille k bits $\iff$ entiers écrits en base 2^k !
  - Exemple : `gmp` (C/C++), `BigInteger` (Java), `int` (Python)



Dans la *vraie vie*

- Base 10 $\rightsquigarrow$ bases 2^{32}, 2^{64}, ...
  - Grands entiers représentés comme liste d'entiers de taille k bits $\iff$ entiers écrits en base 2^k !
  - Exemple : `gmp` (C/C++), `BigInteger` (Java), `int` (Python)
- Remarque par rapport au modèle
  - Modèle du cours : multiplication en temps $O(1)$
  - Irréaliste pour de grands entiers
  - Modèle souvent utilisé : taille d'un registre $= O(\log n)$

Dans la *vraie vie*

- Base 10 $\rightsquigarrow$ bases 2^{32}, 2^{64}, ...
  - Grands entiers représentés comme liste d'entiers de taille k bits $\iff$ entiers écrits en base 2^k !
  - Exemple : `gmp` (C/C++), `BigInteger` (Java), `int` (Python)
- Remarque par rapport au modèle
  - Modèle du cours : multiplication en temps $O(1)$
  - Irréaliste pour de grands entiers
  - Modèle souvent utilisé : taille d'un registre $= O(\log n)$
- Autre utilisation : polynômes

Dans la *vraie vie*

- Base 10 $\rightsquigarrow$ bases 2^{32}, 2^{64}, ...
  - Grands entiers représentés comme liste d'entiers de taille k bits $\iff$ entiers écrits en base 2^k !
  - Exemple : `gmp` (C/C++), `BigInteger` (Java), `int` (Python)
- Remarque par rapport au modèle
  - Modèle du cours : multiplication en temps $O(1)$
  - Irréaliste pour de grands entiers
  - Modèle souvent utilisé : taille d'un registre $= O(\log n)$
- Autre utilisation : polynômes
- Algorithmes plus rapides (1960's)
  - Toom-3 : découpe en 3 morceaux $O(n^{1,465})$
  - Toom-Cook : découpe en r morceaux $O(n^{1+\epsilon})$
  - Algorithmes basés sur la FFT $O(n \log n \log \log n)$

Dans la *vraie vie*

- Base 10 $\rightsquigarrow$ bases 2^{32}, 2^{64}, ...
  - Grands entiers représentés comme liste d'entiers de taille k bits $\iff$ entiers écrits en base 2^k !
  - Exemple : `gmp` (C/C++), `BigInteger` (Java), `int` (Python)
- Remarque par rapport au modèle
  - Modèle du cours : multiplication en temps $O(1)$
  - Irréaliste pour de grands entiers
  - Modèle souvent utilisé : taille d'un registre $= O(\log n)$
- Autre utilisation : polynômes
- Algorithmes plus rapides (1960's)
  - Toom-3 : découpe en 3 morceaux $O(n^{1,465})$
  - Toom-Cook : découpe en r morceaux $O(n^{1+\epsilon})$
  - Algorithmes basés sur la FFT $O(n \log n \log\log n)$
  - Record actuel (mars 2019), utilise la FFT $O(n \log n)$

1. Qlqs rappels sur la récursivité

2. Premier exemple : tri fusion

3. Qu'est-ce que « diviser pour régner » ?

4. Deuxième exemple : multiplication d'entiers

5. Exemple spécial : Calcul de rang

Définition et algorithmes ± naïfs

Entrée Un tableau T de n nombres, et un entier $k \in \{1, \ldots, n\}$

Sortie le $k^{\text{ème}}$ plus petit élément de T, noté $\mathrm{rang}(\mathrm{k}, \mathrm{T})$

Rk : $k = 1 \rightsquigarrow$ min, $k = n \rightsquigarrow$ max, $k = n/2 \rightsquigarrow$ médiane

Définition et algorithmes ± naïfs

Entrée Un tableau T de n nombres, et un entier $k \in \{1, \ldots, n\}$

Sortie le $k^{\text{ème}}$ plus petit élément de T, noté $\mathrm{rang}(\mathrm{k}, \mathrm{T})$

Rk : $k = 1 \rightsquigarrow$ min, $k = n \rightsquigarrow$ max, $k = n/2 \rightsquigarrow$ médiane

Algorithme en $O(n^2)$:

```
pour i = 0 à n−1 faire
    c = 0
    pour j = 0 à n−1 faire
        si T[j] ≤ T[i] alors
            c ← c + 1
    si c = k alors retourner
      T[i]
```

Définition et algorithmes ± naïfs

Entrée Un tableau T de n nombres, et un entier $k \in \{1, \ldots, n\}$

Sortie le $k^{\text{ème}}$ plus petit élément de T, noté rang(k, T)

Rk : $k = 1 \rightsquigarrow$ min, $k = n \rightsquigarrow$ max, $k = n/2 \rightsquigarrow$ médiane

Algorithme en $O(n^2)$:

```
pour i = 0 à n−1 faire
    c = 0
    pour j = 0 à n−1 faire
        si T[j] ≤ T[i] alors
            c ← c + 1
    si c = k alors retourner
      T[i]
```

Algorithme en $O(n \log n)$:

```
Trier T
retourner T[k−1]
```

Définition et algorithmes ± naïfs

Entrée Un tableau T de n nombres, et un entier $k \in \{1,\dots,n\}$

Sortie le $k^{\text{ème}}$ plus petit élément de T, noté rang(k, T)

Rk : $k = 1 \rightsquigarrow$ min, $k = n \rightsquigarrow$ max, $k = n/2 \rightsquigarrow$ médiane

Algorithme en $O(n^2)$:

```
pour i = 0 à n−1 faire
    c = 0
    pour j = 0 à n−1 faire
        si T[j] ≤ T[i] alors
            c ← c+1
    si c = k alors retourner
      T[i]
```

Algorithme en $O(n \log n)$:

```
Trier T
retourner T[k−1]
```

Algorithme en O(n) ?

Stratégie « diviser pour régner »

Diviser Choisir un pivot $p = T[i_0] \in T$ pour séparer T en trois :

- $\mathrm{T}_{\mathrm{inf}}$ contient les éléments x de T vérifiant $x < p$. $\mathrm{n}_{\mathrm{inf}} = |T_{\mathrm{inf}}|$
- T_{eq} contient les éléments x de T vérifiant $x = p$. $\mathrm{n}_{\mathrm{eq}} = |T_{\mathrm{eq}}|$
- $\mathrm{T}_{\mathrm{sup}}$ contient les éléments x de T vérifiant $x > p$. $\mathrm{n}_{\mathrm{sup}} = |T_{\mathrm{sup}}|$

Stratégie « diviser pour régner »

Diviser Choisir un pivot $p = T[i_0] \in T$ pour séparer T en trois :

- $\mathrm{T}_{\mathrm{inf}}$ contient les éléments x de T vérifiant $x < p$. $\mathrm{n}_{\mathrm{inf}} = |T_{\mathrm{inf}}|$
- T_{eq} contient les éléments x de T vérifiant $x = p$. $\mathrm{n}_{\mathrm{eq}} = |T_{\mathrm{eq}}|$
- $\mathrm{T}_{\mathrm{sup}}$ contient les éléments x de T vérifiant $x > p$. $\mathrm{n}_{\mathrm{sup}} = |T_{\mathrm{sup}}|$

Exemple : $T =$

3	7	21	9	12	16	7	4	1	2

On choisit le pivot $T[1] = 7$, on a :

$\mathrm{T}_{\mathrm{inf}} =$

3	4	1	2

$n_{inf} = 4$

$\mathrm{T}_{\mathrm{eq}} =$

7	7

$n_{inf} = 2$

$\mathrm{T}_{\mathrm{sup}} =$

21	9	12	16

$n_{inf} = 4$

Du coup :

- Le 2ème rang de T est le 2ème rang de T_{inf}
- Le 5ème rang de T est $T[1] = 7$
- Le 9ème rang de T est le 3ème rang de T_{sup}

Stratégie « diviser pour régner »

Diviser Choisir un pivot $p = T[i_0] \in T$ pour séparer T en trois :

- $\mathrm{T}_{\mathrm{inf}}$ contient les éléments x de T vérifiant $x < p$. $\mathrm{n}_{\mathrm{inf}} = |T_{\mathrm{inf}}|$
- T_{eq} contient les éléments x de T vérifiant $x = p$. $\mathrm{n}_{\mathrm{eq}} = |T_{\mathrm{eq}}|$
- $\mathrm{T}_{\mathrm{sup}}$ contient les éléments x de T vérifiant $x > p$. $\mathrm{n}_{\mathrm{sup}} = |T_{\mathrm{sup}}|$

Récursion Trouver $\mathrm{rang}(k, T)$ dans T_{inf} ou dans T_{sup}

$$\mathrm{rang}(k, T) = \begin{cases} \mathrm{rang}(k, T_{\mathrm{inf}}) & \text{si } k \leq n_{\mathrm{inf}} \\ p & \text{si } n_{\mathrm{inf}} < k \leq n_{\mathrm{inf}} + n_{\mathrm{eq}} \\ \mathrm{rang}(k - n_{\mathrm{inf}} - n_{\mathrm{eq}}, T_{\mathrm{sup}}) & \text{si } n_{\mathrm{inf}} + n_{\mathrm{eq}} < k \end{cases}$$

Combiner Rien à faire...

Idée d'algorithme

- Idée d'algo : on choisit un pivot, puis on crée T_{inf} et T_{sup} soit on a trouvé le $k$ième rang, soit on relance récursivement sur T_{inf} ou sur T_{sup}.

Idée d'algorithme

- Idée d'algo : on choisit un pivot, puis on crée T_{inf} et T_{sup} soit on a trouvé le $k$ième rang, soit on relance récursivement sur T_{inf} ou sur T_{sup}.

- Si on pouvait trouver à chaque étape un pivot tel que
  $n_{\text{inf}} \sim n/2$ et $n_{\text{sup}} \sim n/2$
  Alors on aurait comme équation de récurrence :

  $$t(n) \leq t(n/2) + O(n)$$

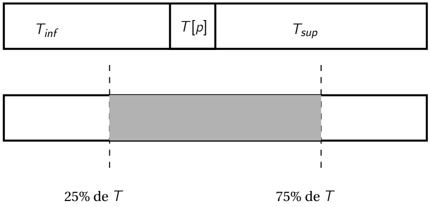
  Le 'Master Theorem' dit $(a = 1,\ b = 2$ et $d = 1,\ b^d = 2 > 1 = a)$
  $t(n) = O(n^d) = O(n)$ Bingo !

Idée d'algorithme, on chauffe...

- Mais c'est impossible de garantir $n_{\text{inf}} \sim n/2$ et $n_{\text{sup}} \sim n/2$...

Idée d'algorithme, on chauffe...

- Mais c'est impossible de garantir $n_{\mathrm{inf}} \sim n/2$ et $n_{\mathrm{sup}} \sim n/2$...
- Regardons ce qui se passe sur T trié :

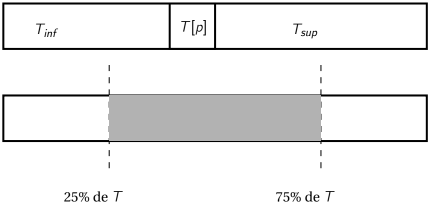


Si $T[p]$ est choisi dans la zone hachurée, alors on est sûr que

$$n_{\mathrm{inf}} \leq \lceil 3n/4 \rceil \quad \text{et} \quad n_{\mathrm{sup}} \leq \lceil 3n/4 \rceil$$

Idée d'algorithme, on chauffe...

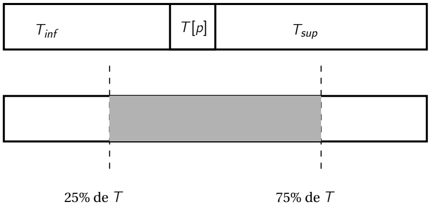
- Mais c'est impossible de garantir $n_{\text{inf}} \sim n/2$ et $n_{\text{sup}} \sim n/2$...
- Regardons ce qui se passe sur T trié :



  Si $T[p]$ est choisi dans la zone hachurée, alors on est sûr que
  $$n_{\text{inf}} \leq \lceil 3n/4 \rceil \quad \text{et} \quad n_{\text{sup}} \leq \lceil 3n/4 \rceil$$
- Alors on aurait comme équation de récurrence :
  $$t(n) \leq t(\lceil 3n/4 \rceil) + O(n)$$
  Le 'Master Theorem' dit ($a = 1$, $b = 4/3$ et $d = 1$, $b^d = 4/3 > 1 = a$) $t(n) = O(n^d) = O(n)$. Bingo aussi !

Idée d'algorithme, on chauffe...

- Mais c'est impossible de garantir $n_{\text{inf}} \sim n/2$ et $n_{\text{sup}} \sim n/2$...
- Regardons ce qui se passe sur T trié :



Si $T[p]$ est choisi dans la zone hachurée, alors on est sûr que
$$n_{\text{inf}} \leq \lceil 3n/4 \rceil \quad \text{et} \quad n_{\text{sup}} \leq \lceil 3n/4 \rceil$$

- Alors on aurait comme équation de récurrence :
$$t(n) \leq t(\lceil 3n/4 \rceil) + O(n)$$
Le 'Master Theorem' dit ($a = 1$, $b = 4/3$ et $d = 1$, $b^d = 4/3 > 1 = a$) $t(n) = O(n^d) = O(n)$. Bingo aussi !
- Et on a 1 chance sur 2 de tirer $T[p]$ dans la zone hachurée !

L'algorithme

Algorithme : $\text{RANG}(T, k)$

si $k = 1$ alors retourner $T[0]$;
$n_{\text{inf}} \leftarrow n$; `// n est la taille de T`
$n_{\text{sup}} \leftarrow n$;
tant que $n_{inf} > \lceil 3n/4 \rceil$ ou $n_{sup} > \lceil 3n/4 \rceil$ faire
  Choisir p au hasard entre 0 et $n-1$;
  Calculer $T[p]$, T_{inf} et T_{sup} puis n_{inf}, n_{eq} et n_{sup} ;
si $k \leq n_{inf}$ alors
  retourner $\mathit{RANG}(T_{inf}, k)$
sinon si $n_{inf} < k \leq n_{inf} + n_{eq}$ alors
  retourner p
sinon
  retourner $\mathit{RANG}(T_{sup}, k - n_{inf} - n_{eq})$

- En moyenne, on ne fait que 2 tours de boucles 'Tant que'
  $\rightsquigarrow t(n) \simeq O(n)$