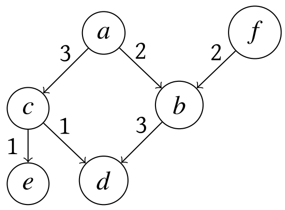
Programmation dynamique

I Exemple fondateur : plus large chemin dans un graphe orienté acyclique

On considère ici un graphe orienté acyclique et pondéré $G = (S, A, \pi)$. On a donc une fonction $\pi : A \to \mathbb{R}$ de pondération des arêtes.

Voici un exemple d'un tel graphe :



On cherche à résoudre le problème suivant : étant donnés $s, t \in S$ tels que $s \rightsquigarrow t$, déterminer $f(s,t) = \max \{ \pi(\varphi) \mid \varphi : s \rightsquigarrow t \}$, voire déterminer le chemin réalisant ce maximum.

Pour un graphe quelconque, ce problème est difficile dans le sens où on ne connait d'autres solutions que d'énumérer tous les chemins, et il y en a un nombre exponentiel.

> **Remarque** C'est un problème dit NP-complet dans le cas de poids positifs. On peut le montrer en remarquant qu'un plus long chemin dans un graphe non orienté complet va forcément passer par toutes les sommets (sinon on pourrait faire grossir le chemin) et permettra donc de résoudre le problème du voyageur de commerce.

On peut établir une récurrence pour calculer $f(s,t)$. En effet, comme le graphe est acyclique, seul le chemin vide relie $s \rightsquigarrow s$ et donc $f(s,s) = 0$. Sinon, on peut essayer les chemins possibles :

$$f(s,t) = \begin{cases} 0 & \text{si } s = t \\ \max \{ \ \pi(s \to s') + f(s',t) \mid s \to s' \in A, s' \rightsquigarrow t \ \} & \text{sinon.} \end{cases}$$

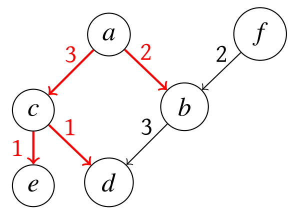
Comme on sait que $s \rightsquigarrow t$, le maximum du second cas est toujours bien défini car l'ensemble est non vide. En considérant que $\max(\emptyset) = -\infty$,cela revient à étendre f par $f(s,t) = -\infty$ quand $s \not\rightsquigarrow t$, on peut alors définir une récurrence plus simple :

$$f(s,t) = \begin{cases} 0 & \text{si } s = t \\ \max \{ \ \pi(s \to s') + f(s',t) \mid s \to s' \in A \ \} & \text{sinon.} \end{cases}$$

Pour calculer f, on peut alors directement implémenter cette récurrence. En faisant cela, on va sûrement devoir calculer de nombreuses fois les mêmes valeurs, comme c'était le cas avec la suite de Fibonacci.

On pourrait ainsi résoudre ce problème de la même manière qu'on l'a fait pour de telles fonctions récursives : en utilisant un cache mémoire permettant d'éviter le recalcul coûteux de certaines valeurs.

On peut aussi tirer partie d'un parcours en profondeur du graphe. En effet, si on effectue un parcours en profondeur récursif depuis le sommet a dans l'exemple au-dessus et qu'on affiche le nom du sommet **au moment où on sort de la fonction récursive de parcours** on pourrait avoir l'affichage suivant : $decba$. On a marqué en rouge le parcours sur le graphe :



Le premier sommet affiché est nécessairement un puits, un sommet sans voisins sortants. Plus globalement, si $x \rightsquigarrow y$ on a forcément y affiché **avant** y. On parle de tri topologique.

> **Remarque** Cette notion est développée dans le chapitre sur les algorithmes de graphe.

Ici, ce qui nous intéresse est directement lié au calcul des valeurs de $f(s,t)$. En effet, si on calcule $f(x,t)$ dans l'ordre donné par ce tri, on sera certain que toutes les valeurs dont on a besoin dans les max ont été calculées.

Dans notre exemple, si on cherche un chemin le plus long entre a et d, on va donc remplir le tableau suivant dans cet ordre :

Ici $f(d,d) = 0$.

d	e	c	b	a
0				

Comme il n'y a pas d'arêtes issus de e :

d	e	c	b	a
0	$-\infty$			

Ensuite, pour c, on effectue le maximum $\max(1-\infty, 1+0) = 1$:

d	e	c	b	a
0	$-\infty$	1		

Pour b on a un seul chemin $f(b,d) = 3$:

d	e	c	b	a
0	$-\infty$	1	3	

On a alors $f(a,d) = \max(3 + f(c,d), 2 + f(b,d)) = \max(4,5) = 5$. Ces deux valeur sont bien été calculées.

d	e	c	b	a
0	$-\infty$	1	3	5

On conclut ainsi que $f(a,d) = 5$.

Notons qu'on aurait pu essayer d'effectuer un parcours glouton depuis a en choisissant l'arête de plus grand poids en descendant vers d. Dans ce cas, on aurait choisi le chemin $a \to c \to d$ de poids 4 qui n'est le plus grand.

Pour résumer ce qu'on vient de dire :

- Il n'est pas possible de trouver le chemin le plus large avec un algorithme glouton.
- On peut écrire une récurrence exacte pour ce problème.
- Cette récurrence peut être calculée naïvement de manière exponentielle.
- On peut établir un mécanisme de cache pour ne calculer qu'une fois chaque valeur nécessaire.
- En raisonnant sur le graphe, on peut aussi déterminer un ordre permettant de remplir directement les valeurs nécessaires dans un tableau en faisant en sorte que toutes les valeurs utiles à un moment aient été précalculées.

Ce programme d'analyse de certains problèmes est très classique et est en général dénommé *programmation dynamique*.

Remarque Le terme dynamique fait référence au fait que l'ordre de traitement n'est pas connu à l'avance. Le terme programmation était pour Bellman une manière d'obtenir des financements de l'armée américaine en reliant cela à ce qu'elle croyait immédiatement pertinent au moment de ses découvertes.

Le problème du chemin le plus large est emblématique de la programmation dynamique : tout problème s'y ramène. En pratique il n'est pas nécessaire de définir le graphe ou d'exprimer un tri topologique explicitement, cependant, on a fait le choix ici de conserver ces termes tout le long du chapitre par cohérence.

II Principe de la programmation dynamique

II.1 Problème et équation de Bellman

On considère ici un cadre assez large où on cherche à calculer une fonction $f : E \to \mathbb{R}$ où est le domaine d'entrée et où on dispose :

- de valeurs connus pour $E_0 \subset E$
- d'une équation pour $x \in E \backslash E_0$ permettant d'exprimer $f(x)$ en fonction de $f(y)$ pour $y \in dep(x)$ où $dep : E \to \mathcal{P}(E)$ est telle que toute chaîne de $(a_1, a_2, ...,)$ où $a_{i+1} \in dep(a_i)$ est finie et où pour toute chaîne de longueur maximale, le dernier élément $a_p \in E_0$.

En fait, cela signifie que dep induit un graphe orienté acyclique sur les sommets E où $v_-(x) = dep(x)$.

Il s'agit donc d'une équation de récurrence dans le sens où on exprime $f(x)$ en fonction d'autres appels à f. On parle de sous-problèmes.

Exemple

- Fibonacci. Ici, c'est un exemple *jouet* uniquement présent pour se concentrer sur l'équation. On a donc $F_0 = 0, F_1 = 1$ et pour $n > 1$, $F_n = F_{n-1} + F_{n-2}$.
- Rendu de monnaie. On cherche à calculer $f(n)$ le plus petit nombre de pièce pour rendre la monnaie dans le système monétaire S. On a $f(0) = 0$ et l'équation

$$f(n) = 1 + \min \{ \ f(n-p) \mid p \in S, p \leq n \ \}$$

II.2 Résolution par récurrence naïve

On peut résoudre l'équation de récurrence naïvement par une fonction récursive.

Comme on a pu le voir, par exemple avec Fibonacci, les récurrences font rapidement apparaitre des arbres d'appels de taille exponentielle.

> **Remarque** On peut se poser la question de la différence avec la méthode diviser pour régner. Dans diviser pour régner, les sous-problèmes ont un arbre récursif d'appel sans partage possible.

Exemple Fibonacci. On a

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

C
```c
int fibo(int n)
{
    if (n <= 1)
        return n;
    return fibo(n-1) + fibo(n-2);
}
```

Python
```python
def fibo(n):
    if n <= 1:
        return n
    return fibo(n-1) + fibo(n-2)
```

Rendu de monnaie. On peut directement écrire :

OCaml
```ocaml
let rec rendu s n =
    if n = 0
    then 0
    else
        let m = ref n in
        for i = 0 to Array.length s - 1 do
            if s.(i) <= n
            then m := min !m (rendu s (n - s.(i)))
        done;
        1 + !m
```

C
```c
int rendu(int *s, int ls, int n)
{
    if (n == 0)
        return 0;
    int m =n;
    for(int i = 0; i < ls; i++)
    {
        if(s[i] <= n)
        {
            int v = rendu(s, ls, n-s[i]);
            if (v < m)
                m = v;
        }
    }
    return 1 + m;
}
```

Python
```python
def rendu(s, n):
    if n == 0:
        return 0
    m = n
    for piece in s:
        if piece <= n:
            m = min(m, rendu(s, n-piece))
    return 1 + m
```

II.3 Mémoïsation et cache dynamique

Une manière d'améliorer la résolution par récurrence est d'utiliser un cache d'appel, c'est-à-dire un dictionnaire dont les clés sont dans E et les valeurs dans $\mathbb{R}$: pour calculer $f(x)$ on regarde si on a déjà une correspondance dans le cache, auquel cas on renvoie cette valeur, sinon, on calcule $f(x)$ avec la récurrence (qui utilisera donc le cache), puis on rajoute une entrée dans le cache pour x.

On dit qu'on a **mémoïsé** la fonction f.

En procédant ainsi, on a linéarisé la récurrence et on effectue uniquement les calculs nécessaires une seule fois.

Pour définir ce cache, on peut utiliser une table de hachage comme celle fournit par le module `Hashtbl` en `OCaml` ou par les dictionnaires de `Python`.

Exemple Fibonacci.

OCaml

```ocaml
let fibo n =
  let cache = Hashtbl.create (n+1) in
  let rec fibo_aux n =
    try
      Hashtbl.find cache n
    with Not_found ->
      let v =
        if n <= 1 then n
        else fibo_aux (n-1) + fibo_aux (n-2)
      in Hashtbl.add cache n v;
      v
  in fibo_aux n
```

Python

```python
cache = {}

def fibo(n) :
    if n in cache :
        return cache[n]
    if n <= 1 :
        v = n
    else :
        v = fibo(n-1) + fibo(n-2)
    cache[n] = v
    return v
```

Rendu de monnaie.

OCaml

```ocaml
let rendu s n =
  let cache = Hashtbl.create (n+1) in
  let rec rendu_aux n =
    try
      Hashtbl.find cache n
    with Not_found ->
      let v =
        if n = 0 then 0
        else let m = ref n in
          for i = 0 to Array.length s - 1 do
            if s.(i) <= n
            then m := min !m (rendu s (n - s.(i)))
          done;
          1 + !m
      in Hashtbl.add cache n v;
      v
  in rendu_aux n
```

Python

```python
cache = {}

def rendu(s, n) :
    if n in cache :
        return cache[n]

    if n == 0 :
        v = 0
    else :
        m = n
        for piece in s :
            if piece <= n :
                m = min(m, rendu(s, n-piece))
        v = 1 + m
    cache[n] = v
    return v
```

On remarque que le code a toujours la même structure on part de

OCaml

```ocaml
let rec f x = expr
```

Python

```python
def f(x) :
    return expr
```

et on passe à :

OCaml
```
let f x =
    (* ici n doit etre proche du nombre d'appels *)
    let cache = Hashtbl.create n in
    let rec f_aux x =
        try
            Hashtbl.find cache x
        with Not_found ->
            (* expr' = expr en changeant f x
               par f_aux x *)
            let v = expr' in
            Hashtbl.add cache x v;
            v
    in f_aux x
```

Python
```
cache = {}

def f(x) :
    if x in cache :
        return cache[x]

    v = expr
    cache[x] = v
    return v
```

Remarque Il n'est pas possible d'effectuer cette transformation d'expression directement depuis `OCaml` puisqu'il serait nécessaire d'aller modifier le code de l'expression.

II.3.i `OCaml` : Cacher le cache dans une clôture

De la manière dont on a codé le cache, on va créer un nouveau cache à chaque calcul de $f(x)$ et jeter l'ancien. Il peut-être intéressant de conserver un cache entre plusieurs appels. Pour cela, on peut utiliser la notion de clôture : une *clôture* est la donnée d'une fonction et des valeurs connues au moment de sa définition.

Ainsi, en écrivant plutôt :

OCaml
```
let f =
    let cache = Hashtbl.create n in
    let rec f_aux x =
        try
            Hashtbl.find cache x
        with Not_found ->
            let v = expr' in
            Hashtbl.add cache x v;
            v
    in f_aux
```

On crée une fonction `f_aux` qui connait le `cache` et peut l'utiliser et ce sera la valeur de `f`. Quand on appelle `f x` puis `f y`, on va en fait appeler `f_aux x` et `f_aux y` qui connaissent et manipulent le même `cache`.

Remarque On peut expliciter le fonction avec un `fun` :

OCaml
```
let f =
    let cache = Hashtbl.create n in
    let rec f_aux x =
        try
            Hashtbl.find cache x
        with Not_found ->
            let v = expr' in
            Hashtbl.add cache x v;
            v
    in
    fun x -> f_aux x
```

Dans le cas de `fibo`, on aurait alors le programme suivant :

OCaml
```
let fibo =
    let cache = Hashtbl.create (n+1) in
    let rec fibo_aux n =
        try
            Hashtbl.find cache n
```

```
    with Not_found ->
      let v = if n <= 1
          then n
          else fibo_aux (n-1) + fibo_aux (n-2)
      in Hashtbl.add cache n v;
      v
  in fibo_aux
```

II.3.ii Python : Cacher le cache dans un paramètre optionnel

En Python, on peut également passer par une sous fonction et une cloture, comme dans le code suivant :

Python
```python
def fibo(n) :
    cache = {}
    def fibo_aux(n) :
        if n in cache :
            return cache[n]
        if n <= 1 :
            v = n
        else :
            v = fibo_aux(n-1) + fibo_aux(n-2)
        cache[n] = v
        return v
    return fibo_aux(n)
```

Cependant, on utilise en général une unique fonction et un paramètre optionnel dans ce cas-là.

Il est possible de déclarer des paramètres comme optionnel à une fonction en rajoutant, après les arguments non optionnels, des arguments de la forme `argument=valeur_defaut`.

Ainsi,

Python
```python
def f(x, y=3) :
    return x+y
```

Permet de calculer `f(1,2)` pour renvoyer `1+2` mais si on utilise uniquement `f(1)` la valeur de y correspondra à la valeur déclarée par défaut, ainsi la fonction renverra `1+3`.

Une idée est alors de passer le cache comme paramètre optionnel avec une valeur par défaut `{}`. Cependant, **il s'agit ici d'une erreur classique** en Python : les valeurs par défaut sont figées au moment de la création de la fonction et tous les appels sans valeur explicite du cache auront le même cache, et pas un nouveau cache créé pour l'occasion, ce qui n'est clairement pas le comportement voulu.

L'approche usuelle est donc de passer une valeur comme `None` au cache par défaut et de le créer dans ce cas-là :

Python
```python
def fibo(n, cache=None) :
    if cache is None :
        cache = {} # créé à la volée
    if n in cache :
        return cache[n]
    if n <= 1 :
        v = n
    else :
        # ici crucial d'appeler avec le cache
        v = fibo(n-1,cache) + fibo(n-2,cache)
    cache[n] = v
    return v
```

II.4 Graphe de dépendances et tabulation

Comme on a un graphe de dépendance entre sous-problèmes qui est acyclique, on peut en déduire une tri topologique $x_0 < x_1 < \cdots < x_{n-1}$ qui garantit que si $f(x_i)$ a besoin dans son calcul de la valeur $f(x_j)$, alors $j < i$.

On peut alors créer un tableau de n cases où on place à la case i la valeur de $f(x_i)$.

Ce tableau peut se calculer par un remplissage par indice croissant. On dit qu'on a **tabulé** le problème.

Remarque Il s'agit ici d'une politique agressive de remplissage du cache de memoïsation où on remplit le cache sans même savoir si on va utiliser une valeur. Ce qu'on semble perdre en calculant *trop*, on le regagne dans la simplicité de la gestion d'un tableau plutôt qu'une table. Comme souvent, la question va être celle du compromis entre le nombre de valeurs utiles pour le calcul de $f(x)$ et le nombre de valeurs avant x dans le tri topologique considéré.

Exemple Fibonacci. Ici, on a directement l'ordre des entiers qui est un tri topologique.

OCaml
```ocaml
let fibo n =
  if n <= 1 then n
  else
    let t = Array.make (n+1) 0 in
    t.(1) <- 1;
    for i = 2 to n do
      t.(i) <- t.(i-1) + t.(i-2)
    done;
    t.(n)
```

C
```c
int fibo(int n)
{
  if (n <= 1)
    return n;
  int *t = malloc(sizeof(int) * (n+1));
  t[0] = 0; t[1] = 1;
  for (int i = 2; i <= n; i++)
    t[i] = t[i-1] + t[i-2];
  return t[n];
}
```

Python
```python
def fibo(n) :
  if n <= 1 :
    return n

  t = [ 0 ] * (n+1)
  t[1] = 1

  for i in range(2,n+1) :
    t[i] = t[i-1] + t[i-2]

  return t[n]
```

Rendu de monnaie.

OCaml
```ocaml
let rendu s n =
  if n = 0 then 0
  else
    let t = Array.make (n+1) 0 in
    for i = 1 to n do
      let m = ref i in
      for j = 0 to Array.length s - 1 do
        let p = s.(j) in
        if p <= i
        then m := min !m t.(i - p)
      done;
      t.(i) <- 1 + !m
    done;
    t.(n)
```

C
```c
int rendu(int *s, int ls, int n)
{
  if (n == 0)
    return 0;

  int *t = malloc(sizeof(int) * (n+1));
  t[0] = 0;
  for(int i = 0; i <= n; i++)
  {
    int m = i;
    for(int j = 0; j < ls; j++)
    {
      if(s[j] <= i)
      {
        int v = t[i-s[j]];
        if (v < m)
          m = v;
      }
    }
    t[i] = 1 + m;
  }

  return t[n];
}
```

Python
```python
def rendu(s, n) :
    if n == 0 :
        return 0

    t = [ 0 ] * (n+1)

    for i in range(1, n+1) :
        m = i
        for piece in s :
            if piece <= i and t[i-piece] < m :
                m = t[i-piece]
        t[i] = 1 + m

    return t[n]
```

II.5 Reconstruction

La plupart des problèmes ne sont pas juste des problèmes de calcul d'une valeur $f(x)$ mais des problèmes où on considère un ensemble F de valeurs à construire avec une fonction $\Phi : E \to F$ et une fonction d'objectif $\varphi : F \to \mathbb{R}$ et où $f(x) = \varphi(\Phi(x))$.

L'exemple principale est celui d'associer à tout $x \in E$ un ensemble de solutions $sol(x) \subset F$ et à chercher la solution minimisant (ou maximisant) l'objectif :

$$\Phi(x) = argmin\{\ \varphi(y) \mid y \in sol(x)\ \}$$

■ **Note 1** Unifier avec la présentation des gloutons

■

On a alors deux stratégies pour obtenir $\Phi(x)$ quand on a trouvé comment calculer $f(x)$. La première consiste à calculer Φ et f ensemble. Par exemple, dans le rendu de monnaie, on sait qu'on calcule le minimum la pièce qui le réalise :

OCaml
```ocaml
let rendu s n =
    if n = 0 then []
    else
        let t = Array.make (n+1) 0 in
        let util = Array.make (n+1) (-1) in
        for i = 1 to n do
            let m = ref i in
            for j = 0 to Array.length s - 1 do
                let p = s.(j) in
                if p <= i && t.(i-p) < !m
                then begin
                    util.(i) <- p;
                    m := t.(i - p)
                end
            done;
            t.(i) <- 1 + !m
        done;
        let rec rendu_expl n =
            if n = 0
            then []
            else let p = util.(n) in
                p :: rendu_expl (n-p)
        in rendu_expl n
```

Python
```python
def rendu(s, n) :
    if n == 0 :
        return []

    t = [ 0 ] * (n+1)
    util = [ None ] * (n+1)

    for i in range(1, n+1) :
        m = i
        for piece in s :
            if piece <= i and t[i-piece] < m :
                m = t[i-piece]
                util[i] = piece
        t[i] = 1 + m

    res = []
    while n > 0 :
        res.append(util[n])
        n = n - util[n]

    return res
```

Cependant, si on connait le tableau `t` complet, il est facile de constater qu'on peut retrouver une pièce ayant réalisé le minimum en cherchant $p \leq x$ tel que $t.(x) = 1 + t.(x-p)$. On en déduit une reconstruction de la solution uniquement avec les valeurs.

Il est en général intéressant de faire cela, même si ce n'est pas forcément efficace.

II.6 Dimensionnalité réelle

Dans de nombreux cas, le problème n'est pas exprimé sur l'entrée E mais sur un autre ensemble plus petit. Par exemple, on peut chercher une valeur particulière à calculer depuis un tableau et penser que l'entrée est donnée par la taille du tableau, par exemple en raisonnant sur les préfixes, et, en fait, il est nécessaire de considérer tous les sous-tableaux pour calculer la solution. Ainsi, E est alors les couples d'indices (i, j) et on retrouve le problème initial en considérant $(0, n-1)$ où n est la longueur du tableau.

Il n'est ainsi par rare d'avoir un problème portant sur un objet de taille n mais faisant apparaitre des d-uplets et donc avec une taille réelle en n^d.

II.7 Complexité

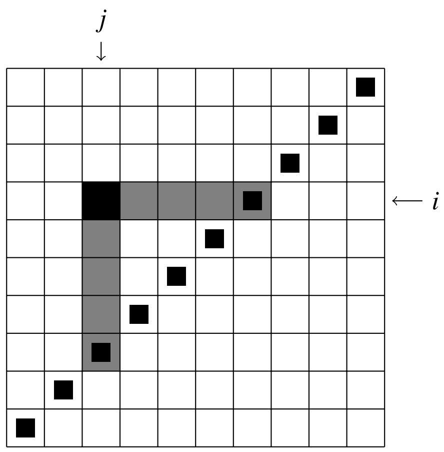
Une fois identifié la vraie entrée E, la complexité est en $O(|avant(x)|)$ où $avant(x) = \{ \ y \in E \mid y \leq x \ \}$ dans le tri topologique considéré.

Comme vu dans la partie précédente, on est le plus souvent en $O(n^d)$ où d est la dimensionnalité réelle du problème.

II.8 Déterminer un tri topologique

Le plus souvent, on va tabuler le problème sur des tableaux à d dimensions. La question du tri topologique revient alors à déterminer d boucles `for` pour remplir dans le bon ordre.

Il peut être intéressant de réaliser un schéma pour visualiser les dépendances et déterminer un ordre de remplissage.



II.9 Résumé

Face à un problème qui semble soluble par programmation dynamique on va donc :

1. Identifier la notion de problème et de sous-problèmes permettant d'établir une équation de Bellman pour un **calcul**.

Là, on a deux choix :

2. Résoudre l'équation de Bellman naïvement avec un cache de mémoïsation.

ou

2. Tabuler les résultats après avoir déduit un ordre de parcours compatible avec les dépendances de l'équation.

Puis,

3. On décore le code du calcul pour permettre de construire les objets optimaux recherchés.

Qu'on peut éventuellement faire directement depuis la table ou le cache de calcul.

III Exemples

Pour chacun de ces exemples, on cherchera à mettre en place une solution par mémoïsation puis par tabulation, avec une reconstruction directe ou indirecte.

III.1 Sous-séquence contigüe maximale

On considère un tableau t de taille n contenant des entiers et on demande de trouver le couple (i, l) tel que la somme

$$t[i] + \cdots + t[i+l]$$

soit la plus grande possible.

Si on considère le tableau :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t[i]	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

La plus grande somme est $t[7] + t[8] + t[9] + t[10] = 43$.

III.1.i Notion de sous-problème et récurrence

Ici, on va introduire un sous-problème différent qui est celui de calculer

$$f(i) = \max_{i \leq k < n} \sum_{j=i}^{k} t[j]$$

c'est-à-dire la plus grande somme démarrant par $t[i]$.

La connaissance d'une seule valeur de f ne sera pas suffisante pour déterminer la plus grande somme quelconque. Cependant, il va suffir de calculer un maximum des valeurs prises par f.

Pour le tableau précédent on obtient :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t[i]	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
f(i)	13	-3	-4	21	1	4	20	43	25	5	12	-5	-4	18	3	7

L'intérêt d'avoir rigidifié la somme en fixant un point de départ est de pouvoir exprimer la relation suivante :

$$f(n-1) = t[n-1] \qquad \forall i < n-1, f(i) = t[i] + \begin{cases} f(i+1) & \text{si } f(i+1) > 0 \\ 0 & \text{sinon} \end{cases}$$

Notons qu'on peut écrire aussi $f(i) = t[i] + max(f(i+1), 0)$.

III.1.ii Récurrence naïve et mémoïsation

On écrit directement le programme résolvant cette récurrence :

OCaml
```ocaml
let rec f t i =
    let n = Array.length t in
    if i = n - 1
    then t.(n-1)
    else t.(i) + max 0 (f t (i+1))
```

Python
```python
def f(t, i) :
    if i == len(t) - 1 :
        return t[i]
    return t[i] + max(0, f(t,i+1))
```

On peut alors répondre au problème initial avec :

OCaml
```ocaml
let maxsubarray t =
    let n = Array.length t in
    let m = ref (f t 0) in
    for i = 1 to n-1 do
        m := max !m (f t i)
    done;
    !m
```

Python
```python
def maxsubarray(t) :
    m = f(t, 0)
    for i in range(1, len(t)) :
        m = max(m, f(t,i))
    return m
```

On adapte directement le code vu précédemment pour ajouter un cache de mémoïsation :

OCaml
```ocaml
let cache = Hashtbl.create 42
    (* taille quelconque ici *)

let rec f t i =
    try
        Hashtbl.find cache i
    with Not_found ->
        let v = let n = Array.length t in
            if i = n - 1
            then t.(n-1)
            else t.(i) + max 0 (f t (i+1))
        in Hashtbl.add cache i v; v
```

Python
```python
cache = {}

def f(t, i) :
    if i in cache :
        return cache[i]

    if i == len(t) - 1 :
        v = t[i]
    else :
        v = t[i] + max(0, f(t,i+1))
    cache[i] = v
    return v
```

En plaçant le cache en dehors de la fonction, on permet à `maxsubarray` de ne pas recalculer les valeurs mais on rend la fonction à usage unique : il faudrait vider le cache au début de `maxsubarray`.

III.1.ii.a `OCaml` : cache dans une clôture paramétrée

Il est beaucoup plus élégant de cacher le cache dans une clôture, mais en faisant cela, on va se heurter au fait qu'il faut un cache partagé entre les différentes valeurs de i mais commun à un t donné. L'astuce réside dans le fait de *couper* la fonction en insérant la création du cache après le paramètre `t` :

OCaml
```ocaml
let f t =
    let n = Array.length t in
    let cache = Hashtbl.create (n+1) in
        (* ici on a une bonne valeur *)
    let rec f_aux i =
        try
            Hashtbl.find cache i
        with Not_found ->
            let v = let n = Array.length t in
                if i = n - 1
                then t.(n-1)
                else t.(i) + max 0 (f_aux (i+1))
            in Hashtbl.add cache i v; v
    in f_aux
```

Remarque On finit sur `f_aux` qui renvoie une fonction `int -> int` mais on aurait pu écrire `fun i -> f_aux i` pour rendre la fonctionnelle plus explicite.

III.1.iii Tabulation

On tabule `f` directement en remplissant un tableau de droite à gauche vu que $f(i)$ dépend de $f(i+1)$:

OCaml
```ocaml
let maxsubarray t =
    let n = Array.length t in
    let f = Array.make n 0 in
    f.(n-1) <- t.(n-1);
    for i = n-2 downto 0 do
        f.(i) <- t.(i) + max 0 f.(i+1)
    done;
    let m = ref f.(0) in
    for i = 1 to n-1 do
        m := max !m f.(i)
    done;
    !m
```

Python
```python
def maxsubarray(t) :
    n = len(t)
    f = [ 0 ] * n
    f[n-1] = t[n-1]
    for i in reversed(range(0,n-1)) :
        f[i] = t[i] + max(0, f[i+1])

    # on peut se servir de max
    # sur un tableau ici
    return max(f)
```

III.1.iv Reconstruction

Lorsqu'on effectue le calcul du maximum dans la fonction précédente, il est direct de garder dans une variable l'indice où il se produit. Cependant, en procédant ainsi, on ne peut pas en déduire facilement la longueur de la somme. Pour ce faire, on va créer un nouveau tableau indiquant à l'indice i la longueur de la plus grande somme commençant par $t[i]$.

En effet, si on note $l(i)$ où $f(i) = \sum_{k=i}^{i+l(i)} t[k]$ on a $l(n-1) = 0$ et

$$\forall i < n-1, l(i) = \begin{cases} 1 + l(i+1) & \text{si } f(i+1) > 0 \\ 0 & \text{sinon} \end{cases}$$

Sur l'exemple précédent on va donc calculer les valeurs suivantes :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t[i]	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
f(i)	13	-3	-4	21	1	4	20	43	25	5	12	-5	-4	18	3	7
l(i)	0	0	8	7	6	5	4	3	2	1	0	0	3	2	1	0

Il n'y a alors plus qu'à lire le couple $(i, l(i))$ correspondant au maximum pour f.
On programme cela directement en décorant le programme précédent :

OCaml

```ocaml
let maxsubarray t =
    let n = Array.length t in
    let f = Array.make n 0 in
    let l = Array.make n 0 in
    f.(n-1) <- t.(n-1);
    for i = n-2 downto 0 do
        if f.(i+1) > 0
        then begin
            f.(i) <- t.(i) + f.(i+1);
            l.(i) <- 1 + l.(i+1)
        end else f.(i) <- t.(i)
    done;
    let m = ref 0 in
    for i = 1 to n-1 do
        if f.(!m) < f.(i)
        then m := i
    done;
    !m, l.(!m)
```

Python

```python
def maxsubarray(t) :
    n = len(t)
    f = [ 0 ] * n
    l = [ 0 ] * n
    f[n-1] = t[n-1]
    for i in reversed(range(0,n-1)) :
        if f[i+1] > 0 :
            f[i] = t[i] + max(0, f[i+1])
            l[i] = 1 + l[i+1]
        else :
            f[i] = t[i]

    m = 0
    for i in range(1, n) :
        if f[m] < f[i] :
            m = i

    return m, l[m]
```

III.2 Chemins monotones

On considère ici le problème de la plus grande somme en partant du sommet d'un triangle de nombres et descendant soit en bas à gauche ou en bas à droite d'un cran.

```
                    75
                  95 64
                17 47 82
              18 35 87 10
            20 04 82 47 65
          19 01 23 75 03 34
        88 02 77 73 07 63 67
      99 65 04 28 06 16 70 92
    41 41 26 56 83 40 80 70 33
  41 48 72 33 47 32 37 16 94 29
53 71 44 65 25 43 91 52 97 51 14
```

```
   70 11 33 28 77 73 17 78 39 68 17 57
  91 71 52 38 17 14 91 43 58 50 27 29 48
 63 66 04 68 89 53 67 30 73 16 69 87 40 31
04 62 98 27 23 09 70 98 73 93 38 53 60 04 23
```

Remarque Il s'agit des problèmes Project Euler 18 et Project Euler 67

Tout d'abord, on va représenter le problème sous la forme d'un tableau de tableau, ligne par ligne, ainsi :

OCaml
```ocaml
let t = [|
    [| 75 |];
    [| 95; 64 |];
    [| 17; 47; 82 |];
    ...
|]
```

Python
```python
t = [
  [ 75 ],
  [ 95, 64 ],
  [ 17, 47, 82 ],
  ...
  ]
```

Ce tableau a n lignes ayant de 1 à n éléments. Les successeurs de l'élément d'indice (i, j) sont les éléments d'indice $(i+1, j)$ et $(i+1, j+1)$. On notera ici $t(i, j)$ la valeur en (i, j) dans le tableau t.

III.2.i Notion de sous-problèmes et récurrence

Ici, on considère $f(i, j)$ la plus grande somme à partir de la ième ligne et jème colonne. On a cherche donc $f(0, 0)$ pour répondre au problème initial.

Sur la dernière ligne, pas le choix, on ne peut considérer que l'élément lui-même, on a donc $f(n-1, j) = t(n-1, j)$.

Pour les autres éléments, il faut considérer les deux possibilités de premier déplacement, en bas à gauche ou à droite, et prendre le choix qui aboutit à la plus grande somme :

$$\forall i [\![0, n-2]\!], \forall j \in [\![0, n-1]\!], f(i, j) = t(i, j) + \max(f(i+1, j), f(i+1, j+1))$$

Et ainsi, on a les dépendances $dep(i, j) = \{(i+1, j), (i+1, j+1)\}$.

III.2.ii Résolution naïve et mémoïsation

On implémente directement cette récurrence :

OCaml
```ocaml
let rec maxpath t i j =
    if i = Array.length t - 1
    then t.(i).(j)
    else t.(i).(j) + max (maxpath t (i+1) j)
                     (maxpath t (i+1) (j+1))
```

Python
```python
def maxpath(t, i, j) :
    if i == len(t) - 1 :
        return t[i][j]
    return t[i][j] + max(maxpath(t, i+1, j),
                         maxpath(t, i+1, j+1))
```

Remarque Pour lire le triangle depuis un fichier, on peut utiliser la fonction suivante :

OCaml
```ocaml
let lit_triangle f =
    let rec aux () =
        try
            let l = input_line f in
            Array.of_list
                (List.map int_of_string
                    (String.split_on_char ' ' l))
            :: aux ()
        with End_of_file -> []
    in Array.of_list (aux ())
```

Python
```python
def lit_triangle(f) :
    t = []
    for l in open(f) :
        t.append([ int(tok) for tok
                   in l.split() ])
    return t
```

On modifie alors cette fonction assez rapidement pour utiliser la mémoïsation. On prend garde notamment au fait que les clés sont maintenant des couples d'indices (i, j).

OCaml

```ocaml
let cache = Hashtbl.create 42

let rec maxpath t i j =
    try
        Hashtbl.find cache (i,j)
    with Not_found ->
        let v = if i = Array.length t - 1
            then t.(i).(j)
            else t.(i).(j) + max (maxpath t (i+1) j)
                                 (maxpath t (i+1) (j+1))
        in Hashtbl.add cache (i,j) v ; v
```

Python

```python
cache = {}

def maxpath(t, i, j) :
    if (i,j) in cache :
        return cache[(i,j)]
    if i == len(t) - 1 :
        v = t[i][j]
    else :
        v= t[i][j] + max(maxpath(t, i+1, j),
                         maxpath(t, i+1, j+1))
    cache[(i,j)] = v
    return v
```

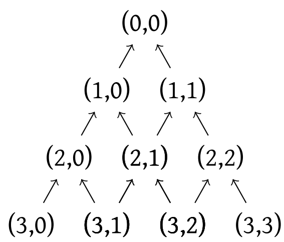
Remarque `OCaml` - Là encore, on peut cacher le dictionnaire dans une clôture pour créer un cache par tableau `t`.

OCaml

```ocaml
let maxpath t =
    let n = Array.length t in
    let cache = Hashtbl.create (n*n) in
    let rec aux i j =
        try
            Hashtbl.find cache (i,j)
        with Not_found ->
            let v = if i = Array.length t - 1
                then t.(i).(j)
                else t.(i).(j) + max (aux (i+1) j)
                                     (aux (i+1) (j+1))
            in Hashtbl.add cache (i,j) v ; v
    in aux
```

III.2.iii Tabulation

On peut représenter le graphe de dépendances des sous problèmes :



On remarque assez naturellement qu'on peut obtenir un tri topologique en traitant ce graphe du bas vers le haut. Ici, il suffit donc de tabuler dans le même format triangulaire que `t` (ou dans une matrice par simplicité) et de remplir `t(i,j)` par `i` décroissant puis `j` croissant (ou décroissant).

On en déduit alors le programme suivant :

OCaml

```ocaml
let maxpath t =
  let n = Array.length t in
  let som = Array.make_matrix n n 0 in
  for j = 0 to n-1 do
    som.(n-1).(j) <- t.(n-1).(j)
  done;
  for i = n-2 downto 0 do
    for j = 0 to i do
      som.(i).(j) <- t.(i).(j)
        + max som.(i+1).(j) som.(i+1).(j+1)
    done
  done;
  som.(0).(0)
```

Python

```python
def maxpath(t) :
    n = len(t)
    # matrice sans partage par compréhension
    som = [ [ 0 ] * n for _ in range(n) ]
    for j in range(n) :
        som[n-1][j] = t[n-1][j]
    for i in reversed(range(n-1)) :
        for j in range(i+1) :
            som[i][j] = t[i][j] \
                + max(som[i+1][j], som[i+1][j+1])
    return som[0][0]
```

III.2.iv Reconstruction

Pour reconstruire un chemin de plus grande somme, il suffit de se souvenir des embranchements pris en calculant le maximum. Pour cela, on construit de même un tableau (ou une matrice pour simplifier) `choix` où `choix(i,j)` indique `Gauche` ou `Droite`.

OCaml

```ocaml
type choix = Gauche | Droite

let maxpath t =
  let n = Array.length t in
  let som = Array.make_matrix n n 0 in
  let ch = Array.make_matrix n n None in
  for j = 0 to n-1 do
    som.(n-1).(j) <- t.(n-1).(j)
  done;
  for i = n-2 downto 0 do
    for j = 0 to i do
      ch.(i).(j) <-
        Some (if som.(i+1).(j) > som.(i+1).(j+1)
              then Gauche else Droite);
      som.(i).(j) <- t.(i).(j)
        + max som.(i+1).(j) som.(i+1).(j+1)
    done
  done;
  let rec chemin i j =
    (i,j) :: match ch.(i).(j)
    with None -> []
    | Some Gauche -> chemin (i+1) j
    | Some Droite -> chemin (i+1) (j+1)
  in chemin 0 0
```

Python

```python
def maxpath(t) :
    n = len(t)
    # matrice sans partage par compréhension
    som = [ [ 0 ] * n for _ in range(n) ]
    ch = [ [ None ] * n for _ in range(n) ]
    for j in range(n) :
        som[n-1][j] = t[n-1][j]
    for i in reversed(range(n-1)) :
        for j in range(i+1) :
            if som[i+1][j] > som[i+1][j+1] :
                ch[i][j] = 'gauche'
            else :
                ch[i][j] = 'droite'
            som[i][j] = t[i][j] \
                + max(som[i+1][j], som[i+1][j+1])

    # on remonte alors dans le tableau pour
    # reconstruire le chemin
    posi, posj = 0, 0
    chemin = []
    while ch[posi][posj] != None :
        chemin.append( (posi, posj) )
        posi = posi+1
        if ch[posi][posj] == 'droite' :
            posj = posj+1
    chemin.append( (posi, posj) )
    return chemin
```

III.2.v Complexité

Il faut explorer chaque case pour pouvoir conclure, donc on sait que la complexité temporelle est au mieux en $O(n^2)$ ce qui est atteint ici. Comme on a utilisé des tableaux de mêmes dimensions pour la tabulation ou le cache, on est également en $O(n^2)$ en espace.

III.3 Produit de matrices

On souhaite calculer un produit $A_0 \dots A_{n-1}$ de matrices. Pour cela, on aimerait découper le calcul en produit de deux matrices, cela revient à placer des parenthèses.

Le problème est qu'une multiplication matriciel effectue un nombre d'opérations de l'ordre de mnp quand on multiplie une matrice (m,n) et une matrice (n,p). Pour simplifier, on considèra que la constante ici vaut 1 et on cherche le plus petit nombre d'opérations à effectuer.

III.3.i Notion de sous-problème et récurrence

Tout d'abord, on remarque que si le produit $A_i A_{i+1}$ est licite, alors A_i a autant de colonnes que A_{i+1} a de lignes. Il suffit donc ne donner que le tableau des lignes et le nombre de colonne de A_n.

On considère donc le tableau `t` de `n+1` valeurs où la matrice A_i a `t[i]` lignes `t[i+1]` colonnes.

Exemple Si on considère $A_0 \dots A_5$ de dimensions respectives $(30, 35)$, $(35, 15)$, $(15, 5)$, $(5, 10)$, $(10, 20)$ et $(20, 25)$, on aura ainsi le tableau :

i	0	1	2	3	4	5	6
t[i]	30	35	15	5	10	20	25

Effectuer un produit et placer des parenthèses va naturellement induire deux sous-problèmes portant sur des matrices rangées à la suite. C'est pour cela qu'on introduit le problème du calcul de $f(i, j)$ plus petit nombre d'opérations pour multiplier les matrices $A_i A_{i+1} \dots A_j$.

Si $i = j$, on n'a pas de mulitplications à faire et on a directement $f(i, i) = 0$. Sinon, on va forcément faire un produit $(A_i \dots A_k)(A_{k+1} \dots A_j)$ pour $i \leq k < j$. Il suffit d'effectuer le moins couteux :

$$\forall i < j, f(i, j) = \min_{i \leq k < j} (t[i]t[k+1]t[j+1] + f(i, k) + f(k+1, j))$$

III.3.ii Récurrence naïve et mémoïsation

OCaml
```
let rec matmult t i j =
    if i = j
    then 0
    else let m = ref (t.(i) * t.(i+1) * t.(j+1)
             + matmult t (i+1) j) in
       for k = i+1 to j-1 do
         m := min !m
           (t.(i) * t.(k+1) * t.(j+1)
             + matmult t i k + matmult t (k+1) j)
       done;
       !m
```

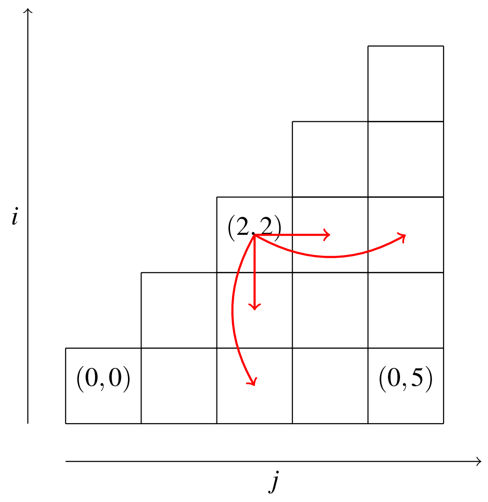
Python
```
def matmult(t, i, j) :
    if i == j :
        return 0
    m = t[i] * t[i+1] * t[j+1]
        \ + matmult(t, i+1, j)
    for k in range(i+1, j) :
        m = min(m,
            t[i] * t[k+1] * t[j+1]
             \ + matmult(t, i, k)
             \ + matmult(t, k+1, j))
    return m
```

OCaml
```
let matmult t =
    let n = Array.length t in
    let cache = Hashtbl.create (n*n) in
    let rec aux i j =
        try
            Hashtbl.find cache (i,j)
        with Not_found ->
            let v =  if i = j
                then 0
                else
                    let m = ref (t.(i) * t.(i+1) * t.(j+1)
                        + matmult t (i+1) j) in
                    for k = i+1 to j-1 do
                        m := min !m
                            (t.(i) * t.(k+1) * t.(j+1)
                                + aux i k
                                + aux (k+1) j)
                    done;
                    !m
            in Hashtbl.add cache (i,j) v; v
    in aux
```

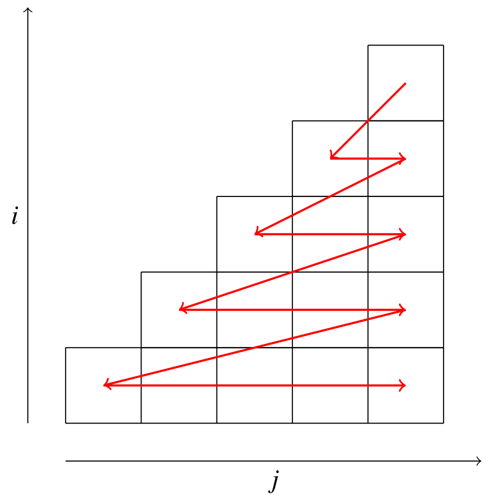
Python
```
cache = {}

def matmult(t, i, j) :
    if (i,j) in cache :
        return cache[(i,j)]

    if i == j :
        v = 0
    else :
        m = t[i] * t[i+1] * t[j+1]
          \ + matmult(t, i+1, j)
        for k in range(i+1, j) :
            m = min(m,
                t[i] * t[k+1] * t[j+1]
                 \ + matmult(t, i, k)
                 \ + matmult(t, k+1, j))
        v = m
    cache[(i,j)] = v
    return v
```

III.3.iii Tabulation

Ici, il est plus simple de représenter le graphe de dépendance sur une grille où chaque case représente un couple (i, j). Ainsi, dans l'exemple considéré, les cases $(0, 2)$, $(1, 2)$, $(2, 3)$ et $(3, 4)$ dépendent de $(2, 2)$. Ces relations de dépendance sont indiqués sur le schéma suivant en rouge :



Pour trouver un tri topologique, il faut renverser ces dépendances : pour calculer (i, j) on va avoir besoin de toutes les cases au dessus et à gauche jusqu'à tomber sur la diagonale. On peut donc remplir de haut en bas et de gauche à droite, ce qui assure qu'à tout moment, lorsqu'on remplit une case, on a déjà calculé ses dépendances :



Cela correspond au programme suivant :

OCaml

```ocaml
let matmult t =
    let n = Array.length t in
    let cout = Array.make_matrix (n-1) (n-1) 0 in
    for i = n-2 downto 0 do
        for j = i+1 to n-2 do
            let m = ref (t.(i) * t.(i+1) * t.(j+1)
                        + cout.(i+1).(j)) in
            for k = i+1 to j-1 do
                m := min !m
                    (t.(i) * t.(k+1) * t.(j+1)
                        + cout.(i).(k)
                        + cout.(k+1).(j))
            done;
            cout.(i).(j) <- !m
        done
    done;
    cout.(0).(n-2)
```

Python

```python
def matmult(t) :
    n = len(t)
    cout = [ [ 0 ] * (n-1) for _ in range(n-1) ]
    for i in reversed(range(0,n-1)) :
        for j in range(i+1,n-1) :
            m = t[i] * t[i+1] * t[j+1]
              \ + matmult(t, i+1, j)
            for k in range(i+1, j) :
                m = min(m,
                    t[i] * t[k+1] * t[j+1]
                     \ + matmult(t, i, k)
                     \ + matmult(t, k+1, j))
            cout[i][j] = m
    return cout[0][n-2]
```

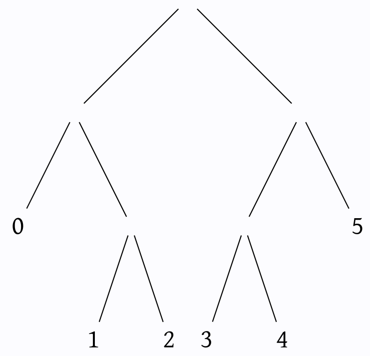
III.3.iv `OCaml` : Reconstruction

Pour recontruire un parenthésage on va constuire un arbre en plaçant dans un tableau l'arbre du calcul de $A_i \dots A_j$ à la case (i, j).

OCaml

```ocaml
type arbre = Feuille of int | Noeud of arbre * arbre

let matmult t =
    let n = Array.length t in
    let cout = Array.make_matrix (n-1) (n-1) 0 in
    let arbres = Array.make_matrix (n-1) (n-1) (Feuille (-1)) in
    for i = n-2 downto 0 do
        arbres.(i).(i) <- Feuille i;
        for j = i+1 to n-2 do
            let m = ref (t.(i) * t.(i+1) * t.(j+1)
                        + cout.(i+1).(j)) in
            let a = ref (Noeud(arbres.(i).(i),
                                arbres.(i+1).(j))) in
            for k = i+1 to j-1 do
                let v = t.(i) * t.(k+1) * t.(j+1)
                        + cout.(i).(k)
                        + cout.(k+1).(j)
                in
                if v < !m
                then begin
                    m := v;
                    a := Noeud(arbres.(i).(k),
                                arbres.(k+1).(j))
                end
            done;
            cout.(i).(j) <- !m;
            arbres.(i).(j) <- !a
        done
    done;
    arbres.(0).(n-2)
```

Exemple Dans l'exemple précédent on va renvoyer l'arbre



qui correspond au produit $(A_0(A_1A_2))((A_3A_4)A_5)$.

III.4 Plus longue sous-séquence croissante

On se donne un tableau t de n nombres entiers et on demande de déterminer un p-uplet $(i_1, \dots, i_p)$, où $t(i_1) < t(i_2) < \dots < t(i_p)$, pour lequel p est maximal.

III.4.i Notion de sous-problème et récurrence

On reprend ici directement ce qui avait été fait pour le premier exemple : on considère $f(i)$ la longueur du plus grand p-uplet $(i, i_2, \dots, i_p)$ où $t(i) < t(i_1) < \dots < t(i_p)$. Autrement dit, on oblige la sous-séquence à commencer par $t(i)$.

On a donc $f(n-1) = 1$ car elle ne peut contenir qu'une valeur. Ensuite, on considère

$$\Phi(i) = \{\ j \mid j > i \wedge t[j] > t[i]\ \}$$

et on a

$$\forall i < n-1, f(i) = \begin{cases} 1 & \text{si } \Phi(i) = \emptyset \\ 1 + \max_{j\in\Phi(i)} f(j) & \text{sinon.} \end{cases}$$

III.4.ii Récurrence naïve et mémoïsation

OCaml

```ocaml
let rec croissante t i =
    let n = Array.length t in
    if i = n-1
    then 1
    else let m = ref 1 in
        for j = i+1 to n-1 do
            if t.(j) > t.(i)
            then m := max !m (1 + croissante t j)
        done;
        !m
```

Python

```python
def croissante(t, i) :
    n = len(t)
    if i == n-1 :
        return 1
    m = 1
    for j in range(i+1,n) :
        if t[j] > t[i] :
            m = max(m,
                1 + croissante(t, j))
    return m
```

OCaml

```ocaml
let croissante t =
    let n = Array.length t in
    let cache = Hashtbl.create n in
    let rec aux i =
        try
            Hashtbl.find cache i
        with Not_found ->
            let v = if i = n-1
                    then 1
                    else let m = ref 1 in
                        for j = i+1 to n-1 do
                            if t.(j) > t.(i)
                            then m := max !m (1 + aux j)
                        done;
                        !m
            in Hashtbl.add cache i v; v
    in aux
```

Python

```python
cache = {}

def croissante(t, i) :
    n = len(t)
    if i in cache :
        return cache[i]

    if i == n-1 :
        v = 1
    else :
        m = 1
        for j in range(i+1,n) :
            if t[j] > t[i] :
                m = max(m,
                    1 + croissante(t, j))
        v = m
    cache[i] = v
    return v
```

III.4.iii Tabulation

On reprend ici aussi le même principe que pour la tabulation de la sous-séquence maximale, c'est-à-dire qu'on va remplir de droite à gauche.

OCaml

```ocaml
let croissante t =
    let n = Array.length t in
    let f = Array.make n 1 in
    for i = n-2 downto 0 do
        for j = i+1 to n-1 do
            if t.(j) > t.(i)
            then f.(i) <- 1 + f.(j)
        done
    done;
    f
```

Python

```python
def croissante(t) :
    n = len(t)
    f = [ 1 ] * n
    for i in reversed(range(n-1)) :
        for j in range(i+1, n) :
            if t[j] > t[i]
            then f[j] = 1 + f[j]
    return f
```

III.4.iv `OCaml` : Reconstruction

On va conserver dans un tableau le suivant potentiel d'un indice dans une plus grande sous-séquence croissante commençant par lui.

OCaml

```ocaml
let croissante t =
    let n = Array.length t in
    let f = Array.make n 1 in
    let suivant = Array.make n None in
    for i = n-2 downto 0 do
        for j = i+1 to n-1 do
            if t.(j) > t.(i) && f.(i) < 1 + f.(j)
            then begin
                f.(i) <- 1 + f.(j);
                suivant.(i) <- Some j
            end
        done;
    done;
    f, suivant

let sous_sequence_croissante t =
    let f, suivant = croissante t in
    let n = Array.length t in
    let mi = ref 0 in
    for i = 1 to n-1 do
        if f.(i) > f.(!mi)
```

```
    then mi := i
done;
let seq = ref [!mi] in
while suivant.(!mi) <> None do
    mi := Option.get suivant.(!mi);
    seq := !mi :: !seq
done;
List.rev !seq
```

III.5 Distance d'édition

Étant donné deux chaînes de caractères (on pourra considérer des `char vect` pour simplifier) A et B de longueurs respectives n et m, on veut transformer la chaîne A en la chaîne B en effectuant un minimum d'opérations parmi celles-ci :

- supprimer un caractère n'importe où
- insérer un nouveau caractère n'importe où
- changer un caractère en n'importe quel autre.

Ce nombre d'opération est appelée la distance d'édition de A à B. Écrire un algorithme permettant de la calculer.
Par exemple : chien $\to$ chen $\to$ chan $\to$ chat

III.6 Plus longue sous-séquence palindromique

On se donne une chaine de caractère et on cherche la plus longue chaine extraite qui soit un palindrome.
Par exemple, dans `abracadabra` il y a `aradara` comme plus long palindrome extrait.
Déterminer une plus longue sous-séquence palindromique d'une chaine de caractères.