

### Principe général

Pour résoudre un problème donné, une stratégie peut-être de se ramener à un ou plusieurs sous problèmes de *même type* mais *plus petit*. En notant  $P(n)$ , un problème de taille  $n$ , la résolution de  $P_n$  conduit à la résolution de  $k$  problèmes de tailles  $P(\frac{n}{p})$ . Un fois ces problèmes résolus, leurs solutions sont combinées afin de former celle du problème initial.

### Principe général

Pour résoudre un problème donné, une stratégie peut-être de se ramener à un ou plusieurs sous problèmes de *même type* mais *plus petit*. En notant  $P(n)$ , un problème de taille  $n$ , la résolution de  $P_n$  conduit à la résolution de  $k$  problèmes de tailles  $P(\frac{n}{p})$ . Un fois ces problèmes résolus, leurs solutions sont combinées afin de former celle du problème initial.

On distingue :

### Principe général

Pour résoudre un problème donné, une stratégie peut-être de se ramener à un ou plusieurs sous problèmes de *même type* mais *plus petit*. En notant  $P(n)$ , un problème de taille  $n$ , la résolution de  $P_n$  conduit à la résolution de  $k$  problèmes de tailles  $P(\frac{n}{p})$ . Un fois ces problèmes résolus, leurs solutions sont combinées afin de former celle du problème initial.

On distingue :

- la méthode **diviser pour régner**, dans laquelle les sous-problèmes sont indépendants

## Principe général

Pour résoudre un problème donné, une stratégie peut-être de se ramener à un ou plusieurs sous problèmes de *même type* mais *plus petit*. En notant  $P(n)$ , un problème de taille  $n$ , la résolution de  $P_n$  conduit à la résolution de  $k$  problèmes de tailles  $P(\frac{n}{p})$ . Un fois ces problèmes résolus, leurs solutions sont combinées afin de former celle du problème initial.

On distingue :

- la méthode **diviser pour régner**, dans laquelle les sous-problèmes sont indépendants
- la méthode de **programmation dynamique** dans laquelle, certains sous problèmes se chevauchent.

### Exemple introductif

Le tri fusion est l'exemple typique d'une résolution par la méthode diviser pour régner. En effet, pour trier une liste  $l$  de taille  $n$ ,

### Exemple introductif

Le tri fusion est l'exemple typique d'une résolution par la méthode diviser pour régner. En effet, pour trier une liste  $l$  de taille  $n$ ,

- **Diviser** : on sépare  $l$  en deux moitiés (à une unité près)  $l_1$  et  $l_2$ . Dans cet exemple  $P(n)$  se ramène à la résolution de 2 instances de résolution de  $P(n/2)$ .

### Exemple introductif

Le tri fusion est l'exemple typique d'une résolution par la méthode diviser pour régner. En effet, pour trier une liste  $l$  de taille  $n$ ,

- **Diviser** : on sépare  $l$  en deux moitiés (à une unité près)  $l_1$  et  $l_2$ . Dans cet exemple  $P(n)$  se ramène à la résolution de 2 instances de résolution de  $P(n/2)$ .
- **Régner** : on trie  $l_1$  et  $l_2$

### Exemple introductif

Le tri fusion est l'exemple typique d'une résolution par la méthode diviser pour régner. En effet, pour trier une liste  $l$  de taille  $n$ ,

- **Diviser** : on sépare  $l$  en deux moitiés (à une unité près)  $l_1$  et  $l_2$ . Dans cet exemple  $P(n)$  se ramène à la résolution de 2 instances de résolution de  $P(n/2)$ .
- **Régner** : on trie  $l_1$  et  $l_2$
- **Combiner** : on fusionne les listes triées afin de construire la solution au problème initial



### Tri fusion en OCaml

- 1 Ecrire une fonction `separe int list -> int list * int list` qui prend en argument une liste d'entiers et renvoie les deux moitiés de cette liste.

### Tri fusion en OCaml

- 1 Ecrire une fonction `separe int list -> int list * int list` qui prend en argument une liste d'entiers et renvoie les deux moitiés de cette liste.
  
  
  
  
  
  
  
  
  
  
- 2 Ecrire une fonction `fusion int list -> int list -> int list` qui fusionne les deux listes données en arguments.

### Tri fusion en OCaml

- 1 Ecrire une fonction `separe int list -> int list * int list` qui prend en argument une liste d'entiers et renvoie les deux moitiés de cette liste.
  
- 2 Ecrire une fonction `fusion int list -> int list -> int list` qui fusionne les deux listes données en arguments.

## Tri fusion en OCaml

- 1 Ecrire une fonction `separe int list -> int list * int list` qui prend en argument une liste d'entiers et renvoie les deux moitiés de cette liste.

```
1  let rec separe l =  
2    match l with  
3    | [] -> [], []  
4    | [x] -> [x], []  
5    | h1::h2::t -> let ft1,ft2 = (separe t) in h1::ft1, h2::ft2;;
```

- 2 Ecrire une fonction `fusion int list -> int list -> int list` qui fusionne les deux listes données en arguments.

## Tri fusion en OCaml

- ① Ecrire une fonction `separe int list -> int list * int list` qui prend en argument une liste d'entiers et renvoie les deux moitiés de cette liste.

```

1  let rec separe l =
2    match l with
3    | [] -> [], []
4    | [x] -> [x], []
5    | h1::h2::t-> let ft1,ft2 = (separe t) in h1::ft1, h2::ft2;;

```

- ② Ecrire une fonction `fusion int list -> int list -> int list` qui fusionne les deux listes données en arguments.

```

1  let rec fusion l1 l2 =
2    match l1,l2 with
3    | [], x -> x
4    | x, [] -> x
5    | h1::t1, h2::t2 -> if h1<h2 then h1::(fusion t1 l2) else
    ↪ h2::(fusion l1 t2);;

```

### Tri fusion en OCaml

- ④ Ecrire une fonction `tri_fusion int list -> int list` qui renvoie la liste donnée en argument triée.

### Tri fusion en OCaml

- ⑥ Ecrire une fonction `tri_fusion int list -> int list` qui renvoie la liste donnée en argument triée.

```
1  let rec tri_fusion l =  
2    match l with  
3    | [] -> []  
4    | [x] -> [x]  
5    | _ -> let l1,l2 = separe l in  
6            let t11 = tri_fusion l1 and t12 = tri_fusion l2 in  
7            (fusion t11 t12);;
```

# C15 Décomposition en sous problèmes

## 2. Diviser pour regner

### Equation de complexité

On note :

- $C(n)$  la complexité de la résolution d'un problème de taille  $n$ .



# C15 Décomposition en sous problèmes

## 2. Diviser pour regner

### Equation de complexité

On note :

- $C(n)$  la complexité de la résolution d'un problème de taille  $n$ .
- $k$  le nombre de sous problèmes à résoudre et  $\frac{n}{p}$  leur taille.

# C15 Décomposition en sous problèmes

## 2. Diviser pour regner

### Equation de complexité

On note :

- $C(n)$  la complexité de la résolution d'un problème de taille  $n$ .
- $k$  le nombre de sous problèmes à résoudre et  $\frac{n}{p}$  leur taille.
- $T(n)$  le coût de construction de la solution de taille  $n$ .

# C15 Décomposition en sous problèmes

## 2. Diviser pour regner

### Equation de complexité

On note :

- $C(n)$  la complexité de la résolution d'un problème de taille  $n$ .
- $k$  le nombre de sous problèmes à résoudre et  $\frac{n}{p}$  leur taille.
- $T(n)$  le coût de construction de la solution de taille  $n$ .

On en déduit l' équation de complexité :

# C15 Décomposition en sous problèmes

## 2. Diviser pour regner

### Equation de complexité

On note :

- $C(n)$  la complexité de la résolution d'un problème de taille  $n$ .
- $k$  le nombre de sous problèmes à résoudre et  $\frac{n}{p}$  leur taille.
- $T(n)$  le coût de construction de la solution de taille  $n$ .

On en déduit l' équation de complexité :

$$C(n) = kC\left(\frac{n}{p}\right) + T(n)$$

# C15 Décomposition en sous problèmes

## 2. Diviser pour régner

### Equation de complexité

On note :

- $C(n)$  la complexité de la résolution d'un problème de taille  $n$ .
- $k$  le nombre de sous problèmes à résoudre et  $\frac{n}{p}$  leur taille.
- $T(n)$  le coût de construction de la solution de taille  $n$ .

On en déduit l' équation de complexité :

$$C(n) = kC\left(\frac{n}{p}\right) + T(n)$$

On suppose de plus que la résolution d'un problème de taille inférieure à un entier  $m$  donnée s'effectue en temps constant

## Equation de complexité

On note :

- $C(n)$  la complexité de la résolution d'un problème de taille  $n$ .
- $k$  le nombre de sous problèmes à résoudre et  $\frac{n}{p}$  leur taille.
- $T(n)$  le coût de construction de la solution de taille  $n$ .

On en déduit l' équation de complexité :

$$C(n) = kC\left(\frac{n}{p}\right) + T(n)$$

On suppose de plus que la résolution d'un problème de taille inférieure à un entier  $m$  donnée s'effectue en temps constant

## Exemple

Dans le cas du tri fusion,

# C15 Décomposition en sous problèmes

## 2. Diviser pour régner

### Equation de complexité

On note :

- $C(n)$  la complexité de la résolution d'un problème de taille  $n$ .
- $k$  le nombre de sous problèmes à résoudre et  $\frac{n}{p}$  leur taille.
- $T(n)$  le coût de construction de la solution de taille  $n$ .

On en déduit l' équation de complexité :

$$C(n) = kC\left(\frac{n}{p}\right) + T(n)$$

On suppose de plus que la résolution d'un problème de taille inférieure à un entier  $m$  donnée s'effectue en temps constant

### Exemple

Dans le cas du tri fusion,

- Donner les valeurs de  $k$ ,  $p$   $m$  et écrire les équations de complexité.

# C15 Décomposition en sous problèmes

## 2. Diviser pour regner

### Equation de complexité

On note :

- $C(n)$  la complexité de la résolution d'un problème de taille  $n$ .
- $k$  le nombre de sous problèmes à résoudre et  $\frac{n}{p}$  leur taille.
- $T(n)$  le coût de construction de la solution de taille  $n$ .

On en déduit l' équation de complexité :

$$C(n) = kC\left(\frac{n}{p}\right) + T(n)$$

On suppose de plus que la résolution d'un problème de taille inférieure à un entier  $m$  donnée s'effectue en temps constant

### Exemple

Dans le cas du tri fusion,

- Donner les valeurs de  $k$ ,  $p$   $m$  et écrire les équations de complexité.
- Donner un  $O$  de  $T(n)$ .



# C15 Décomposition en sous problèmes

## 2. Diviser pour regner

### Résolution

# C15 Décomposition en sous problèmes

## 2. Diviser pour regner

### Résolution

Si la complexité d'un problème est définie par une équation de la forme :  
 $C(n) = kC(\frac{n}{p}) + f(n)$ , où  $f$  est un polynôme de degré  $d$ , alors

# C15 Décomposition en sous problèmes

## 2. Diviser pour regner

### Résolution

Si la complexité d'un problème est définie par une équation de la forme :  
 $C(n) = kC(\frac{n}{p}) + f(n)$ , où  $f$  est un polynôme de degré  $d$ , alors

- Si  $d < \frac{\log(k)}{\log(p)}$ , alors  $C \in O(n^{\frac{\log(k)}{\log(p)}})$

# C15 Décomposition en sous problèmes

## 2. Diviser pour regner

### Résolution

Si la complexité d'un problème est définie par une équation de la forme :  
 $C(n) = kC(\frac{n}{p}) + f(n)$ , où  $f$  est un polynôme de degré  $d$ , alors

- Si  $d < \frac{\log(k)}{\log(p)}$ , alors  $C \in O(n^{\frac{\log(k)}{\log(p)}})$
- Si  $d = \frac{\log(k)}{\log(p)}$ , alors  $C \in O(n^d \log(n))$

# C15 Décomposition en sous problèmes

## 2. Diviser pour regner

### Résolution

Si la complexité d'un problème est définie par une équation de la forme :  
 $C(n) = kC(\frac{n}{p}) + f(n)$ , où  $f$  est un polynôme de degré  $d$ , alors

- Si  $d < \frac{\log(k)}{\log(p)}$ , alors  $C \in O(n^{\frac{\log(k)}{\log(p)}})$
- Si  $d = \frac{\log(k)}{\log(p)}$ , alors  $C \in O(n^d \log(n))$
- Si  $d > \frac{\log(k)}{\log(p)}$ , alors  $C \in O(n^d)$

# C15 Décomposition en sous problèmes

## 2. Diviser pour regner

### Résolution

Si la complexité d'un problème est définie par une équation de la forme :  
 $C(n) = kC(\frac{n}{p}) + f(n)$ , où  $f$  est un polynôme de degré  $d$ , alors

- Si  $d < \frac{\log(k)}{\log(p)}$ , alors  $C \in O(n^{\frac{\log(k)}{\log(p)}})$
- Si  $d = \frac{\log(k)}{\log(p)}$ , alors  $C \in O(n^d \log(n))$
- Si  $d > \frac{\log(k)}{\log(p)}$ , alors  $C \in O(n^d)$

⚠ Ce théorème appelé (*master theorem*) est hors-programme.

# C15 Décomposition en sous problèmes

## 2. Diviser pour regner

### Résolution

Si la complexité d'un problème est définie par une équation de la forme :  
 $C(n) = kC(\frac{n}{p}) + f(n)$ , où  $f$  est un polynôme de degré  $d$ , alors

- Si  $d < \frac{\log(k)}{\log(p)}$ , alors  $C \in O(n^{\frac{\log(k)}{\log(p)}})$
- Si  $d = \frac{\log(k)}{\log(p)}$ , alors  $C \in O(n^d \log(n))$
- Si  $d > \frac{\log(k)}{\log(p)}$ , alors  $C \in O(n^d)$

**⚠** Ce théorème appelé (*master theorem*) est hors-programme.  
Mais il permet de résoudre instantanément les équations de complexité de la méthode diviser pour régner !

# C15 Décomposition en sous problèmes

## 2. Diviser pour régner

### Complexité d'une méthode diviser pour régner

- Dans la plupart des cas, on peut résoudre l'équation de complexité **sans** le *master theorem*.



### Complexité d'une méthode diviser pour régner

- Dans la plupart des cas, on peut résoudre l'équation de complexité **sans** le *master theorem*.

Par exemple, dans le cas du tri fusion, les équations sont :

$$\begin{cases} C(0) & \in O(1) \\ C(2n) & = 2C(n) + O(n) \end{cases}$$

### Complexité d'une méthode diviser pour régner

- Dans la plupart des cas, on peut résoudre l'équation de complexité **sans** le *master theorem*.

Par exemple, dans le cas du tri fusion, les équations sont :

$$\begin{cases} C(0) & \in O(1) \\ C(2n) & = 2C(n) + O(n) \end{cases}$$

Pour simplifier on suppose que  $n$  est une puissance exacte de 2 et on obtient :

$C(2^{k+1}) \leq 2C(2^k) + M2^k$  et en divisant par  $2^{k+1}$ , on obtient

### Complexité d'une méthode diviser pour régner

- Dans la plupart des cas, on peut résoudre l'équation de complexité **sans** le *master theorem*.

Par exemple, dans le cas du tri fusion, les équations sont :

$$\begin{cases} C(0) & \in O(1) \\ C(2n) & = 2C(n) + O(n) \end{cases}$$

Pour simplifier on suppose que  $n$  est une puissance exacte de 2 et on obtient :

$C(2^{k+1}) \leq 2C(2^k) + M2^k$  et en divisant par  $2^{k+1}$ , on obtient

$$u_{k+1} \leq u_k + M \text{ où } u_k = \frac{C(2^k)}{2^k}.$$

### Complexité d'une méthode diviser pour régner

- Dans la plupart des cas, on peut résoudre l'équation de complexité **sans** le *master theorem*.

Par exemple, dans le cas du tri fusion, les équations sont :

$$\begin{cases} C(0) & \in O(1) \\ C(2n) & = 2C(n) + O(n) \end{cases}$$

Pour simplifier on suppose que  $n$  est une puissance exacte de 2 et on obtient :

$C(2^{k+1}) \leq 2C(2^k) + M2^k$  et en divisant par  $2^{k+1}$ , on obtient

$$u_{k+1} \leq u_k + M \text{ où } u_k = \frac{C(2^k)}{2^k}.$$

par récurrence immédiate,  $u_k \leq u_0 + kM$

### Complexité d'une méthode diviser pour régner

- Dans la plupart des cas, on peut résoudre l'équation de complexité **sans** le *master theorem*.

Par exemple, dans le cas du tri fusion, les équations sont :

$$\begin{cases} C(0) & \in O(1) \\ C(2n) & = 2C(n) + O(n) \end{cases}$$

Pour simplifier on suppose que  $n$  est une puissance exacte de 2 et on obtient :

$C(2^{k+1}) \leq 2C(2^k) + M2^k$  et en divisant par  $2^{k+1}$ , on obtient

$$u_{k+1} \leq u_k + M \text{ où } u_k = \frac{C(2^k)}{2^k}.$$

par récurrence immédiate,  $u_k \leq u_0 + kM$  et donc  $C(n) \leq nu_0 + Mn \log_2(n)$

### Complexité d'une méthode diviser pour régner

- Dans la plupart des cas, on peut résoudre l'équation de complexité **sans** le *master theorem*.

Par exemple, dans le cas du tri fusion, les équations sont :

$$\begin{cases} C(0) & \in O(1) \\ C(2n) & = 2C(n) + O(n) \end{cases}$$

Pour simplifier on suppose que  $n$  est une puissance exacte de 2 et on obtient :

$C(2^{k+1}) \leq 2C(2^k) + M2^k$  et en divisant par  $2^{k+1}$ , on obtient

$$u_{k+1} \leq u_k + M \text{ où } u_k = \frac{C(2^k)}{2^k}.$$

par récurrence immédiate,  $u_k \leq u_0 + kM$  et donc  $C(n) \leq nu_0 + Mn \log_2(n)$

c'est à dire  $C(n) \in O(n \log n)$ .

- Sinon, on peut utiliser le *master theorem* afin d'obtenir la complexité, puis la prouver par récurrence.

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Tranche maximale dans un tableau

On considère un tableau  $T$  de  $n$  entiers, le but du problème est de déterminer la somme maximale d'une tranche (c'est à dire d'éléments contigus de  $T$ ). Par exemple si  $T = [2, -7, -5, 4, -1, 10, -4, 9, -2]$  alors la somme maximale d'une tranche est :

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Tranche maximale dans un tableau

On considère un tableau  $T$  de  $n$  entiers, le but du problème est de déterminer la somme maximale d'une tranche (c'est à dire d'éléments contigus de  $T$ ). Par exemple si  $T = [2, -7, -5, 4, -1, 10, -4, 9, -2]$  alors la somme maximale d'une tranche est : 18, et elle est obtenue en prenant la tranche  $[4, -1, 10, -4, 9]$ .



# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Tranche maximale dans un tableau

On considère un tableau  $T$  de  $n$  entiers, le but du problème est de déterminer la somme maximale d'une tranche (c'est à dire d'éléments contigus de  $T$ ). Par exemple si  $T = [2, -7, -5, 4, -1, 10, -4, 9, -2]$  alors la somme maximale d'une tranche est : 18, et elle est obtenue en prenant la tranche  $[4, -1, 10, -4, 9]$ .

## Tranche maximale dans un tableau

On considère un tableau  $T$  de  $n$  entiers, le but du problème est de déterminer la somme maximale d'une tranche (c'est à dire d'éléments contigus de  $T$ ). Par exemple si  $T = [2, -7, -5, 4, -1, 10, -4, 9, -2]$  alors la somme maximale d'une tranche est : 18, et elle est obtenue en prenant la tranche  $[4, -1, 10, -4, 9]$ .

- 1 Proposer un algorithme de complexité quadratique permettant de résoudre ce problème.

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Tranche maximale dans un tableau

On considère un tableau  $T$  de  $n$  entiers, le but du problème est de déterminer la somme maximale d'une tranche (c'est à dire d'éléments contigus de  $T$ ). Par exemple si  $T = [2, -7, -5, 4, -1, 10, -4, 9, -2]$  alors la somme maximale d'une tranche est : 18, et elle est obtenue en prenant la tranche  $[4, -1, 10, -4, 9]$ .

- 1 Proposer un algorithme de complexité quadratique permettant de résoudre ce problème.
- 2 En donner une implémentation en langage C

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Tranche maximale dans un tableau

On considère un tableau  $T$  de  $n$  entiers, le but du problème est de déterminer la somme maximale d'une tranche (c'est à dire d'éléments contigus de  $T$ ). Par exemple si  $T = [2, -7, -5, 4, -1, 10, -4, 9, -2]$  alors la somme maximale d'une tranche est : 18, et elle est obtenue en prenant la tranche  $[4, -1, 10, -4, 9]$ .

- 1 Proposer un algorithme de complexité quadratique permettant de résoudre ce problème.
- 2 En donner une implémentation en langage C
- 3 Une solution plus efficace :
  - 1 Proposer un nouvel algorithme basé sur la méthode *diviser pour régner*

## Tranche maximale dans un tableau

On considère un tableau  $T$  de  $n$  entiers, le but du problème est de déterminer la somme maximale d'une tranche (c'est à dire d'éléments contigus de  $T$ ). Par exemple si  $T = [2, -7, -5, 4, -1, 10, -4, 9, -2]$  alors la somme maximale d'une tranche est : 18, et elle est obtenue en prenant la tranche  $[4, -1, 10, -4, 9]$ .

- 1 Proposer un algorithme de complexité quadratique permettant de résoudre ce problème.
- 2 En donner une implémentation en langage C
- 3 Une solution plus efficace :
  - 1 Proposer un nouvel algorithme basé sur la méthode *diviser pour régner*
  - 2 Donner une implémentation en C de ce nouvel algorithme

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Tranche maximale dans un tableau

On considère un tableau  $T$  de  $n$  entiers, le but du problème est de déterminer la somme maximale d'une tranche (c'est à dire d'éléments contigus de  $T$ ). Par exemple si  $T = [2, -7, -5, 4, -1, 10, -4, 9, -2]$  alors la somme maximale d'une tranche est : 18, et elle est obtenue en prenant la tranche  $[4, -1, 10, -4, 9]$ .

- 1 Proposer un algorithme de complexité quadratique permettant de résoudre ce problème.
- 2 En donner une implémentation en langage C
- 3 Une solution plus efficace :
  - 1 Proposer un nouvel algorithme basé sur la méthode *diviser pour régner*
  - 2 Donner une implémentation en C de ce nouvel algorithme
  - 3 Déterminer sa complexité

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Résolution avec une complexité quadratique

- 1 On calcule les  $S_{i,j}$  (somme de la tranche des éléments du tableau compris entre les indices  $i$  et  $j$  inclus) de proche en proche, en utilisant  $S_{ij} = S_{i,j-1} + t_j$  et on prend le maximum des valeurs obtenus.

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pou régner

### Résolution avec une complexité quadratique

- 1 On calcule les  $S_{i,j}$  (somme de la tranche des éléments du tableau compris entre les indices  $i$  et  $j$  inclus) de proche en proche, en utilisant  $S_{ij} = S_{i,j-1} + t_j$  et on prend le maximum des valeurs obtenus. On utilise donc deux boucles imbriquées dans laquelle on effectue uniquement des opérations élémentaires, la complexité est donc quadratique.



## Résolution avec une complexité quadratique

- 1 On calcule les  $S_{i,j}$  (somme de la tranche des éléments du tableau compris entre les indices  $i$  et  $j$  inclus) de proche en proche, en utilisant  $S_{ij} = S_{i,j-1} + t_j$  et on prend le maximum des valeurs obtenus. On utilise donc deux boucles imbriquées dans laquelle on effectue uniquement des opérations élémentaires, la complexité est donc quadratique.
- 2 Implémentation :

```
1  int tmaxi(int tab[], int size){
2      int tmax = tab[0];
3      int tij;
4      for (int i=0;i<size;i++){
5          tij=0;
6          for (int j=i;j<size;j++){
7              tij = tij + tab[j];
8              if (tij>tmax) {tmax = tij;}}
9      }
10     return tmax;}
```

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Résolution : diviser pour régner

① On note  $k = \lfloor \frac{n}{2} \rfloor$ .

## Résolution : diviser pour régner

- 1 On note  $k = \lfloor \frac{n}{2} \rfloor$ .
  - **Diviser** : on sépare  $T$  en deux sous tableaux  $T_g = [t_0 \dots t_{k-1}]$  et  $T_d = [t_{k+1} \dots t_{n-1}]$ .
    - ⚠ Attention : on remarquera bien que  $t_k$  n'est dans aucun des deux sous tableaux !

## Résolution : diviser pour régner

① On note  $k = \lfloor \frac{n}{2} \rfloor$ .

- **Diviser** : on sépare  $T$  en deux sous tableaux  $T_g = [t_0 \dots t_{k-1}]$  et  $T_d = [t_{k+1} \dots t_{n-1}]$ .

⚠ Attention : on remarquera bien que  $t_k$  n'est dans aucun des deux sous tableaux !

- **Régner** : on recherche les tranches maximales des sous tableaux  $T_g$  et  $T_d$  ainsi que celle des tranches contenant l'élément  $t_k$ .

## Résolution : diviser pour régner

① On note  $k = \lfloor \frac{n}{2} \rfloor$ .

- **Diviser** : on sépare  $T$  en deux sous tableaux  $T_g = [t_0 \dots t_{k-1}]$  et  $T_d = [t_{k+1} \dots t_{n-1}]$ .

⚠ Attention : on remarquera bien que  $t_k$  n'est dans aucun des deux sous tableaux !

- **Régner** : on recherche les tranches maximales des sous tableaux  $T_g$  et  $T_d$  ainsi que celle des tranches contenant l'élément  $t_k$ .
- **Combiner** on prend le maximum des trois valeurs obtenues.

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

Résolution : diviser pour régner

## Résolution : diviser pour régner

## ② Implémentation :

```
1  int somme_maxi_aux(int tab[], int size, int start, int end) {
2      if (end==start)
3          {return tab[start];}
4      if (end==start+1)
5          {return max3(tab[start],tab[end],tab[start]+tab[end]);}
6      int s1, s2, s3;
7      int mid = (start+end)/2;
8      s1 = somme_maxi_aux(tab, size, start, mid-1);
9      s2 = max_tranche(tab,start,mid,end);
10     s3 = somme_maxi_aux(tab, size, mid+1, end);
11     return max3(s1,s2,s3);}
```

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

Résolution : diviser pour régner



# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Résolution : diviser pour régner

- ④ Calcul de la complexité

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Résolution : diviser pour régner

#### ④ Calcul de la complexité

Pour résoudre un problème de taille  $n$ , on doit en résoudre deux de tailles  $\lfloor \frac{n}{2} \rfloor$  et rechercher le maximum des tranches contenant l'élément  $t_k$ . Cette opération a une complexité linéaire, on a donc :

$$\begin{cases} C(0) & \in O(1) \\ C(2n) & = 2C(n) + O(n) \end{cases}$$

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Résolution : diviser pour régner

#### ④ Calcul de la complexité

Pour résoudre un problème de taille  $n$ , on doit en résoudre deux de tailles  $\lfloor \frac{n}{2} \rfloor$  et rechercher le maximum des tranches contenant l'élément  $t_k$ . Cette opération a une complexité linéaire, on a donc :

$$\begin{cases} C(0) & \in O(1) \\ C(2n) & = 2C(n) + O(n) \end{cases}$$

On retrouve les mêmes équations de complexité que dans le cas du tri fusion. Et donc la complexité est la même :  $O(n \log n)$ .

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Multiplication matricielle

Soient  $A$  et  $B$  deux matrices carrés de tailles  $n$ , on note  $C = A \times B$

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pou régner

### Multiplication matricielle

Soient  $A$  et  $B$  deux matrices carrés de tailles  $n$ , on note  $C = A \times B$

1. Rappeler l'expression de  $C_{ij}$ .

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Multiplication matricielle

Soient  $A$  et  $B$  deux matrices carrés de tailles  $n$ , on note  $C = A \times B$

- 1 Rappeler l'expression de  $C_{ij}$ .
- 2 Quelle est la complexité d'un algorithme calculant les coefficients de  $C$  en utilisant l'expression précédente ?

## Multiplication matricielle

Soient  $A$  et  $B$  deux matrices carrés de tailles  $n$ , on note  $C = A \times B$

- ① Rappeler l'expression de  $C_{ij}$ .
- ② Quelle est la complexité d'un algorithme calculant les coefficients de  $C$  en utilisant l'expression précédente ?
- ③ on sépare  $A$  et  $B$  en blocs de tailles égales :

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \text{ et } B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

Déterminer la complexité d'un algorithme qui effectue la multiplication par bloc :

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Algorithme de Strassen

L'algorithme de Strassen est une approche diviser pour régner :



# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Algorithme de Strassen

L'algorithme de Strassen est une approche diviser pour régner :

- **diviser** : on sépare  $A$  et  $B$  en blocs de tailles égales :

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \text{ et } B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

## Algorithme de Strassen

L'algorithme de Strassen est une approche diviser pour régner :

- **diviser** : on sépare  $A$  et  $B$  en blocs de tailles égales :

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \text{ et } B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

- **régner** On calcule **seulement 7** produits matriciels :

$$M_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$

$$M_2 = (A_{2,1} + A_{2,2})B_{1,1}$$

$$M_3 = A_{1,1}(B_{1,2} - B_{2,2})$$

$$M_4 = A_{2,2}(B_{2,1} - B_{1,1})$$

$$M_5 = (A_{1,1} + A_{1,2})B_{2,2}$$

$$M_6 = (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$$

$$M_7 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$

## Algorithme de Strassen

L'algorithme de Strassen est une approche diviser pour régner :

- **diviser** : on sépare  $A$  et  $B$  en blocs de tailles égales :

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \text{ et } B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

- **régner** On calcule **seulement 7** produits matriciels :

$$M_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$

$$M_2 = (A_{2,1} + A_{2,2})B_{1,1}$$

$$M_3 = A_{1,1}(B_{1,2} - B_{2,2})$$

$$M_4 = A_{2,2}(B_{2,1} - B_{1,1})$$

$$M_5 = (A_{1,1} + A_{1,2})B_{2,2}$$

$$M_6 = (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$$

$$M_7 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$

- **combiner** On combine les solutions afin de construire les blocs de la matrice  $C$  :

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$

$$C_{1,2} = M_3 + M_5$$

$$C_{2,1} = M_2 + M_4$$

$$C_{2,2} = M_1 - M_2 + M_3 + M_6$$

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pou régner

### Complexité

L'équation de complexité s'écrit alors :

$$C(2n) = 7C(n) + O(n^2)$$

## Complexité

L'équation de complexité s'écrit alors :

$$C(2n) = 7C(n) + O(n^2)$$

Et on montre que  $C(n) \in O(n^{\log_2 7})$ , et  $\log_2 7 \simeq 2,807$

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pou régner

### Complexité

L'équation de complexité s'écrit alors :

$$C(2n) = 7C(n) + O(n^2)$$

Et on montre que  $C(n) \in O(n^{\log_2 7})$ , et  $\log_2 7 \simeq 2,807$

On obtient donc une complexité meilleure que l'algorithme "naïf"

## Complexité

L'équation de complexité s'écrit alors :

$$C(2n) = 7C(n) + O(n^2)$$

Et on montre que  $C(n) \in O(n^{\log_2 7})$ , et  $\log_2 7 \simeq 2,807$

On obtient donc une complexité meilleure que l'algorithme "naïf"

A noter qu'à cause des tailles respectives des *facteurs cachés* dans l'algorithme de naïf et dans l'algorithme de Strassen, ce dernier ne devient plus efficace en terme de temps de calcul que pour de grandes valeurs de  $n$ .

# C15 Décomposition en sous problèmes

## 3. Exemples résolus de la méthode diviser pour régner

### Complexité

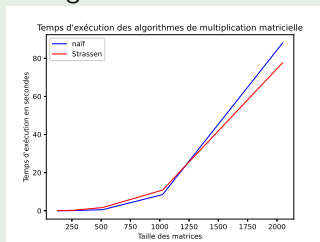
L'équation de complexité s'écrit alors :

$$C(2n) = 7C(n) + O(n^2)$$

Et on montre que  $C(n) \in O(n^{\log_2 7})$ , et  $\log_2 7 \simeq 2,807$

On obtient donc une complexité meilleure que l'algorithme "naïf"

A noter qu'à cause des tailles respectives des *facteurs cachés* dans l'algorithme de naïf et dans l'algorithme de Strassen, ce dernier ne devient plus efficace en terme de temps de calcul que pour de grandes valeurs de  $n$ .





### Exemple

- 1 Ecrire une fonction récursive *naïve* en Ocaml qui prend en argument un entier  $n$  et renvoie le  $n$ ième terme de la suite de Fibonacci définie par :

$$\begin{cases} f_0 = 1, \\ f_1 = 1, \\ f_n = f_{n-1} + f_{n-2} \text{ pour tout } n \geq 2. \end{cases}$$

### Exemple

- ① Ecrire une fonction récursive *naïve* en Ocaml qui prend en argument un entier  $n$  et renvoie le  $n$ ième terme de la suite de Fibonacci définie par :

$$\begin{cases} f_0 = 1, \\ f_1 = 1, \\ f_n = f_{n-1} + f_{n-2} \text{ pour tout } n \geq 2. \end{cases}$$

```
1 let rec fibo n =  
2   if n < 2 then 1 else fibo (n-1) + fibo (n-2);;
```

### Exemple

- 1 Ecrire une fonction récursive *naïve* en Ocaml qui prend en argument un entier  $n$  et renvoie le  $n$ ième terme de la suite de Fibonacci définie par :

$$\begin{cases} f_0 = 1, \\ f_1 = 1, \\ f_n = f_{n-1} + f_{n-2} \text{ pour tout } n \geq 2. \end{cases}$$

```
1 let rec fibo n =  
2   if n < 2 then 1 else fibo (n-1) + fibo (n-2);;
```

- 2 Tracer le graphe des appels récursifs de cette fonction pour  $n = 5$

# C15 Décomposition en sous problèmes

## 4. Rappel : mémoïsation

### Exemple

- 1 Ecrire une fonction récursive *naïve* en Ocaml qui prend en argument un entier  $n$  et renvoie le  $n$ ième terme de la suite de Fibonacci définie par :

$$\begin{cases} f_0 = 1, \\ f_1 = 1, \\ f_n = f_{n-1} + f_{n-2} \text{ pour tout } n \geq 2. \end{cases}$$

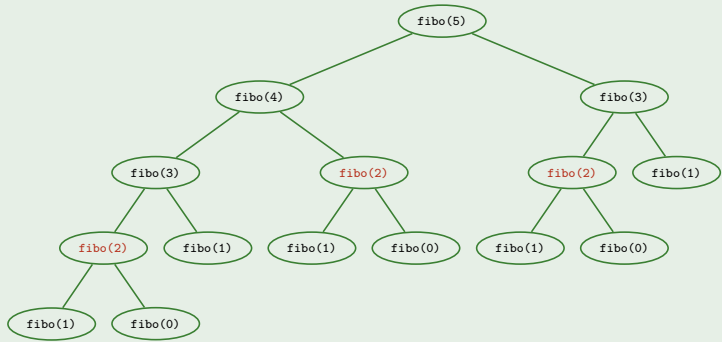
```
1 let rec fibo n =  
2   if n < 2 then 1 else fibo (n-1) + fibo (n-2);;
```

- 2 Tracer le graphe des appels récursifs de cette fonction pour  $n = 5$
- 3 Commenter

# C15 Décomposition en sous problèmes

## 4. Rappel : mémoïsation

### Exemple







### Mémoïsation

- La **mémoïsation** consiste à stocker dans une structure de données les valeurs renvoyées par une fonction afin de ne pas les recalculer lors des appels identiques suivant.



### Mémoïsation

- La **mémoïsation** consiste à stocker dans une structure de données les valeurs renvoyées par une fonction afin de ne pas les recalculer lors des appels identiques suivant.
- Les tableaux associatifs dont les clés sont les arguments de la fonction et les valeurs les résultats correspondant sont des structures de données adaptées à ce stockage car on teste l'appartenance et on retrouve une valeur efficacement.

### Mémoïsation

- La **mémoïsation** consiste à stocker dans une structure de données les valeurs renvoyées par une fonction afin de ne pas les recalculer lors des appels identiques suivant.
- Les tableaux associatifs dont les clés sont les arguments de la fonction et les valeurs les résultats correspondant sont des structures de données adaptées à ce stockage car on teste l'appartenance et on retrouve une valeur efficacement.
- On rappelle qu'un tableau associatif peut-être implémenté de façon efficace par :
  - une table de hachage
  - un arbre binaire de recherche lorsque les clés sont ordonnées.

# C15 Décomposition en sous problèmes

## 5. Programmation dynamique : exemple introductif

### Position du problème

On considère une barre de métal de longueur entière 12 et pouvant être découpée en morceaux de longueurs entières ayant chacun un prix comme indiqué ci-dessous :

longueur ( $i$ )	1	2	3	4	5	6	7	8	9	10	11	12
prix ( $p_i$ )	2	4	7	8	12	14	18	23	24	25	26	31

# C15 Décomposition en sous problèmes

## 5. Programmation dynamique : exemple introductif

### Position du problème

On considère une barre de métal de longueur entière 12 et pouvant être découpée en morceaux de longueurs entières ayant chacun un prix comme indiqué ci-dessous :

longueur ( $i$ )	1	2	3	4	5	6	7	8	9	10	11	12
prix ( $p_i$ )	2	4	7	8	12	14	18	23	24	25	26	31

Le prix de vente des différents morceaux varie donc suivant la découpe utilisée, par exemples : la découpe (2, 4, 6) a un prix de vente de  $4 + 8 + 14 = 26$ , tandis que la découpe (7, 5) a un prix de vente de  $18 + 12 = 30$

# C15 Décomposition en sous problèmes

## 5. Programmation dynamique : exemple introductif

### Position du problème

On considère une barre de métal de longueur entière 12 et pouvant être découpée en morceaux de longueurs entières ayant chacun un prix comme indiqué ci-dessous :

longueur ( $i$ )	1	2	3	4	5	6	7	8	9	10	11	12
prix ( $p_i$ )	2	4	7	8	12	14	18	23	24	25	26	31

Le prix de vente des différents morceaux varie donc suivant la découpe utilisée, par exemples : la découpe (2, 4, 6) a un prix de vente de  $4 + 8 + 14 = 26$ , tandis que la découpe (7, 5) a un prix de vente de  $18 + 12 = 30$

Le but du problème est de trouver la valeur maximale des découpes possibles.

# C15 Décomposition en sous problèmes

## 5. Programmation dynamique : exemple introductif

### Position du problème

On considère une barre de métal de longueur entière 12 et pouvant être découpée en morceaux de longueurs entières ayant chacun un prix comme indiqué ci-dessous :

longueur ( $i$ )	1	2	3	4	5	6	7	8	9	10	11	12
prix ( $p_i$ )	2	4	7	8	12	14	18	23	24	25	26	31

Le prix de vente des différents morceaux varie donc suivant la découpe utilisée, par exemples : la découpe (2, 4, 6) a un prix de vente de  $4 + 8 + 14 = 26$ , tandis que la découpe (7, 5) a un prix de vente de  $18 + 12 = 30$

**Le but du problème est de trouver la valeur maximale des découpes possibles.**

On note  $N$  la longueur de la barre,  $(v_i)_{1 \leq i \leq N}$ , la valeur maximale de la découpe d'une barre de taille  $i$  et  $(p_i)_{1 \leq i \leq N}$  le prix d'un morceaux de longueur  $i$ .

# C15 Décomposition en sous problèmes

## 5. Programmation dynamique : exemple introductif

### Position du problème

On considère une barre de métal de longueur entière 12 et pouvant être découpée en morceaux de longueurs entières ayant chacun un prix comme indiqué ci-dessous :

longueur ( $i$ )	1	2	3	4	5	6	7	8	9	10	11	12
prix ( $p_i$ )	2	4	7	8	12	14	18	23	24	25	26	31

Le prix de vente des différents morceaux varie donc suivant la découpe utilisée, par exemples : la découpe (2, 4, 6) a un prix de vente de  $4 + 8 + 14 = 26$ , tandis que la découpe (7, 5) a un prix de vente de  $18 + 12 = 30$

**Le but du problème est de trouver la valeur maximale des découpes possibles.**

On note  $N$  la longueur de la barre,  $(v_i)_{1 \leq i \leq N}$ , la valeur maximale de la découpe d'une barre de taille  $i$  et  $(p_i)_{1 \leq i \leq N}$  le prix d'un morceaux de longueur  $i$ .

- 1 Donner les valeurs de  $v_0$ ,  $v_1$ ,  $v_2$  et  $v_3$ .

# C15 Décomposition en sous problèmes

## 5. Programmation dynamique : exemple introductif

### Position du problème

On considère une barre de métal de longueur entière 12 et pouvant être découpée en morceaux de longueurs entières ayant chacun un prix comme indiqué ci-dessous :

longueur ( $i$ )	1	2	3	4	5	6	7	8	9	10	11	12
prix ( $p_i$ )	2	4	7	8	12	14	18	23	24	25	26	31

Le prix de vente des différents morceaux varie donc suivant la découpe utilisée, par exemples : la découpe (2, 4, 6) a un prix de vente de  $4 + 8 + 14 = 26$ , tandis que la découpe (7, 5) a un prix de vente de  $18 + 12 = 30$

**Le but du problème est de trouver la valeur maximale des découpes possibles.**

On note  $N$  la longueur de la barre,  $(v_i)_{1 \leq i \leq N}$ , la valeur maximale de la découpe d'une barre de taille  $i$  et  $(p_i)_{1 \leq i \leq N}$  le prix d'un morceaux de longueur  $i$ .

- 1 Donner les valeurs de  $v_0$ ,  $v_1$ ,  $v_2$  et  $v_3$ .
- 2 Etablir une relation de récurrence liant les  $(v_i)_{0 \leq i \leq N}$ .



## Position du problème

On considère une barre de métal de longueur entière 12 et pouvant être découpée en morceaux de longueurs entières ayant chacun un prix comme indiqué ci-dessous :

longueur ( $i$ )	1	2	3	4	5	6	7	8	9	10	11	12
prix ( $p_i$ )	2	4	7	8	12	14	18	23	24	25	26	31

Le prix de vente des différents morceaux varie donc suivant la découpe utilisée, par exemples : la découpe (2, 4, 6) a un prix de vente de  $4 + 8 + 14 = 26$ , tandis que la découpe (7, 5) a un prix de vente de  $18 + 12 = 30$

**Le but du problème est de trouver la valeur maximale des découpes possibles.**

On note  $N$  la longueur de la barre,  $(v_i)_{1 \leq i \leq N}$ , la valeur maximale de la découpe d'une barre de taille  $i$  et  $(p_i)_{1 \leq i \leq N}$  le prix d'un morceaux de longueur  $i$ .

- 1 Donner les valeurs de  $v_0$ ,  $v_1$ ,  $v_2$  et  $v_3$ .
- 2 Etablir une relation de récurrence liant les  $(v_i)_{0 \leq i \leq N}$ .
- 3 En déduire une fonction récursive en C calculant la valeur de la découpe maximale.

# C15 Décomposition en sous problèmes

## 5. Programmation dynamique : exemple introductif

### Position du problème

On considère une barre de métal de longueur entière 12 et pouvant être découpée en morceaux de longueurs entières ayant chacun un prix comme indiqué ci-dessous :

longueur ( $i$ )	1	2	3	4	5	6	7	8	9	10	11	12
prix ( $p_i$ )	2	4	7	8	12	14	18	23	24	25	26	31

Le prix de vente des différents morceaux varie donc suivant la découpe utilisée, par exemples : la découpe (2, 4, 6) a un prix de vente de  $4 + 8 + 14 = 26$ , tandis que la découpe (7, 5) a un prix de vente de  $18 + 12 = 30$

**Le but du problème est de trouver la valeur maximale des découpes possibles.**

On note  $N$  la longueur de la barre,  $(v_i)_{1 \leq i \leq N}$ , la valeur maximale de la découpe d'une barre de taille  $i$  et  $(p_i)_{1 \leq i \leq N}$  le prix d'un morceaux de longueur  $i$ .

- 1 Donner les valeurs de  $v_0$ ,  $v_1$ ,  $v_2$  et  $v_3$ .
- 2 Etablir une relation de récurrence liant les  $(v_i)_{0 \leq i \leq N}$ .
- 3 En déduire une fonction récursive en C calculant la valeur de la découpe maximale.
- 4 Vérifier qu'on se trouve dans une situation de chevauchement des appels récursifs et proposer une nouvelle version de votre fonction utilisant la mémoïsation.

# C15 Décomposition en sous problèmes

## 5. Programmation dynamique : exemple introductif

### Position du problème

On considère une barre de métal de longueur entière 12 et pouvant être découpée en morceaux de longueurs entières ayant chacun un prix comme indiqué ci-dessous :

longueur ( $i$ )	1	2	3	4	5	6	7	8	9	10	11	12
prix ( $p_i$ )	2	4	7	8	12	14	18	23	24	25	26	31

Le prix de vente des différents morceaux varie donc suivant la découpe utilisée, par exemples : la découpe (2, 4, 6) a un prix de vente de  $4 + 8 + 14 = 26$ , tandis que la découpe (7, 5) a un prix de vente de  $18 + 12 = 30$

**Le but du problème est de trouver la valeur maximale des découpes possibles.**

On note  $N$  la longueur de la barre,  $(v_i)_{1 \leq i \leq N}$ , la valeur maximale de la découpe d'une barre de taille  $i$  et  $(p_i)_{1 \leq i \leq N}$  le prix d'un morceaux de longueur  $i$ .

- 1 Donner les valeurs de  $v_0$ ,  $v_1$ ,  $v_2$  et  $v_3$ .
- 2 Etablir une relation de récurrence liant les  $(v_i)_{0 \leq i \leq N}$ .
- 3 En déduire une fonction récursive en C calculant la valeur de la découpe maximale.
- 4 Vérifier qu'on se trouve dans une situation de chevauchement des appels récursifs et proposer une nouvelle version de votre fonction utilisant la mémoïsation.
- 5 Etudier les complexités des deux versions.

## Résolution

①  $v_0 = 0, v_1 = 2, v_2 = 4$  et  $v_3 = 7$

## Résolution

- 1  $v_0 = 0, v_1 = 2, v_2 = 4$  et  $v_3 = 7$
- 2 En supposant qu'on connaisse les valeurs maximales de découpe pour *toutes* les tailles inférieures à  $n$ , la découpe maximale pour la taille  $n$  s'en déduit en prenant le maximum parmi les découpes maximales d'une barre de longueur  $n - k \leq n - 1$  et du prix d'un morceau de taille  $k$ , c'est à dire :  $v_n = \max \{v_{n-k} + p_k, 1 \leq k \leq n\}$

## Résolution

- 1  $v_0 = 0, v_1 = 2, v_2 = 4$  et  $v_3 = 7$
- 2 En supposant qu'on connaisse les valeurs maximales de découpe pour *toutes* les tailles inférieures à  $n$ , la découpe maximale pour la taille  $n$  s'en déduit en prenant le maximum parmi les découpes maximales d'une barre de longueur  $n - k \leq n - 1$  et du prix d'un morceau de taille  $k$ , c'est à dire :  $v_n = \max \{v_{n-k} + p_k, 1 \leq k \leq n\}$
- 3 Implémentation en C :

```
1 // Découpe maximale d'une barre de taille n
2 int vmax(int barre[], int n) {
3     int cmax = 0;
4     int vnk;
5     if (n == 0) {return 0;}
6     for (int k = 1; k <= n; k++){
7         vnk = vmax(barre, n-k);
8         if (barre[k] + vnk > cmax) {cmax = vnk + barre[k];}
9     }
10    return cmax;}

```

## Résolution

- ④ Pour calculer  $v_5$ , on doit calculer  $v_4, v_3, \dots, v_0$ . Mais l'appel à  $v_4$  demande aussi le calcul de  $v_3, v_2, \dots, v_0$ . On se trouve donc bien dans le cas d'un chevauchement d'appels récursifs.

On peut proposer la version avec memoisation suivante :

```
1 // Avec mémoisation (v[n]=-1 indique une valeur non encore calculée)
2 int vmax_memo(int barre[], int n, int v[]){
3     if (v[n]!=-1) {return v[n];}
4     int cmax = 0;
5     int vnk;
6     for (int k = 1; k <= n; k++){
7         vnk = vmax_memo(barre, n-k, v);
8         if (barre[k] + vnk > cmax) {cmax = vnk + barre[k];}
9     }
10    v[n] = cmax;
11    return cmax;}
```

## Résolution

- 5 On peut construire l'arbre des appels récursifs pour  $n = 5$  :

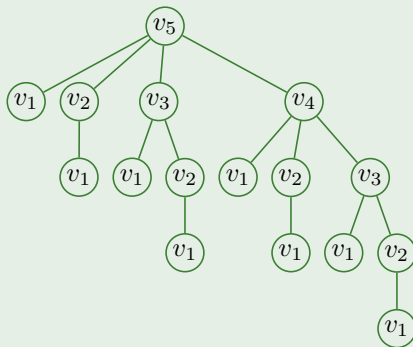


# C15 Décomposition en sous problèmes

## 5. Programmation dynamique : exemple introductif

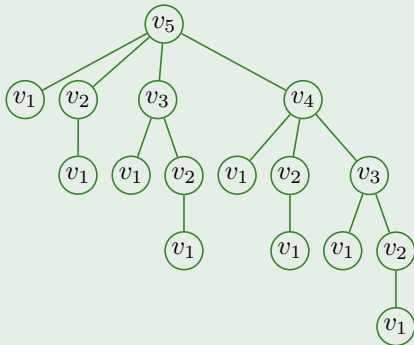
### Résolution

- 5 On peut construire l'arbre des appels récursifs pour  $n = 5$  :



## Résolution

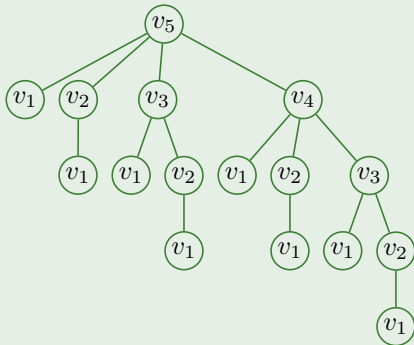
- 5 On peut construire l'arbre des appels récursifs pour  $n = 5$  :



En notant  $a_n$  le nombre d'appels pour calculer  $v_n$ , on a  $a_n = 1 + \sum_{k=0}^{n-1} a_k$ .

## Résolution

- 5 On peut construire l'arbre des appels récursifs pour  $n = 5$  :



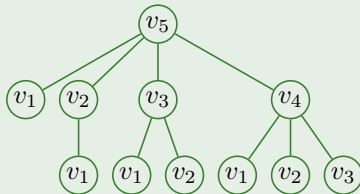
En notant  $a_n$  le nombre d'appels pour calculer  $v_n$ , on a  $a_n = 1 + \sum_{k=0}^{n-1} a_k$ . On obtient  $a_n = 2^n$  et donc la complexité est au moins quadratique!

### Résolution

- ⑤ Dans le cas de la mémorisation, les appels récursifs déjà calculés sont obtenus directement, ce qui donne l'arbre d'appel suivant :

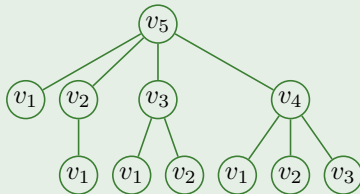
## Résolution

- 5 Dans le cas de la mémorisation, les appels récursifs déjà calculés sont obtenus directement, ce qui donne l'arbre d'appel suivant :



## Résolution

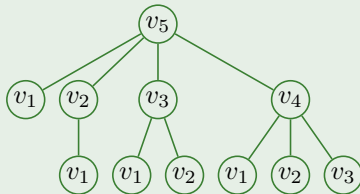
- ⑤ Dans le cas de la mémorisation, les appels récursifs déjà calculés sont obtenus directement, ce qui donne l'arbre d'appel suivant :



En notant  $b_n$  le nombre d'appels pour calculer  $v_n$ , on a  $b_n = \sum_{k=0}^{n-1} k$ .

## Résolution

- 5 Dans le cas de la mémorisation, les appels récursifs déjà calculés sont obtenus directement, ce qui donne l'arbre d'appel suivant :



En notant  $b_n$  le nombre d'appels pour calculer  $v_n$ , on a  $b_n = \sum_{k=0}^{n-1} k$ . La complexité est donc quadratique.

## Calcul de bas en haut (*bottom up*)

La mémorisation construit la solution de façon "descendante", on lance les appels récursif sur les plus grandes valeurs de taille de la barre. Une autre stratégie dite **ascendante** ou **de bas en haut** (*bottom up*) consiste à construire la solution en partant des instances les plus petites du problème.



## Calcul de bas en haut (*bottom up*)

La mémorisation construit la solution de façon "descendante", on lance les appels récursif sur les plus grandes valeurs de taille de la barre. Une autre stratégie dite **ascendante** ou **de bas en haut** (*bottom up*) consiste à construire la solution en partant des instances les plus petites du problème.

Pour la découpe de la barre on part donc des valeurs connues  $v_0$  et  $v_1$  et on construit  $v_2$  puis  $v_3$ , en utilisant la relation de récurrence  $v_n = \max \{v_{n-k} + p_k, 1 \leq k \leq n\}$

## Calcul de bas en haut (*bottom up*)

La mémorisation construit la solution de façon "descendante", on lance les appels récursif sur les plus grandes valeurs de taille de la barre. Une autre stratégie dite **ascendante** ou **de bas en haut** (*bottom up*) consiste à construire la solution en partant des instances les plus petites du problème.

Pour la découpe de la barre on part donc des valeurs connues  $v_0$  et  $v_1$  et on construit  $v_2$  puis  $v_3$ , en utilisant la relation de récurrence  $v_n = \max \{v_{n-k} + p_k, 1 \leq k \leq n\}$

Ce qui se traduit par une solution **itérative** :

# C15 Décomposition en sous problèmes

## 5. Programmation dynamique : exemple introductif

### Calcul de bas en haut (*bottom up*)

La mémorisation construit la solution de façon "descendante", on lance les appels récursifs sur les plus grandes valeurs de taille de la barre. Une autre stratégie dite **ascendante** ou **de bas en haut** (*bottom up*) consiste à construire la solution en partant des instances les plus petites du problème.

Pour la découpe de la barre on part donc des valeurs connues  $v_0$  et  $v_1$  et on construit  $v_2$  puis  $v_3$ , en utilisant la relation de récurrence  $v_n = \max \{v_{n-k} + p_k, 1 \leq k \leq n\}$

Ce qui se traduit par une solution **itérative** :

```
1  int vmax_iter(int barre[], int n){
2      int vmax[n];
3      vmax[0] = 0;
4      vmax[1] = barre[1];
5      for (int i=2; i<=n; i++){
6          vmax[i] = 0;
7          for (int j=1; j<=i; j++){
8              if (vmax[j]+barre[i-j]>vmax[i])
9                  {vmax[i] = vmax[j]+barre[i-j];}}
10     return vmax[n];}
```

### Construction d'une solution

On a pour le moment déterminé la valeur maximale de la découpe, mais pas la découpe elle-même. D'autre part, plusieurs découpes différentes peuvent avoir cette même valeur maximale. Pour rechercher *une* découpe de valeur maximale, on peut par exemple :

- construire le tableau  $(v_k)_{0 \leq k \leq N}$  et l'utiliser afin d'en déduire la découpe.

## Construction d'une solution

On a pour le moment déterminé la valeur maximale de la découpe, mais pas la découpe elle-même. D'autre part, plusieurs découpes différentes peuvent avoir cette même valeur maximale. Pour rechercher *une* découpe de valeur maximale, on peut par exemple :

- construire le tableau  $(v_k)_{0 \leq k \leq N}$  et l'utiliser afin d'en déduire la découpe.  
Par exemple, si  $v_{12} = v_8 + p_4$ , cela signifie que pour avoir la valeur maximale de la découpe d'une barre de taille 12, une possibilité est d'utiliser une découpe maximale d'une barre de taille 8 et un morceau de taille 4. En remontant ainsi de proche en proche, on obtient une découpe maximale possible

## Construction d'une solution

On a pour le moment déterminé la valeur maximale de la découpe, mais pas la découpe elle-même. D'autre part, plusieurs coupes différentes peuvent avoir cette même valeur maximale. Pour rechercher *une* découpe de valeur maximale, on peut par exemple :

- construire le tableau  $(v_k)_{0 \leq k \leq N}$  et l'utiliser afin d'en déduire la découpe.  
Par exemple, si  $v_{12} = v_8 + p_4$ , cela signifie que pour avoir la valeur maximale de la découpe d'une barre de taille 12, une possibilité est d'utiliser une découpe maximale d'une barre de taille 8 et un morceau de taille 4. En remontant ainsi de proche en proche, on obtient une découpe maximale possible
- Modifier notre fonction afin qu'elle renvoie la découpe maximale et non pas la valeur de cette découpe.

### Construction d'une solution

On a pour le moment déterminé la valeur maximale de la découpe, mais pas la découpe elle-même. D'autre part, plusieurs découpes différentes peuvent avoir cette même valeur maximale. Pour rechercher *une* découpe de valeur maximale, on peut par exemple :

- construire le tableau  $(v_k)_{0 \leq k \leq N}$  et l'utiliser afin d'en déduire la découpe.  
Par exemple, si  $v_{12} = v_8 + p_4$ , cela signifie que pour avoir la valeur maximale de la découpe d'une barre de taille 12, une possibilité est d'utiliser une découpe maximale d'une barre de taille 8 et un morceau de taille 4. En remontant ainsi de proche en proche, on obtient une découpe maximale possible
- Modifier notre fonction afin qu'elle renvoie la découpe maximale et non pas la valeur de cette découpe.

Ces deux possibilités seront abordées en TP.

### Principes généraux

La programmation dynamique s'applique généralement à la résolution d'un problème d'optimisation vérifiant les conditions suivantes :



### Principes généraux

La programmation dynamique s'applique généralement à la résolution d'un problème d'optimisation vérifiant les conditions suivantes :

- 1 **Sous structure optimale** : ce problème peut-être résolu à partir de problèmes similaires mais plus petits

### Principes généraux

La programmation dynamique s'applique généralement à la résolution d'un problème d'optimisation vérifiant les conditions suivantes :

- 1 **Sous structure optimale** : ce problème peut-être résolu à partir de problèmes similaires mais plus petits

La découpe maximale d'une barre de taille  $N$  s'obtient comme découpe maximale d'une barre de taille strictement inférieure  $k$  et d'un morceau de taille  $N - k$ .

### Principes généraux

La programmation dynamique s'applique généralement à la résolution d'un problème d'optimisation vérifiant les conditions suivantes :

- 1 **Sous structure optimale** : ce problème peut-être résolu à partir de problèmes similaires mais plus petits  
La découpe maximale d'une barre de taille  $N$  s'obtient comme découpe maximale d'une barre de taille strictement inférieure  $k$  et d'un morceau de taille  $N - k$ .
- 2 **Chevauchement de sous problème** : une solution récursive produit des appels identiques. Pour pallier ce problème, on utilise la mémoïsation dans les solutions récursives ou une solution de bas en haut itérative.

### Principes généraux

La programmation dynamique s'applique généralement à la résolution d'un problème d'optimisation vérifiant les conditions suivantes :

- 1 **Sous structure optimale** : ce problème peut-être résolu à partir de problèmes similaires mais plus petits

La découpe maximale d'une barre de taille  $N$  s'obtient comme découpe maximale d'une barre de taille strictement inférieure  $k$  et d'un morceau de taille  $N - k$ .

- 2 **Chevauchement de sous problème** : une solution récursive produit des appels identiques. Pour pallier ce problème, on utilise la mémoïsation dans les solutions récursives ou une solution de bas en haut itérative.

Pour rechercher la découpe maximale d'un barre de taille 5, on est amené à chercher celle d'une barre de taille 4,3,2,1. Et pour chercher celle d'une barre de taille 4, on fera de nouveau appel à celle d'une barre de taille 3,2,1 ...

### Principes généraux

La programmation dynamique s'applique généralement à la résolution d'un problème d'optimisation vérifiant les conditions suivantes :

- 1 **Sous structure optimale** : ce problème peut-être résolu à partir de problèmes similaires mais plus petits

La découpe maximale d'une barre de taille  $N$  s'obtient comme découpe maximale d'une barre de taille strictement inférieure  $k$  et d'un morceau de taille  $N - k$ .

- 2 **Chevauchement de sous problème** : une solution récursive produit des appels identiques. Pour pallier ce problème, on utilise la mémoïsation dans les solutions récursives ou une solution de bas en haut itérative.

Pour rechercher la découpe maximale d'un barre de taille 5, on est amené à chercher celle d'une barre de taille 4,3,2,1. Et pour chercher celle d'une barre de taille 4, on fera de nouveau appel à celle d'une barre de taille 3,2,1 ...

- ! L'étape cruciale est de déterminer les relations de récurrence entre les différentes instances du problème.

### Sous structure optimale : contre-exemples

On donne ici deux exemples de problèmes n'ayant *pas* la propriété de **sous structure optimale** :

### Sous structure optimale : contre-exemples

On donne ici deux exemples de problèmes n'ayant *pas* la propriété de **sous structure optimale** :

- Nombre minimal de multiplications dans le calcul de  $a^n$

### Sous structure optimale : contre-exemples

On donne ici deux exemples de problèmes n'ayant *pas* la propriété de **sous structure optimale** :

- Nombre minimal de multiplications dans le calcul de  $a^n$

Par exemple le calcul optimal de  $a^{15}$  demande 5 multiplications :

$$a^{15} = a^3 \times ((a^3)^2)^2$$

$a^6$  doit donc être calculé en utilisant le calcul de  $a^3$  au lieu de par exemple

$a^6 = (a^2)^3$  qui demande aussi 3 multiplications. C'est à dire qu'on peut avoir trouvé une solution optimale  $n = 6$  mais qui n'intervient *pas* dans le calcul pour  $n = 15$ .



### Sous structure optimale : contre-exemples

On donne ici deux exemples de problèmes n'ayant *pas* la propriété de **sous structure optimale** :

- Nombre minimal de multiplications dans le calcul de  $a^n$

Par exemple le calcul optimal de  $a^{15}$  demande 5 multiplications :

$$a^{15} = a^3 \times ((a^3)^2)^2$$

$a^6$  doit donc être calculé en utilisant le calcul de  $a^3$  au lieu de par exemple

$a^6 = (a^2)^3$  qui demande aussi 3 multiplications. C'est à dire qu'on peut avoir trouvé une solution optimale  $n = 6$  mais qui n'intervient *pas* dans le calcul pour  $n = 15$ .

- Recherche du plus long chemin élémentaire dans un graphe

## Sous structure optimale : contre-exemples

On donne ici deux exemples de problèmes n'ayant *pas* la propriété de **sous structure optimale** :

- Nombre minimal de multiplications dans le calcul de  $a^n$

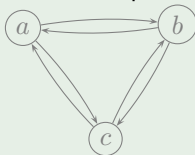
Par exemple le calcul optimal de  $a^{15}$  demande 5 multiplications :

$$a^{15} = a^3 \times ((a^3)^2)^2$$

$a^6$  doit donc être calculé en utilisant le calcul de  $a^3$  au lieu de par exemple

$a^6 = (a^2)^3$  qui demande aussi 3 multiplications. C'est à dire qu'on peut avoir trouvé une solution optimale  $n = 6$  mais qui n'intervient *pas* dans le calcul pour  $n = 15$ .

- Recherche du plus long chemin élémentaire dans un graphe



Dans le graphe ci-contre, le plus long chemin élémentaire de  $a$  vers  $c$  est  $a \rightarrow b \rightarrow c$  et son sous-chemin  $a \rightarrow b$  n'est pas une la solution du sous problème consistant à rechercher le plus long chemin élémentaire de  $a$  vers  $b$ . Et de même pour son sous chemin  $b \rightarrow c$ .

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Position du problème

On considère deux chaînes de caractères  $u$  et  $v$  de longueurs respectives  $n$  et  $m$ .  
On cherche à déterminer la longueur de leur plus longue sous séquence commune (PLSSC), c'est à dire la chaîne  $w$  telle que :

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Position du problème

On considère deux chaînes de caractères  $u$  et  $v$  de longueurs respectives  $n$  et  $m$ . On cherche à déterminer la longueur de leur plus longue sous séquence commune (PLSSC), c'est à dire la chaîne  $w$  telle que :

- $w$  est une sous séquence (c'est à dire une suite extraite) de  $u$ ,

## Position du problème

On considère deux chaînes de caractères  $u$  et  $v$  de longueurs respectives  $n$  et  $m$ . On cherche à déterminer la longueur de leur plus longue sous séquence commune (PLSSC), c'est à dire la chaîne  $w$  telle que :

- $w$  est une sous séquence (c'est à dire une suite extraite) de  $u$ ,
- $w$  est une sous séquence de  $v$ ,

## Position du problème

On considère deux chaînes de caractères  $u$  et  $v$  de longueurs respectives  $n$  et  $m$ . On cherche à déterminer la longueur de leur plus longue sous séquence commune (PLSSC), c'est à dire la chaîne  $w$  telle que :

- $w$  est une sous séquence (c'est à dire une suite extraite) de  $u$ ,
- $w$  est une sous séquence de  $v$ ,
- $w$  est de longueur maximale.

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Position du problème

On considère deux chaînes de caractères  $u$  et  $v$  de longueurs respectives  $n$  et  $m$ . On cherche à déterminer la longueur de leur plus longue sous séquence commune (PLSSC), c'est à dire la chaîne  $w$  telle que :

- $w$  est une sous séquence (c'est à dire une suite extraite) de  $u$ ,
- $w$  est une sous séquence de  $v$ ,
- $w$  est de longueur maximale.

Par exemple,  $u$ ="PROGRAMMATION" et  $v$ ="DYNAMIQUE" ont comme sous séquence commune "AMI" (et c'est la plus longue)

## 7. Exemple résolu : plus longue sous séquence commune

## Position du problème

On considère deux chaînes de caractères  $u$  et  $v$  de longueurs respectives  $n$  et  $m$ . On cherche à déterminer la longueur de leur plus longue sous séquence commune (PLSSC), c'est à dire la chaîne  $w$  telle que :

- $w$  est une sous séquence (c'est à dire une suite extraite) de  $u$ ,
- $w$  est une sous séquence de  $v$ ,
- $w$  est de longueur maximale.

Par exemple,  $u = \text{"PROGRAMMATION"}$  et  $v = \text{"DYNAMIQUE"}$  ont comme sous séquence commune "AMI" (et c'est la plus longue)

- PROGRAMMATION



## 7. Exemple résolu : plus longue sous séquence commune

## Position du problème

On considère deux chaînes de caractères  $u$  et  $v$  de longueurs respectives  $n$  et  $m$ . On cherche à déterminer la longueur de leur plus longue sous séquence commune (PLSSC), c'est à dire la chaîne  $w$  telle que :

- $w$  est une sous séquence (c'est à dire une suite extraite) de  $u$ ,
- $w$  est une sous séquence de  $v$ ,
- $w$  est de longueur maximale.

Par exemple,  $u$ ="PROGRAMMATION" et  $v$ ="DYNAMIQUE" ont comme sous séquence commune "AMI" (et c'est la plus longue)

- PROGRAMMATION
- DYNAMIQUE

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Position du problème

On considère deux chaînes de caractères  $u$  et  $v$  de longueurs respectives  $n$  et  $m$ . On cherche à déterminer la longueur de leur plus longue sous séquence commune (PLSSC), c'est à dire la chaîne  $w$  telle que :

- $w$  est une sous séquence (c'est à dire une suite extraite) de  $u$ ,
- $w$  est une sous séquence de  $v$ ,
- $w$  est de longueur maximale.

Par exemple,  $u = \text{"PROGRAMMATION"}$  et  $v = \text{"DYNAMIQUE"}$  ont comme sous séquence commune "AMI" (et c'est la plus longue)

- PROGRAMMATION
- DYNAMIQUE

Donc ici, la longueur de la PLSSC est 3.

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Résolution

On cherche les relations de récurrence entre des instances du sous-problème. Pour cela on note  $u_i$  ( $0 \leq i \leq n$ ) la chaîne composée des  $i$  premiers caractères de  $u$ , et  $v_j$  ( $0 \leq j \leq m$ ) celle composée des  $j$  premiers caractères de  $v$ . Et on note  $\text{lplssc}(u_i, v_j)$  la longueur de la PLSSC de  $u_i$  et de  $v_j$ .

## Résolution

On cherche les relations de récurrence entre des instances du sous-problème. Pour cela on note  $u_i$  ( $0 \leq i \leq n$ ) la chaîne composée des  $i$  premiers caractères de  $u$ , et  $v_j$  ( $0 \leq j \leq m$ ) celle composée des  $j$  premiers caractères de  $v$ . Et on note  $\text{lplssc}(u_i, v_j)$  la longueur de la PLSSC de  $u_i$  et de  $v_j$ .

- Si  $u[i] = v[j]$  alors, quelle est la relation entre  $\text{lplssc}(u_i, v_j)$  et  $\text{lplssc}(u_{i-1}, v_{j-1})$  ?

## Résolution

On cherche les relations de récurrence entre des instances du sous-problème. Pour cela on note  $u_i$  ( $0 \leq i \leq n$ ) la chaîne composée des  $i$  premiers caractères de  $u$ , et  $v_j$  ( $0 \leq j \leq m$ ) celle composée des  $j$  premiers caractères de  $v$ . Et on note  $\text{lplssc}(u_i, v_j)$  la longueur de la PLSSC de  $u_i$  et de  $v_j$ .

- Si  $u[i] = v[j]$  alors, quelle est la relation entre  $\text{lplssc}(u_i, v_j)$  et  $\text{lplssc}(u_{i-1}, v_{j-1})$  ?
- Sinon, exprimer  $\text{plssc}(u_i, v_j)$  en fonction de  $\text{plssc}(u_i, v_{j-1})$  et  $\text{plssc}(u_{i-1}, v_j)$

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Résolution

On cherche les relations de récurrence entre des instances du sous-problème. Pour cela on note  $u_i$  ( $0 \leq i \leq n$ ) la chaîne composée des  $i$  premiers caractères de  $u$ , et  $v_j$  ( $0 \leq j \leq m$ ) celle composée des  $j$  premiers caractères de  $v$ . Et on note  $\text{lplssc}(u_i, v_j)$  la longueur de la PLSSC de  $u_i$  et de  $v_j$ .

- Si  $u[i] = v[j]$  alors, quelle est la relation entre  $\text{lplssc}(u_i, v_j)$  et  $\text{lplssc}(u_{i-1}, v_{j-1})$  ?
- Sinon, exprimer  $\text{plssc}(u_i, v_j)$  en fonction de  $\text{plssc}(u_i, v_{j-1})$  et  $\text{plssc}(u_{i-1}, v_j)$
- Déterminer les cas de base (ceux où  $u$  et  $v$  sont des chaînes vides notés  $\epsilon$ ) :

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Résolution

On cherche les relations de récurrence entre des instances du sous-problème. Pour cela on note  $u_i$  ( $0 \leq i \leq n$ ) la chaîne composée des  $i$  premiers caractères de  $u$ , et  $v_j$  ( $0 \leq j \leq m$ ) celle composée des  $j$  premiers caractères de  $v$ . Et on note  $\text{lplssc}(u_i, v_j)$  la longueur de la PLSSC de  $u_i$  et de  $v_j$ .

- Si  $u[i] = v[j]$  alors, quelle est la relation entre  $\text{lplssc}(u_i, v_j)$  et  $\text{lplssc}(u_{i-1}, v_{j-1})$  ?  
 $\text{lplssc}(u_i, v_j) = 1 + \text{lplssc}(u_{i-1}, v_{j-1})$
- Sinon, exprimer  $\text{plssc}(u_i, v_j)$  en fonction de  $\text{plssc}(u_i, v_{j-1})$  et  $\text{plssc}(u_{i-1}, v_j)$
- Déterminer les cas de base (ceux où  $u$  et  $v$  sont des chaînes vides notés  $\epsilon$ ) :

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Résolution

On cherche les relations de récurrence entre des instances du sous-problème. Pour cela on note  $u_i$  ( $0 \leq i \leq n$ ) la chaîne composée des  $i$  premiers caractères de  $u$ , et  $v_j$  ( $0 \leq j \leq m$ ) celle composée des  $j$  premiers caractères de  $v$ . Et on note  $lplssc(u_i, v_j)$  la longueur de la PLSSC de  $u_i$  et de  $v_j$ .

- Si  $u[i] = v[j]$  alors, quelle est la relation entre  $lplssc(u_i, v_j)$  et  $lplssc(u_{i-1}, v_{j-1})$  ?  
 $lplssc(u_i, v_j) = 1 + lplssc(u_{i-1}, v_{j-1})$
- Sinon, exprimer  $plssc(u_i, v_j)$  en fonction de  $plssc(u_i, v_{j-1})$  et  $plssc(u_{i-1}, v_j)$   
 $lplssc(u_i, v_j) = \max(lplssc(u_i, v_{j-1}), lplssc(u_{i-1}, v_j))$
- Déterminer les cas de base (ceux où  $u$  et  $v$  sont des chaînes vides notés  $\epsilon$ ) :



## 7. Exemple résolu : plus longue sous séquence commune

## Résolution

On cherche les relations de récurrence entre des instances du sous-problème. Pour cela on note  $u_i$  ( $0 \leq i \leq n$ ) la chaîne composée des  $i$  premiers caractères de  $u$ , et  $v_j$  ( $0 \leq j \leq m$ ) celle composée des  $j$  premiers caractères de  $v$ . Et on note  $\text{lplssc}(u_i, v_j)$  la longueur de la PLSSC de  $u_i$  et de  $v_j$ .

- Si  $u[i] = v[j]$  alors, quelle est la relation entre  $\text{lplssc}(u_i, v_j)$  et  $\text{lplssc}(u_{i-1}, v_{j-1})$  ?  

$$\text{lplssc}(u_i, v_j) = 1 + \text{lplssc}(u_{i-1}, v_{j-1})$$
- Sinon, exprimer  $\text{plssc}(u_i, v_j)$  en fonction de  $\text{plssc}(u_i, v_{j-1})$  et  $\text{plssc}(u_{i-1}, v_j)$   

$$\text{lplssc}(u_i, v_j) = \max(\text{lplssc}(u_i, v_{j-1}), \text{lplssc}(u_{i-1}, v_j))$$
- Déterminer les cas de base (ceux où  $u$  et  $v$  sont des chaînes vides notés  $\epsilon$ ) :  

$$\text{lplssc}(u_i, \epsilon) = 0$$

$$\text{lplssc}(\epsilon, v_j) = 0$$

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Implémentation en OCaml

On doit donc écrire une fonction `lplssc string -> string -> int` qui prend en argument deux chaînes de caractères `u` et `v` et renvoie la longueur de leur plus longue sous séquence commune.

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Implémentation en OCaml

On doit donc écrire une fonction `lplssc string -> string -> int` qui prend en argument deux chaînes de caractères `u` et `v` et renvoie la longueur de leur plus longue sous séquence commune.

🌀 Comme on travaille récursivement sur la longueur des préfixes on pourra écrire une fonction auxiliaire aux `string -> string -> int -> int -> int` qui prend deux entiers supplémentaires en arguments : les longueurs de chacune des deux chaînes

## Implémentation en OCaml

On doit donc écrire une fonction `lplssc string -> string -> int` qui prend en argument deux chaînes de caractères `u` et `v` et renvoie la longueur de leur plus longue sous séquence commune.

🌀 Comme on travaille récursivement sur la longueur des préfixes on pourra écrire une fonction auxiliaire aux `string -> string -> int -> int -> int` qui prend deux entiers supplémentaires en arguments : les longueurs de chacune des deux chaînes

```
1 let lplssc u v =  
2   let rec aux u v n m =  
3     if n=0 || m=0 then 0 else  
4     if u.[n-1] = v.[m-1] then  
5       1 + aux u v (n-1) (m-1) else  
6       max (aux u v (n-1) m) (aux u v n (m-1)) in  
7   aux u v (String.length u) (String.length v);;
```

## Mémoïsation

Modifier la fonction précédente afin de mémoriser les résultats déjà calculés. On pourra utiliser une matrice `memo` créée en OCaml avec

```
let memo = Array.make_matrix (n+1) (m+1) (-1)
```

La valeur initiale  $-1$  indiquant que la LPLSSC de  $u_i$  et  $v_j$  n'a pas encore été calculée.

## Mémoïsation

Modifier la fonction précédente afin de mémoriser les résultats déjà calculés. On pourra utiliser une matrice `memo` créée en OCaml avec

```
let memo = Array.make_matrix (n+1) (m+1) (-1)
```

La valeur initiale  $-1$  indiquant que la LPLSSC de  $u_i$  et  $v_j$  n'a pas encore été calculée.

## 7. Exemple résolu : plus longue sous séquence commune

## Mémoïsation

Modifier la fonction précédente afin de mémoriser les résultats déjà calculés. On pourra utiliser une matrice memo créée en OCaml avec

```
let memo = Array.make_matrix (n+1) (m+1) (-1)
```

La valeur initiale  $-1$  indiquant que la LPLSSC de  $u_i$  et  $v_j$  n'a pas encore été calculée.

```

1  let lplssc_memo u v memo =
2      let rec aux u v n m memo=
3          if memo.(n).(m) <> -1 then memo.(n).(m) else
4              (if n=0 || m=0 then
5                  (memo.(n).(m) <- 0; 0)
6                  else
7                      (if u.[n-1] = v.[m-1] then
8                          (let r1 = 1 + aux u v (n-1) (m-1) memo in
9                              memo.(n).(m)<-r1; r1) else
10                             (let r2 = max (aux u v (n-1) m memo) (aux u v n (m-1) memo) in
11                                 memo.(n).(m)<-r2; r2))
12              ) in
13      aux u v (String.length u) (String.length v) memo;;

```

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

Version itérative



## Version itérative

```
1 let lplssc_iter u v memo =
2   let n = String.length u in
3   let m = String.length v in
4   for i = 0 to n do
5     memo.(i).(0) <- 0;
6   done;
7   for i = 0 to m do
8     memo.(0).(i) <- 0;
9   done;
10  for i=1 to n do
11    for j=1 to m do
12      if u.[i-1]=v.[j-1] then
13        memo.(i).(j) <- 1 + memo.(i-1).(j-1)
14      else
15        memo.(i).(j) <- max memo.(i-1).(j) memo.(i).(j-1)
16      done;
17    done;
18  memo.(n).(m);;
```

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	€	E	M	P	I	R	E
€	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	2	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	3	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	€	E	M	P	I	R	E
€	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	2	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	3	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	€	E	M	P	I	R	E
€	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	2	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	3	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	2	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	3	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	2	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	3	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale
- sinon on va à gauche ou en haut suivant la case qui a plus grande valeur

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	2	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	3	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale
- sinon on va à gauche ou en haut suivant la case qui a plus grande valeur

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	2	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	3	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale
- sinon on va à gauche ou en haut suivant la case qui a plus grande valeur



# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	2	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	3	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale
- sinon on va à gauche ou en haut suivant la case qui a plus grande valeur

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	2	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	3	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale
- sinon on va à gauche ou en haut suivant la case qui a plus grande valeur

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	2	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	<u>3</u>	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale
- sinon on va à gauche ou en haut suivant la case qui a plus grande valeur

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	2	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	<u>3</u>	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale
- sinon on va à gauche ou en haut suivant la case qui a plus grande valeur

## 7. Exemple résolu : plus longue sous séquence commune

## Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	2	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	<u>3</u>	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale
- sinon on va à gauche ou en haut suivant la case qui a plus grande valeur

## 7. Exemple résolu : plus longue sous séquence commune

## Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	2	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	<u>3</u>	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale
- sinon on va à gauche ou en haut suivant la case qui a plus grande valeur

## 7. Exemple résolu : plus longue sous séquence commune

## Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	<u>2</u>	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	<u>3</u>	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale
- sinon on va à gauche ou en haut suivant la case qui a plus grande valeur

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	1	1	1	1	1
P	0	0	1	<u>2</u>	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	<u>3</u>	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale
- sinon on va à gauche ou en haut suivant la case qui a plus grande valeur



# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	<u>1</u>	1	1	1	1
P	0	0	1	<u>2</u>	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	<u>3</u>	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale
- sinon on va à gauche ou en haut suivant la case qui a plus grande valeur

## 7. Exemple résolu : plus longue sous séquence commune

## Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	<u>1</u>	1	1	1	1
P	0	0	1	<u>2</u>	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	<u>3</u>	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale
- sinon on va à gauche ou en haut suivant la case qui a plus grande valeur

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

La matrice obtenue pour les mots : IMPERIAL et EMPIRE est :

	ε	E	M	P	I	R	E
ε	0	0	0	0	0	0	0
I	0	0	0	0	1	1	1
M	0	0	<u>1</u>	1	1	1	1
P	0	0	1	<u>2</u>	2	2	2
E	0	1	1	2	2	2	3
R	0	1	1	2	2	3	3
I	0	1	1	2	<u>3</u>	3	3
A	0	1	1	2	3	3	3
L	0	1	1	2	3	3	3

On peut utiliser ce résultat pour construire la PLSSC :

- on part de la dernière case en bas et à droite
- si les lettres sur la ligne et la colonne sont identiques on ajoute à la PLSSC et on remonte en diagonale
- sinon on va à gauche ou en haut suivant la case qui a plus grande valeur

# C15 Décomposition en sous problèmes

## 7. Exemple résolu : plus longue sous séquence commune

### Reconstruction de la solution

## Reconstruction de la solution

```
1 let reconstruit u v memo =
2   let n = ref (String.length u) in
3   let m = ref (String.length v) in
4   let res = ref "" in
5   while (!n <> 0) && (!m <> 0) do
6     if (u.[!n-1]=v.[!m-1] && memo.(!n).(!m) = 1 + memo.(!n-1).(!m-1)) then
7       (res := (String.make 1 u.[!n-1]) ^ !res ;
8        n := !n -1;
9        m := !m -1;)
10    else
11      ( if (memo.(!n-1).(!m) > memo.(!n).(!m-1)) then
12        (n := !n -1;)
13        else
14          (m := !m -1;))
15  done;
16  !res;;
```

# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Position du problème

On dispose d'un *système monétaire* c'est à dire d'un ensemble de valeurs possibles pour les pièces et les billets. Le problème du rendu de monnaie consiste à déterminer le nombre minimal de pièces à utiliser pour former une somme donnée.

# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Position du problème

On dispose d'un *système monétaire* c'est à dire d'un ensemble de valeurs possibles pour les pièces et les billets. Le problème du rendu de monnaie consiste à déterminer le nombre minimal de pièces à utiliser pour former une somme donnée. Par exemple si le système monétaire est  $\{1, 3, 4, 5\}$  et la somme 7,

# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Position du problème

On dispose d'un *système monétaire* c'est à dire d'un ensemble de valeurs possibles pour les pièces et les billets. Le problème du rendu de monnaie consiste à déterminer le nombre minimal de pièces à utiliser pour former une somme donnée. Par exemple si le système monétaire est  $\{1, 3, 4, 5\}$  et la somme 7, alors on peut utiliser au minimum 2 pièces ( $4 + 3$ ).



# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Position du problème

On dispose d'un *système monétaire* c'est à dire d'un ensemble de valeurs possibles pour les pièces et les billets. Le problème du rendu de monnaie consiste à déterminer le nombre minimal de pièces à utiliser pour former une somme donnée. Par exemple si le système monétaire est  $\{1, 3, 4, 5\}$  et la somme 7, alors on peut utiliser au minimum 2 pièces ( $4 + 3$ ).

# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Position du problème

On dispose d'un *système monétaire* c'est à dire d'un ensemble de valeurs possibles pour les pièces et les billets. Le problème du rendu de monnaie consiste à déterminer le nombre minimal de pièces à utiliser pour former une somme donnée. Par exemple si le système monétaire est  $\{1, 3, 4, 5\}$  et la somme 7, alors on peut utiliser au minimum 2 pièces ( $4 + 3$ ).

🔄 **Rappel** : l'algorithme glouton qui consiste à rendre à tout moment la pièce de plus forte valeur possible ne fournit pas toujours la solution optimale. Ici, on obtiendrait 5, 1, 1 et donc 3 pièces.

# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Position du problème

On dispose d'un *système monétaire* c'est à dire d'un ensemble de valeurs possibles pour les pièces et les billets. Le problème du rendu de monnaie consiste à déterminer le nombre minimal de pièces à utiliser pour former une somme donnée. Par exemple si le système monétaire est  $\{1, 3, 4, 5\}$  et la somme 7, alors on peut utiliser au minimum 2 pièces ( $4 + 3$ ).

🕒 **Rappel** : l'algorithme glouton qui consiste à rendre à tout moment la pièce de plus forte valeur possible ne fournit pas toujours la solution optimale. Ici, on obtiendrait 5, 1, 1 et donc 3 pièces.

- 1 Ecrire une relation de récurrence entre les différentes instances du problème en donnant les solutions des cas de base.

# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Position du problème

On dispose d'un *système monétaire* c'est à dire d'un ensemble de valeurs possibles pour les pièces et les billets. Le problème du rendu de monnaie consiste à déterminer le nombre minimal de pièces à utiliser pour former une somme donnée. Par exemple si le système monétaire est  $\{1, 3, 4, 5\}$  et la somme 7, alors on peut utiliser au minimum 2 pièces ( $4 + 3$ ).

🕒 **Rappel** : l'algorithme glouton qui consiste à rendre à tout moment la pièce de plus forte valeur possible ne fournit pas toujours la solution optimale. Ici, on obtiendrait 5, 1, 1 et donc 3 pièces.

- 1 Ecrire une relation de récurrence entre les différentes instances du problème en donnant les solutions des cas de base.
- 2 Ecrire un programme en C permettant de répondre au problème.

### Position du problème

On dispose d'un *système monétaire* c'est à dire d'un ensemble de valeurs possibles pour les pièces et les billets. Le problème du rendu de monnaie consiste à déterminer le nombre minimal de pièces à utiliser pour former une somme donnée. Par exemple si le système monétaire est  $\{1, 3, 4, 5\}$  et la somme 7, alors on peut utiliser au minimum 2 pièces ( $4 + 3$ ).

🔄 **Rappel** : l'algorithme glouton qui consiste à rendre à tout moment la pièce de plus forte valeur possible ne fournit pas toujours la solution optimale. Ici, on obtiendrait 5, 1, 1 et donc 3 pièces.

- 1 Ecrire une relation de récurrence entre les différentes instances du problème en donnant les solutions des cas de base.
- 2 Ecrire un programme en C permettant de répondre au problème.
- 3 Construire la liste effective des pièces à rendre.

# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Résolution

# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Résolution

1 On note :

# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Résolution

- 1 On note :
  - $s$  la somme à rendre,



# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Résolution

1 On note :

- $s$  la somme à rendre,
- $(p_i)_{1 \leq i \leq n}$  les valeurs des pièces rangées dans l'ordre *croissant*

### Résolution

1 On note :

- $s$  la somme à rendre,
- $(p_i)_{1 \leq i \leq n}$  les valeurs des pièces rangées dans l'ordre *croissant*
- $m(s, k)$  le nombre minimal de pièce pour rendre la somme  $s$  en utilisant les pièces  $(p_i)_{1 \leq i \leq k}$

## Résolution

① On note :

- $s$  la somme à rendre,
- $(p_i)_{1 \leq i \leq n}$  les valeurs des pièces rangées dans l'ordre *croissant*
- $m(s, k)$  le nombre minimal de pièce pour rendre la somme  $s$  en utilisant les pièces  $(p_i)_{1 \leq i \leq k}$

Avec ces notations, on doit donc trouver  $m(s, n)$  et on dispose des relations suivantes :

# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Résolution

① On note :

- $s$  la somme à rendre,
- $(p_i)_{1 \leq i \leq n}$  les valeurs des pièces rangées dans l'ordre *croissant*
- $m(s, k)$  le nombre minimal de pièce pour rendre la somme  $s$  en utilisant les pièces  $(p_i)_{1 \leq i \leq k}$

Avec ces notations, on doit donc trouver  $m(s, n)$  et on dispose des relations suivantes :

$$\begin{cases} m(0, k) &= 0 \text{ pour tout } 1 \leq k \leq n, \\ m(s, 0) &= +\infty \text{ pour tout } s \in \mathbb{N}^*, \\ m(s, k) &= m(s, k-1) \text{ si } s < p_k, \\ m(s, k) &= \min \{1 + m(s - p_k, k), m(s, k-1)\} \text{ sinon.} \end{cases}$$

# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Résolution

- 2 Programme en C

## Résolution

## ② Programme en C

```
1  int min_pieces(int s, int pieces[], int k)
2  {
3      int m1, m2;
4      if (s == 0) {return 0;}
5      if (k == 0) {return INF;}
6      if (s < pieces[k]) {return min_pieces(s, pieces, k-1);}
7      m1 = 1 + min_pieces(s - pieces[k], pieces, k);
8      m2 = min_pieces(s, pieces, k - 1);
9      if (m1 < m2) {return m1;} else {return m2;}
10 }
```

## Résolution

## ② Programme en C

```
1  int min_pieces(int s, int pieces[], int k)
2  {
3      int m1, m2;
4      if (s == 0) {return 0;}
5      if (k == 0) {return INF;}
6      if (s < pieces[k]) {return min_pieces(s, pieces, k-1);}
7      m1 = 1 + min_pieces(s - pieces[k], pieces, k);
8      m2 = min_pieces(s, pieces, k - 1);
9      if (m1 < m2) {return m1;} else {return m2;}
10 }
```

Comme pour la PLSSC, on peut écrire une solution itérative ou une solution utilisant la mémoïsation.

# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Reconstruction de la solution

- ③ On construit une solution effective à partir des valeurs de la matrice  $m(s, k)$

	0	1 ( $p_1=1$ )	2 ( $p_2=3$ )	3 ( $p_3=4$ )	4 ( $p_4=5$ )
0	0	0	0	0	0
1	$\infty$	1	1	1	1
2	$\infty$	2	2	2	2
3	$\infty$	3	1	1	1
4	$\infty$	4	2	1	1
5	$\infty$	5	3	2	1
6	$\infty$	6	2	2	2
7	$\infty$	7	3	2	2



# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Reconstruction de la solution

- ③ On construit une solution effective à partir des valeurs de la matrice  $m(s, k)$

	0	1 ( $p_1=1$ )	2 ( $p_2=3$ )	3 ( $p_3=4$ )	4 ( $p_4=5$ )
0	0	0	0	0	0
1	$\infty$	1	1	1	1
2	$\infty$	2	2	2	2
3	$\infty$	3	1	1	1
4	$\infty$	4	2	1	1
5	$\infty$	5	3	2	1
6	$\infty$	6	2	2	2
7	$\infty$	7	3	2	2

# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Reconstruction de la solution

- 3 On construit une solution effective à partir des valeurs de la matrice  $m(s, k)$

	0	1 ( $p_1=1$ )	2 ( $p_2=3$ )	3 ( $p_3=4$ )	4 ( $p_4=5$ )
0	0	0	0	0	0
1	$\infty$	1	1	1	1
2	$\infty$	2	2	2	2
3	$\infty$	3	1	1	1
4	$\infty$	4	2	1	1
5	$\infty$	5	3	2	1
6	$\infty$	6	2	2	2
7	$\infty$	7	3	2	2

A red dashed line on the right side of the table, labeled  $p_4$ , indicates the column index for the reconstruction step. The value 2 in the cell (2, 4) is circled in blue, and the value 2 in the cell (7, 4) is circled in red.

# C15 Décomposition en sous problèmes

## 8. Exemple résolu : rendu de monnaie

### Reconstruction de la solution

- ③ On construit une solution effective à partir des valeurs de la matrice  $m(s, k)$

	0	1 ( $p_1=1$ )	2 ( $p_2=3$ )	3 ( $p_3=4$ )	4 ( $p_4=5$ )
0	0	0	0	0	0
1	$\infty$	1	1	1	1
2	$\infty$	2	2	2	2
3	$\infty$	3	1	1	1
4	$\infty$	4	2	1	1
5	$\infty$	5	3	2	1
6	$\infty$	6	2	2	2
7	$\infty$	7	3	2	2

Diagram illustrating the reconstruction of the solution from the matrix  $m(s, k)$ . A dashed red line indicates the path of the solution for  $s=7$  and  $k=4$ . The value 2 in the cell  $(s=2, k=4)$  is circled in blue, and the value 2 in the cell  $(s=7, k=4)$  is circled in red. A box labeled  $p_4$  is positioned to the right of the dashed line.