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 arugments.

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 arugments.

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.

```
let rec separe l =
  match l with
  | [] -> [],[]
  | [x] -> [x],[]
  | 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 arugments.

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.

```
let rec separe l =
  match l with
  | [] -> [],[]
  | [x] -> [x],[]
  | 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 arugments.

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

Tri fusion en OCaml

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

Tri fusion en OCaml

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

```
let rec tri_fusion l =
  match l with
  | [] -> []
  | [x] -> [x]
  |  _ -> let l1,l2 = separe l in
          let tl1 = tri_fusion l1 and tl2 = tri_fusion l2 in
          (fusion tl1 tl2);;
```

Equation de complexité

On note :

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

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.

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.

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 :

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 :
$C(n) = k\,C(\frac{n}{p}) + T(n)$

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 :

$C(n) = k\,C(\frac{n}{p}) + T(n)$

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 :
$C(n) = k\,C(\frac{n}{p}) + T(n)$
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,

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 :
$C(n) = k\,C(\frac{n}{p}) + T(n)$
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.

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 :
$C(n) = k\,C(\frac{n}{p}) + T(n)$
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 $T(n)$.

Résolution

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

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 $C \in O(n^{\frac{\log(k)}{\log(p)}})$

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 $C \in O(n^d \log(n))$

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 $C \in O(n^d)$

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.

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 !

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
  $u_{k+1} \leq u_k + M$ 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
  $u_{k+1} \leq u_k + M$ 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
  $u_{k+1} \leq u_k + M$ où $u_k = \frac{C(2^k)}{2^k}$.
  par récurrence immédiate, $u_k \leq u_0 + kM$ et donc $C(n) \leq nu_0 + M\, n \log_2(n)$

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
  $u_{k+1} \leq u_k + M$ où $u_k = \frac{C(2^k)}{2^k}$.
  par récurrence immédiate, $u_k \leq u_0 + kM$ et donc $C(n) \leq nu_0 + M\,n\log_2(n)$
  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.

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 : .

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 $[4, -1, 10, -4, 9]$.

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.

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

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

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

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.

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 :

```c
int tmaxi(int tab[], int size){
    int tmax = tab[0];
    int tij;
    for (int i=0;i<size;i++){
        tij=0;
        for (int j=i;j<size;j++){
            tij = tij + tab[j];
            if (tij>tmax) {tmax = tij;}}
    }
    return tmax;}
```

Résolution : diviser pour régner

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

Résolution : diviser pour régner

1. On note $k = \left\lfloor \frac{n}{2} \right\rfloor$.
   - 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 $k = \left\lfloor \frac{n}{2} \right\rfloor$.
   - 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 tranche 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 $k = \left\lfloor \frac{n}{2} \right\rfloor$.
   - 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 tranche 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.

Résolution : diviser pour régner

Résolution : diviser pour régner

2. Implémentation :

```c
int somme_maxi_aux(int tab[], int size, int start, int end) {
    if (end==start)
        {return tab[start];}
    if (end==start+1)
        {return max3(tab[start],tab[end],tab[start]+tab[end]);}
    int s1, s2, s3;
    int mid = (start+end)/2;
    s1 = somme_maxi_aux(tab, size, start, mid-1);
    s2 = max_tranchek(tab,start,mid,end);
    s3 = somme_maxi_aux(tab, size, mid+1, end);
    return max3(s1,s2,s3);}
```

Résolution : diviser pour régner

Résolution : diviser pour régner

3. Calcul de la complexité

Résolution : diviser pour régner

3. 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 :
$$\left\{ \begin{array}{lcl} C(0) & \in & O(1) \\ C(2n) & = & 2C(n) + O(n) \end{array} \right.$$

Résolution : diviser pour régner

3. Calcul de la complexité
Pour résoudre un problème de taille n, on doit en résoudre deux de tailles $\left\lfloor \frac{n}{2} \right\rfloor$ 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 : $O(n \log n)$.

Multiplication matricielle

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

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}.

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

Algorithme de Strassen

L'algorithme de Strassen est une approche 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 :
  $A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$ 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 :
  $A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$ 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 :
  $A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$ 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 :
  $C_{1,1} = M_1 + M_4 - M_5 + M_7$
  $C_{1,2} = M_3 + M_5$
  $C_{2,1} = M_2 + M_4$
  $C_{2,2} = M_1 - M_2 + M_3 + M_6$

Complexité

L'équation de complexité s'écrit alors :
$C(2n) = 7\,C(n) + O(n^2)$

Complexité

L'équation de complexité s'écrit alors :
$C(2n) = 7\,C(n) + O(n^2)$
Et on montre que $C(n) \in O(n^{\log_2 7})$, et $\log_2 7 \simeq 2,807$

Complexité

L'équation de complexité s'écrit alors :
$C(2n) = 7\,C(n) + O(n^2)$
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 :
$C(2n) = 7\,C(n) + O(n^2)$
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 naif 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 :
$$\begin{cases} f_0 & = & 1, \\ f_1 & = & 1, \\ f_n & = & f_{n-1} + f_{n-2} \text{ pour tout } n \geq 2. \end{cases}$$

```
let rec fibo n =
  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 :

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

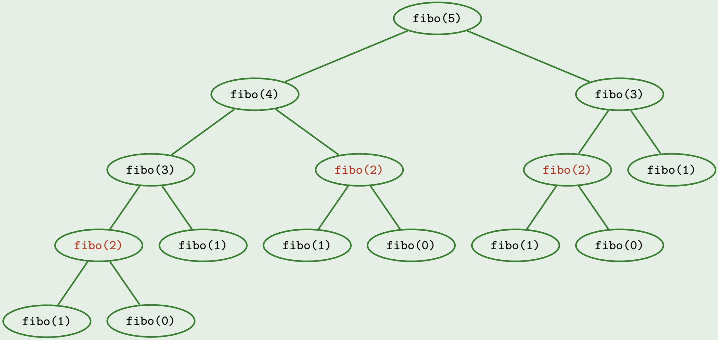
```
let rec fibo n =
  if n < 2 then 1 else fibo (n-1) + fibo (n-2);;
```

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

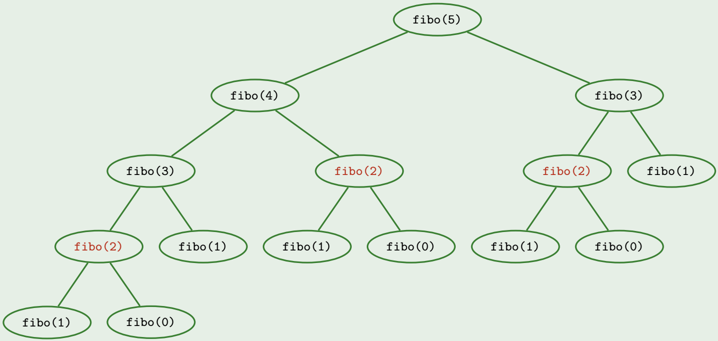
Exemple

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 :
$$\begin{cases} f_0 & = & 1, \\ f_1 & = & 1, \\ f_n & = & f_{n-1} + f_{n-2} \ \text{ pour tout } \ n \geq 2. \end{cases}$$
```
let rec fibo n =
  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

Exemple

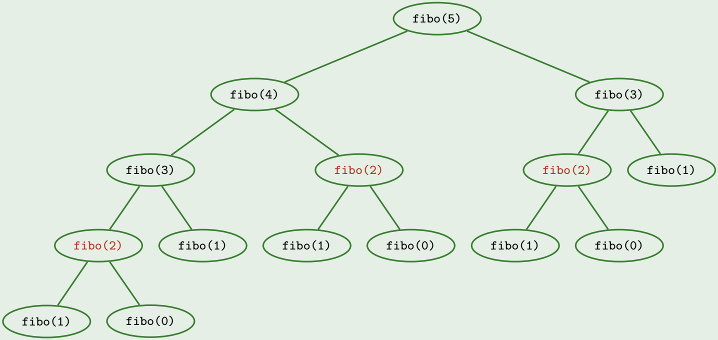

fibo(5)
fibo(4)
fibo(3)
fibo(3)
fibo(2)
fibo(2)
fibo(1)
fibo(2)
fibo(1)
fibo(1)
fibo(0)
fibo(1)
fibo(0)
fibo(1)
fibo(0)

Exemple



Les mêmes appels récursifs apparaissent dans plusieurs branche, on dit qu'il y a *chevauchement des appels récursifs*.

Exemple



Les mêmes appels récursifs apparaissent dans plusieurs branche, on dit qu'il y a *chevauchement des appels récursifs*. (On peut montrer que le nombre d'appels récursifs a_n pour calculer f_n est $a_n = 2\,f_n - 1$ et donc la complexité est exponentielle)

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.

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

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 $18 + 12 = 30$

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.

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, $(v_i)_{0 \leq i \leq N}$, la valeur maximale de la découpe d'une barre de taille i et $(p_i)_{0 \leq i \leq N}$ le prix d'un morceaux de longueur i.

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, $(v_i)_{0 \leq i \leq N}$, la valeur maximale de la découpe d'une barre de taille i et $(p_i)_{0 \leq i \leq N}$ le prix d'un morceaux de longueur i.

1. Donner les valeurs de v_0 et v_1.

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, $(v_i)_{0\leq i\leq N}$, la valeur maximale de la découpe d'une barre de taille i et $(p_i)_{0\leq i\leq N}$ le prix d'un morceaux de longueur i.

1. Donner les valeurs de v_0 et v_1.
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, $(v_i)_{0 \leq i \leq N}$, la valeur maximale de la découpe d'une barre de taille i et $(p_i)_{0 \leq i \leq N}$ le prix d'un morceaux de longueur i.

1. Donner les valeurs de v_0 et v_1.
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.

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, $(v_i)_{0 \leq i \leq N}$, la valeur maximale de la découpe d'une barre de taille i et $(p_i)_{0 \leq i \leq N}$ le prix d'un morceaux de longueur i.

1. Donner les valeurs de v_0 et v_1.
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.

Résolution

1. $v_0 = 0$ et $v_1 = 2$

Résolution

1. $v_0 = 0$ et $v_1 = 2$
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 $k \leq n-1$ et du prix d'un morceau de taille $n-k$, c'est à dire :
$v_n = \max \{v_k + p_{n-k}, 0 \leq k \leq n-1\}$

Résolution

1. $v_0 = 0$ et $v_1 = 2$
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 $k \leq n-1$ et du prix d'un morceau de taille $n-k$, c'est à dire :
   $v_n = \max\left\{v_k + p_{n-k}, 0 \leq k \leq n-1\right\}$
3. Implémentation en C :

```c
int vmax(int barre[], int n) {
    int cmax = 0;
    int vk;
    if (n == 0) {return 0;}
    for (int k = 0; k < n; k++){
        vk = vmax(barre, k);
        if (barre[n-k] + vk > cmax) {cmax = vk + barre[n-k];}
        }
    return cmax;}
```

Résolution

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

   On peut proposer la version avec memoisation suivante :

```c
// Avec mémoisation (v[n]=-1 indique une valeur non encore calculée)
int vmax_memo(int barre[], int n, int v[]){
    if (v[n]!=-1) {return v[n];}
    int cmax = 0;
    int vk;
    for (int k = 0; k < n; k++){
        vk = vmax_memo(barre, k, v);
        if (barre[n-k] + vk > cmax) {cmax = vk + barre[n-k];}
        }
    v[n] = cmax;
    return cmax;}
```

Calcul de bas en haut (*bottom up*)

La mémoïsation 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 mémoïsation 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
$v_n = \max\{v_k + p_{n-k}, 0 \leq k \leq n-1\}$

Calcul de bas en haut (*bottom up*)

La mémoïsation 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
$v_n = \max \{v_k + p_{n-k}, 0 \leq k \leq n-1\}$
Ce qui se traduit en Python par une solution itérative :

Calcul de bas en haut (*bottom up*)

La mémoïsation 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
$v_n = \max\{v_k + p_{n-k}, 0 \leq k \leq n-1\}$
Ce qui se traduit en Python par une solution itérative :

```python
def valeur_max(taille, prix):
    v = {0:0,1:prix[1]}
    for i in range(2,taille+1):
        v[i] = max(v[k]+prix[i-k] for k in range(i))
    return v[taille]
```

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 $(v_k)_{0 \leq k \leq \mathbb{N}}$ 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 $(v_k)_{0 \leq k \leq \mathbb{N}}$ 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 découpes différentes peuvent avoir cette même valeur maximale. Pour rechercher *une* découpe de valeur maximale, on peut par exemple :

- construire le tableau $(v_k)_{0 \leq k \leq \mathbb{N}}$ 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 $(v_k)_{0 \leq k \leq \mathbb{N}}$ 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.

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.
   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.
   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. Les différentes méthodes d'implémentation relèvent du choix du programmeur.

Position du problème

On considère deux chaines 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 chaine w telle que :

Position du problème

On considère deux chaines 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 chaine w telle que :

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

Position du problème

On considère deux chaines 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 chaine w telle que :

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

Position du problème

On considère deux chaines 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 chaine w telle que :

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

Position du problème

On considère deux chaines 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 chaine w telle que :

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

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

Position du problème

On considère deux chaines 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 chaine w telle que :

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

Position du problème

On considère deux chaines 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 chaine w telle que :

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

Position du problème

On considère deux chaines 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 chaine w telle que :

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

Donc ici, la longueur de la plssc est 3.

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 chaine 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 $\mathrm{lplssc}(u_i, v_j)$ 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 chaine 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 chaine 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 $\mathrm{lplssc}(u_i, v_j)$ la longueur de la plssc de u_i et de v_j.

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

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 chaine 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 chaines vides notés "") :

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 chaine 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 $\mathrm{lplssc}(u_i, v_j)$ la longueur de la plssc de u_i et de v_j.

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

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

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 chaine 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 $\mathrm{lplssc}(u_i, v_j)$ la longueur de la plssc de u_i et de v_j.

- Si $u[i] = v[j]$ alors, quelle est la relation entre $\mathrm{lplssc}(u_i, v_j)$ et $\mathrm{lplssc}(u_{i-1}, v_{j-1})$?
  $\mathrm{lplssc}(u_i, v_j) = 1 + \mathrm{lplssc}(u_{i-1}, v_{j-1})$
- Sinon, exprimer $\mathrm{plssc}(u_i, v_j)$ en fonction de $\mathrm{plssc}(u_i, v_{j-1})$ et $\mathrm{plssc}(u_{i-1}, v_j)$
  $\mathrm{lplssc}(u_i, v_j) = \max\left(\mathrm{lplssc}(u_i, v_{j-1}), \mathrm{lplssc}(u_{i-1}, v_j)\right)$
- Déterminer les cas de base (ceux où u et v sont des chaines vides notés "") :
  $\mathrm{lplssc}(u_i, "") = 0$
  $\mathrm{lplssc}("", v_j) = 0$

Programme Python pour plssc

```python
from functools import lru_cache

@lru_cache
def lplssc(mot1,mot2):
    if len(mot2)==0 or len(mot1)==0:
        return 0
    if mot1[-1]==mot2[-1]:
        return 1 + lplssc(mot1[:-1],mot2[:-1])
    return max(lplssc(mot1[:-1],mot2),lplssc(mot1,mot2[:-1]))
```

Construction effective d'une solution

On peut modifier notre programme donnant la longueur de la plssc afin d'obtenir la plssc :

- Si le dernier caractère est commun, il fait partie de la plssc et on relance la récursion sur les parties restantes de chaque chaîne

Construction effective d'une solution

On peut modifier notre programme donnant la longueur de la plssc afin d'obtenir la plssc :

- Si le dernier caractère est commun, il fait partie de la plssc et on relance la récursion sur les parties restantes de chaque chaîne
- Sinon on regarde quel appel produit la plssc et on renvoie le résultat de cet appel

Construction effective d'une solution

On peut modifier notre programme donnant la longueur de la plssc afin d'obtenir la plssc :

- Si le dernier caractère est commun, il fait partie de la plssc et on relance la récursion sur les parties restantes de chaque chaîne
- Sinon on regarde quel appel produit la plssc et on renvoie le résultat de cet appel

```python
@lru_cache
def plssc(mot1,mot2):
    if len(mot2)==0 or len(mot1)==0:
        return ""
    if mot1[-1]==mot2[-1]:
        return plssc(mot1[:-1],mot2[:-1])+mot1[-1]
    if len(plssc(mot1[:-1],mot2))>len(plssc(mot1,mot2[:-1])):
        return  plssc(mot1[:-1],mot2)
    else:
        return plssc(mot1,mot2[:-1])
```

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.

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 $\{5, 4, 3, 1\}$ et la somme 7,

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 $\{5, 4, 3, 1\}$ et la somme 7, alors on peut utiliser au minimum 2 pièces $(4 + 3)$.

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 $\{5, 4, 3, 1\}$ et la somme 7, alors on peut utiliser au minimum 2 pièces $(4 + 3)$.

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 $\{5, 4, 3, 1\}$ 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.

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 $\{5, 4, 3, 1\}$ 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.

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 $\{5, 4, 3, 1\}$ 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 python 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 $\{5, 4, 3, 1\}$ 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 python permettant de répondre au problème.
3. Construire la liste effective des pièces à rendre.

Résolution

Résolution

1. On note :

Résolution

1. On note :
   - S la somme à rendre,

Résolution

1. On note :
    - S la somme à rendre,
    - $(p_i)_{0\leq i\leq n}$ les valeurs des pièces rangées dans l'ordre décroissant

Résolution

1. On note :
   - S la somme à rendre,
   - $(p_i)_{0\leq i\leq n}$ les valeurs des pièces rangées dans l'ordre décroissant
   - $m(S,k)$ le nombre minimal de pièce pour rendre la somme S en utilisant les pièces à partir de la k-ième.

Résolution

1. On note :
   - S la somme à rendre,
   - $(p_i)_{0 \leq i \leq n}$ les valeurs des pièces rangées dans l'ordre décroissant
   - $m(S,k)$ le nombre minimal de pièce pour rendre la somme S en utilisant les pièces à partir de la k-ième.

   Avec ces notations, on doit donc trouver $m(S,0)$ et on dispose des relations suivantes :

Résolution

1. On note :
   - S la somme à rendre,
   - $(p_i)_{0 \leq i \leq n}$ les valeurs des pièces rangées dans l'ordre décroissant
   - $m(S,k)$ le nombre minimal de pièce pour rendre la somme S en utilisant les pièces à partir de la k-ième.

   Avec ces notations, on doit donc trouver $m(S,0)$ et on dispose des relations suivantes :

$$\begin{cases} m(0,k) & = & 0 \text{ pour tout } 0 \leq k \leq n, \\ m(S,n+1) & = & +\infty \\ m(S,k) & = & m(S,k+1) \text{ si } S < p_k, \\ m(S,k) & = & min\left\{1 + m(S-p_k,k), m(S,k+1)\ \right\} sinon. \end{cases}$$

Résolution

2. Programme Python (avec mémoïsation dans le dictionnaire memo)

```python
def rm(somme,k):
    if (somme,k) in memo:
        return memo[(somme,k)]
    if k>=len(systeme):
        memo[(somme,k)]=inf
        return inf
    if systeme[k]>somme:
        memo[(somme,k)] = rm(somme,k+1)
        return memo[(somme,k)]
    memo[(somme,k)] = min(1+rm(somme-systeme[k],k),rm(somme,k+1))
    return memo[(somme,k)]
```

Résolution

3. Pour construire la liste des pièces à rendre, on peut partir des valeurs de $m(S, k)$:

$S \backslash k$	0	1	2	3	4
0	0	0	0	0	∞
1					∞
2					∞
3					∞
4					∞
5					∞
6					∞
7					∞

$$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$$

Résolution

3. Pour construire la liste des pièces à rendre, on peut partir des valeurs de $m(S,k)$:

S \ k	0	1	2	3	4
0	0	0	0	0	∞
1	1	1	1	1	∞
2					∞
3					∞
4					∞
5					∞
6					∞
7					∞

$$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$$

Résolution

3. Pour construire la liste des pièces à rendre, on peut partir des valeurs de $m(S,k)$:

$S \backslash k$	0	1	2	3	4
0	0	0	0	0	∞
1	1	1	1	1	∞
2	2	2	2	2	∞
3					∞
4					∞
5					∞
6					∞
7					∞

$$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$$

Résolution

3. Pour construire la liste des pièces à rendre, on peut partir des valeurs de $m(S,k)$:

S \ k	0	1	2	3	4
0	0	0	0	0	∞
1	1	1	1	1	∞
2	2	2	2	2	∞
3	1	1	1	3	∞
4					∞
5					∞
6					∞
7					∞

$$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$$

Résolution

3. Pour construire la liste des pièces à rendre, on peut partir des valeurs de $m(S,k)$:

S \ k	0	1	2	3	4
0	0	0	0	0	∞
1	1	1	1	1	∞
2	2	2	2	2	∞
3	1	1	1	3	∞
4	1	1	2	4	∞
5					∞
6					∞
7					∞

$$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$$

Résolution

3. Pour construire la liste des pièces à rendre, on peut partir des valeurs de $m(S,k)$:

S \ k	0	1	2	3	4
0	0	0	0	0	∞
1	1	1	1	1	∞
2	2	2	2	2	∞
3	1	1	1	3	∞
4	1	1	2	4	∞
5	1	2	3	5	∞
6					∞
7					∞

$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$

Résolution

3. Pour construire la liste des pièces à rendre, on peut partir des valeurs de $m(S, k)$:

S \ k	0	1	2	3	4
0	0	0	0	0	∞
1	1	1	1	1	∞
2	2	2	2	2	∞
3	1	1	1	3	∞
4	1	1	2	4	∞
5	1	2	3	5	∞
6	2	2	2	6	∞
7					∞

$$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$$

Résolution

3. Pour construire la liste des pièces à rendre, on peut partir des valeurs de $m(S,k)$:

$S \backslash k$	0	1	2	3	4
0	0	0	0	0	∞
1	1	1	1	1	∞
2	2	2	2	2	∞
3	1	1	1	3	∞
4	1	1	2	4	∞
5	1	2	3	5	∞
6	2	2	2	6	∞
7	2	2	3	7	∞

$$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$$

Résolution

3. Pour construire la liste des pièces à rendre, on peut partir des valeurs de $m(S,k)$:

$S \backslash k$	0	1	2	3	4
0	0	0	0	0	∞
1	1	1	1	1	∞
2	2	2	2	2	∞
3	1	1	1	3	∞
4	1	1	2	4	∞
5	1	2	3	5	∞
6	2	2	2	6	∞
7	2	$\boxed{2}$	3	7	∞

$$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$$

Résolution

3. Pour construire la liste des pièces à rendre, on peut partir des valeurs de $m(S,k)$:

S \ k	0	$\boxed{1}$	2	3	4
0	0	0	0	0	∞
1	1	1	1	1	∞
2	2	2	2	2	∞
3	1	1	1	3	∞
4	1	1	2	4	∞
5	1	2	3	5	∞
6	2	2	2	6	∞
7	2	$\boxed{2}$	3	7	∞

$$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$$

Résolution

3. Pour construire la liste des pièces à rendre, on peut partir des valeurs de $m(S,k)$:

S \ k	0	$\boxed{1}$	2	3	4
0	0	0	0	0	∞
1	1	1	1	1	∞
2	2	2	2	2	∞
3	1	1	1	3	∞
4	1	1	2	4	∞
5	1	2	3	5	∞
6	2	2	2	6	∞
7	2	$\boxed{2}$	3	7	∞

$$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$$

Résolution

3. Pour construire la liste des pièces à rendre, on peut partir des valeurs de $m(S,k)$:

S \ k	0	$\boxed{1}$	2	3	4
0	0	0	0	0	∞
1	1	1	1	1	∞
2	2	2	2	2	∞
3	1	1	$\boxed{1}$	3	∞
4	1	1	2	4	∞
5	1	2	3	5	∞
6	2	2	2	6	∞
7	2	$\boxed{2}$	3	7	∞

$$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$$

Résolution

3. Pour construire la liste des pièces à rendre, on peut partir des valeurs de $m(S,k)$:

$S \backslash k$	0	$\boxed{1}$	$\boxed{2}$	3	4
0	0	0	0	0	∞
1	1	1	1	1	∞
2	2	2	2	2	∞
3	1	1	$\boxed{1}$	3	∞
4	1	1	2	4	∞
5	1	2	3	5	∞
6	2	2	2	6	∞
7	2	$\boxed{2}$	3	7	∞

$$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$$

Résolution

3. Cette méthode correspond au programme suivant :

```python
def pieces(memo,somme):
    pieces = []
    while somme!=0:
        min = inf
        piece = None
        for i in range(len(systeme)+1,-1,-1):
            if (somme,i) in memo and memo[(somme,i)]<min:
                min = memo[(somme,i)]
                piece = systeme[i]
        somme = somme-piece
        pieces.append(piece)
    return pieces
```

Résolution

Remarque : sur cet exemple, les appels récursifs depuis $m(7,0)$ ne construisent pas la totalité de la matrice des valeurs de $m(S,k)$. En effet, le calcul de $m(7,0)$ se fait à partir de celui de $m(7,1)$ et de $m(7-p_0,0)$ et ainsi de suite en remontant dans la matrice :

S \ k	0	1	2	3	4
0	0	0	0	0	∞
1					∞
2					∞
3					∞
4					∞
5					∞
6					∞
7	✔				∞

$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$

Résolution

Remarque : sur cet exemple, les appels récursifs depuis $m(7,0)$ ne construisent pas la totalité de la matrice des valeurs de $m(S,k)$. En effet, le calcul de $m(7,0)$ se fait à partir de celui de $m(7,1)$ et de $m(7-p_0,0)$ et ainsi de suite en remontant dans la matrice :

S \ k	0	1	2	3	4
0	0	0	0	0	∞
1					∞
2	✓				∞
3					∞
4					∞
5					∞
6					∞
7	✓	✓			∞

$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$

Résolution

Remarque : sur cet exemple, les appels récursifs depuis $m(7, 0)$ ne construisent pas la totalité de la matrice des valeurs de $m(S, k)$. En effet, le calcul de $m(7, 0)$ se fait à partir de celui de $m(7, 1)$ et de $m(7 - p_0, 0)$ et ainsi de suite en remontant dans la matrice :

S \ k	0	1	2	3	4
0	0	0	0	0	∞
1					∞
2	✔	✔			∞
3		✔			∞
4					∞
5					∞
6					∞
7	✔	✔	✔		∞

$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$

Résolution

Remarque : sur cet exemple, les appels récursifs depuis $m(7,0)$ ne construisent pas la totalité de la matrice des valeurs de $m(S,k)$. En effet, le calcul de $m(7,0)$ se fait à partir de celui de $m(7,1)$ et de $m(7-p_0,0)$ et ainsi de suite en remontant dans la matrice :

S \ k	0	1	2	3	4
0	0	0	0	0	∞
1					∞
2	✔	✔	✔		∞
3		✔	✔		∞
4			✔		∞
5					∞
6					∞
7	✔	✔	✔	✔	∞

$$\boxed{p_0=5, p_1=4, p_2=3, p_3=1}$$

Résolution

Remarque : sur cet exemple, les appels récursifs depuis $m(7,0)$ ne construisent pas la totalité de la matrice des valeurs de $m(S,k)$. En effet, le calcul de $m(7,0)$ se fait à partir de celui de $m(7,1)$ et de $m(7-p_0,0)$ et ainsi de suite en remontant dans la matrice :

S \ k	0	1	2	3	4
0	0	0	0	0	∞
1					∞
2	✓	✓	✓	✓	∞
3		✓	✓	✓	∞
4			✓	✓	∞
5				✓	∞
6					∞
7	✓	✓	✓	✓	∞

$$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$$

Résolution

Remarque : sur cet exemple, les appels récursifs depuis $m(7,0)$ ne construisent pas la totalité de la matrice des valeurs de $m(S,k)$. En effet, le calcul de $m(7,0)$ se fait à partir de celui de $m(7,1)$ et de $m(7-p_0,0)$ et ainsi de suite en remontant dans la matrice :

S \ k	0	1	2	3	4
0	0	0	0	0	∞
1					∞
2	✔	✔	✔	✔	∞
3		✔	✔	✔	∞
4			✔	✔	∞
5				✔	∞
6				✔	∞
7	✔	✔	✔	✔	∞

$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$

Résolution

Remarque : sur cet exemple, les appels récursifs depuis $m(7,0)$ ne construisent pas la totalité de la matrice des valeurs de $m(S,k)$. En effet, le calcul de $m(7,0)$ se fait à partir de celui de $m(7,1)$ et de $m(7-p_0,0)$ et ainsi de suite en remontant dans la matrice :

S \ k	0	1	2	3	4
0	0	0	0	0	∞
1				✓	∞
2	✓	✓	✓	✓	∞
3		✓	✓	✓	∞
4			✓	✓	∞
5				✓	∞
6				✓	∞
7	✓	✓	✓	✓	∞

$$\boxed{p_0 = 5, p_1 = 4, p_2 = 3, p_3 = 1}$$