# INF4705 — Analyse et conception d'algorithmes

## CHAPITRE 6 : ALGORITHMES DE PROGRAMMATION DYNAMIQUE

Afin de motiver cette approche, penchons-nous sur le calcul des coefficients binomiaux ou, de manière équivalente, du nombre de façons de choisir $k$ parmi $n$ objets distincts. (*Évidemment, on pourrait simplement utiliser la formule* $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ *mais nous découvrirons ainsi plusieurs des ingrédients de la programmation dynamique.)*

Relation de récurrence : $\binom{n}{k} = \binom{n-1}{k-1}$ (on choisit le $1^{er}$ (p.ex.) objet) $+ \binom{n-1}{k}$ (on ne le choisit pas) .
On a également que $\binom{n}{0} = \binom{n}{n} = 1$.

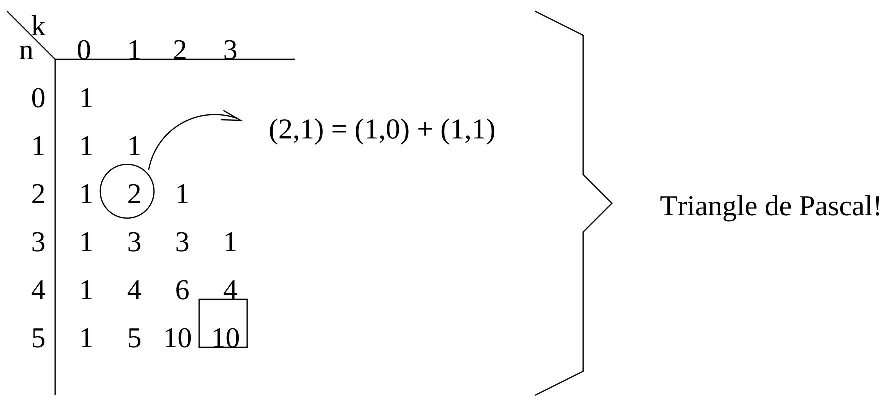
Développons

$$\binom{5}{3} = \binom{4}{2} + \binom{4}{3} = \binom{3}{1} + \binom{3}{2} + \binom{3}{2} + \binom{3}{3} = \binom{2}{0} + \binom{2}{1} + \binom{2}{1} + \binom{2}{2} + \binom{2}{1} + \binom{2}{2} + \binom{3}{3} = \cdots$$

$\binom{2}{1}$ apparaît déjà trois fois et $\binom{2}{2}$ deux fois.

Ainsi, dans une approche diviser-pour-régner pour ce problème, plusieurs sous-exemplaires sont répétés : on refait donc des calculs inutilement.
Si on prend "+1" comme instruction baromètre, on obtient un temps dans $\Omega(\binom{n}{k})$.
Si $k = n/2$ : $\binom{n}{n/2} = \frac{n!}{((n/2)!)^2} = \frac{n}{n/2} \cdot \frac{n-1}{n/2-1} \cdots \frac{n-(n/2-1)}{1} \geq (\frac{n}{n/2})^{n/2} = (\sqrt{2})^n \Rightarrow \Omega((\sqrt{2})^n)$

**idée :** Évitons de calculer plusieurs fois la même chose en conservant les résultats de nos calculs dans un tableau.




On a besoin de remplir $\leq (n+1) \cdot (k+1)$ cases.
Si $k = n/2 :\Rightarrow \mathcal{O}(n^2)$.

**Note :** Plutôt que de maintenir tout un tableau, il suffit de maintenir une seule ligne qu'on met à jour de droite à gauche : nous n´aurons ainsi plus besoin de $\Theta(nk)$ espace mais plutôt de $\Theta(k)$ espace.

## 6.1 Faire de la monnaie

Voici un problème plus conséquent, que nous connaissons déjà. Rappelons qu´il s´agit de minimiser le nombre de pièces utilisées pour totaliser un montant donné.

Étant donné $n$ types de pièces $1, \ldots, n$ de valeur $d_1, \ldots, d_n$ et une somme $N$ à rendre :
définissons un tableau $c[1 \ldots n, 0 \ldots N]$ où $c[i, j]$ est le nombre minimum de pièces requises pour totaliser un montant $j$ en n'utilisant que des pièces parmi $1 \ldots i$.

**Exemple 1** $N = 8,\ n = 3,\ d_1 = 1, d_2 = 4, d_3 = 6$

| *montant :* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* |
|---|---|---|---|---|---|---|---|---|---|
| $d_1 = 1$ | | | | | | | | | |
| $d_2 = 4$ | | | | | | | | | |
| $d_3 = 6$ | | | | | | | | | |

$c[i, j]$ : nous pouvons soit utiliser une pièce $i$ $(1 + c[i, j - d_i])$ ou ne pas en utiliser $(c[i - 1, j])$.

$$c[i, j] = \min(1 + c[i, j - d_i],\ c[i - 1, j])$$

**Ex.** $i = 2, j = 7 :\quad c[2, 7] = \min(1 + c[2, 3],\ c[1, 7])$

Nous avons défini notre tableau mais plusieurs questions se posent encore :

- Où récupérer notre réponse ? À la case $c[n, N]$.
- Nous avons posé une récurrence pour calculer les valeurs dans notre tableau mais nous devons aussi établir les valeurs frontière. Que sont-elles ?
  - Pour calculer $c[1, j]$ nous avons besoin de $c[0, j]$ : puisqu´aucune pièce n´est alors disponible, il est sensé de poser $c[0, j] = \infty$.
  - Nous avons aussi besoin de connaître $c[i, j]$ avec $j$ nul ou négatif.
    - Si $j = 0$ alors la somme à rendre est nulle et nous n´avons besoin d´aucune pièce : $c[i, j] = 0$.
    - Si $j < 0$ alors la somme à rendre est négative et il n´y a aucun moyen d´y arriver : $c[i, j] = \infty$.
- Dans quel ordre remplir notre tableau ? Ligne par ligne, de gauche à droite ; ou colonne par colonne, de haut en bas ; ou même en diagonale à partir de $c[1, 0]$..
- Peut-on ici aussi ne maintenir qu'une ligne ou colonne ? Oui, une ligne qu'on met à jour de gauche à droite. Pourquoi pas avec une colonne ?
- Comment retrouver les pièces qu'il faut donner ? On retrace nos pas à partir de la case réponse, identifiant laquelle des deux alternatives nous avons utilisée.

  **Remarque :** Il nous faut pour cela tout le tableau, pas seulement une ligne.

**Ex.**

| montant : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| $d_1 = 1$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| $d_2 = 4$ | 0 | 1 | 2 | 3 | 1 | 2 | 3 | 4 | 2 |
| $d_3 = 6$ | 0 | 1 | 2 | 3 | 1 | 2 | 1 | 2 | 2 |

Retraçons la solution :
$c[3, 8]\ \langle\rangle \rightarrow c[2, 8]\ \langle\rangle \rightarrow c[2, 4]\ \langle 2\rangle \rightarrow c[2, 0]\ \langle 2, 2\rangle$

**analyse :** On doit remplir un tableau $n \times (N + 1)$, ce qui donne un temps dans $\Theta(nN)$.
Est-ce un algorithme polynomial ?
Non car la taille d'un exemplaire est dans $\mathcal{O}(n + \lg N)$. Si $m = \lg N$ alors l'algorithme prend un temps dans $\Theta(n2^m)$.
On dit qu'il s'agit d'un algorithme *pseudo-polynomial*.
Et si on veut retrouver les pièces ?
En rebroussant jusqu'à $c[i, 0]$, on fera $\leq n$ pas vers le haut et $c[n, N]$ pas vers la gauche : $\mathcal{O}(n + c[n, N])$.

## 6.2 La programmation dynamique

**Principe :** On définit un tableau devant contenir la solution de chacun des sous-exemplaires potentiels de l'exemplaire originel. On remplit le tableau dans un certain ordre en utilisant les valeurs déjà présentes, puis on y lit notre réponse.
**Remarque 1 :** Un algorithme de *programmation dynamique* procède *de bas en haut* ("bottom-up") alors qu'un algorithme *diviser-pour-régner* procède *de haut en bas* ("top-down").
**Remarque 2 :** Définir un tableau qu'on doit remplir est avant tout une image qui permet de bien saisir la méthode. Il s'agit fondamentalement d'une manière de calculer une récurrence.

**Quand peut-on l'utiliser ?** Deux caractéristiques d'un problème font que la programmation dynamique est applicable comme technique de conception d'algorithme :

(i) **sous-structure optimale ("Principe d'optimalité") :** La solution optimale à un exemplaire est la combinaison des solutions optimales de certains de ses sous-exemplaires. Autrement dit, dans une suite optimale de décisions ou de choix, chaque sous-suite doit aussi être optimale.

(ii) **chevauchement des sous-exemplaires :** Une approche récursive résolverait alors de nombreuses fois le même sous-exemplaire. La programmation dynamique, elle, ne le résoudra qu'une seule fois et le stockera dans un tableau, ce qui est beaucoup plus efficace.

**Exemple 2** *Calcul du plus court chemin simple avec arcs de longueur possiblement négative. Soit un graphe non orienté avec les arcs suivants :*

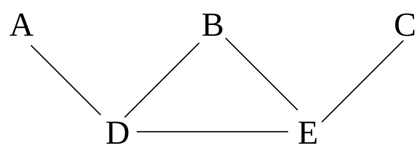
| *arc* | *(s,a)* | *(s,b)* | *(a,b)* | *(b,t)* |
|---|---|---|---|---|
| *longueur* | *1* | *1* | *-1* | *1* |

*Les plus courts chemins simples seront de longueur :*

| *orig,dest* | *s,a* | *s,b* | *s,t* | *a,b* | *a,t* | *b,t* |
|---|---|---|---|---|---|---|
| *longueur* | *0* | *0* | *1* | *-1* | *0* | *1* |

*Le plus court chemin simple de s à t ne peut pas être la combinaison des plus courts chemins simples de s à a et de a à t. Le principe d'optimalité ne s'applique donc pas ici.*

**Exemple 3** *Calcul du plus long chemin* simple *(c.-à-d. ne passant pas deux fois au même endroit).*




*Quelle est la longueur du plus long chemin simple de A à C ?*
*Même si on sait qu'il doit passer par B, la solution optimale ne s'obtient pas des solutions optimales pour A à B et B à C :*

**de A à B :** *A,D,E,B* $\Rightarrow 4$

**de B à C :** *B,D,E,C* $\Rightarrow 4$

**de A à C :** *A,D,B,E,C* $\Rightarrow 5 < 4 + 4 - 1$

*Le principe d'optimalité ne s'applique donc pas ici.*

**Quatre décisions à prendre :**

– Comment définir notre tableau ? (Quels sont les sous-exemplaires d'intérêt ?)
– Dans quel ordre le remplir ? (Quelle est la récurrence ?)
– Quelles sont les valeurs frontière ?
– Où récupérer notre réponse (ou nos réponses) ?

**Schéma général :**

**fonction** programmation_dynamique($x$ {exemplaire}) : {solution}

  définir un tableau $T$ à $d$ dimensions ;

  initialiser les valeurs frontière dans $T$ ;

  **pour** $i_1 = deb_1$ **à** $fin_1$ **faire**

    $\vdots$

    **pour** $i_d = deb_d$ **à** $fin_d$ **faire**

      $T[i_1, \ldots, i_d] \leftarrow$ expression utilisant des cases déjà calculées et $x$ ;

  **retourner** expression utilisant des cases de $T$ ;

Lorsque ce que nous cherchons se trouve dans une seule case du tableau, nous pouvons possiblement gagner de l'efficacité grâce aux *fonctions à mémoire*. Combinons l'élégance de la récursivité avec l'efficacité de la programmation dynamique.

**idée :** Utilisons un tableau global $T$ avec une fonction récursive $f$ :

**fonction** $f(x_1, \ldots, x_d$ {paramètres})

  **si** $T[x_1, \ldots, x_d] \neq$ la valeur d'initialisation **alors retourner** $T[x_1, \ldots, x_d]$ ;

  $s \leftarrow$ calcul récursif de $f(x_1, \ldots, x_d)$ ;

  $T[x_1, \ldots, x_d] \leftarrow s$ ; {on note la valeur de la solution}

  **retourner** $s$ ;

On gagne à ne pas calculer inutilement certaines cases du tableau mais on perd la possibilité de sauver de l'espace en ne maintenant qu'une fraction du tableau (p.e. une ligne).

## 6.3 Sac à dos

**Rappel :** Étant donné $n$ objets $\{1, \ldots, n\}$, de poids et valeur positifs $w_i$ et $v_i$ resp., et un sac à dos pouvant supporter $W$, quels objets (non fragmentés) devrais-je mettre dans mon sac à dos afin de maximiser la valeur totale ?

Définissons un tableau $V[1 \ldots n, 0 \ldots W]$ où $V[i,j]$ est la valeur maximum d'un contenu du sac de poids total n'excédant pas $j$ et n'utilisant que les objets $1 \ldots i$.

**Exemple 4** $n = 4$, $W = 7$

| *poids maximum :* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* |
|---|---|---|---|---|---|---|---|---|
| $w_1 = 1, v_1 = 1$ | | | | | | | | |
| $w_2 = 2, v_2 = 6$ | | | | | | | | |
| $w_3 = 5, v_3 = 18$ | | | | | | | | |
| $w_4 = 6, v_4 = 22$ | | | | | | | | |

$V[i,j]$ : nous pouvons soit utiliser l'objet $i$ ($v_i + V[i-1, j-w_i]$) ou ne pas l'utiliser ($V[i-1,j]$).

$$V[i,j] = \max(v_i + V[i-1, j-w_i],\ V[i-1,j])$$

- Où est notre réponse ? À la case $V[n, W]$.
- Valeurs frontière ? $V[0,j] = 0, \forall j \geq 0;\ V[i,j] = -\infty, \forall j < 0$.
- Dans quel ordre remplir notre tableau ? Comme pour "faire de la monnaie".

**Ex.**

| poids maximum : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $w_1 = 1, v_1 = 1$ | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $w_2 = 2, v_2 = 6$ | 0 | 1 | 6 | 7 | 7 | 7 | 7 | 7 |
| $w_3 = 5, v_3 = 18$ | 0 | 1 | 6 | 7 | 7 | 18 | 19 | 24 |
| $w_4 = 6, v_4 = 22$ | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 24 |

Retraçons la solution : $V[4,7]\ \langle\rangle \rightarrow V[3,7]\ \langle\rangle \rightarrow V[2,2]\ \langle 3\rangle \rightarrow V[1,0]\ \langle 3,2\rangle \rightarrow V[0,0]\ \langle 3,2\rangle$

**analyse :**
On doit remplir un tableau $n \times (W+1)$, ce qui donne un temps dans $\Theta(nW)$.
Et si on veut retrouver les objets ?
En rebroussant jusqu'à $V[0,0]$, on fera $n$ pas vers le haut : $\Theta(n)$.

## 6.4 Voyageur de commerce

**Rappel :** Une tournée d'un ensemble de $n$ points est un itinéraire passant par chacun exactement une fois et retournant au point de départ. Quelle est la tournée la plus courte ?

Sans perte de généralité, nous identifierons ces points par les entiers $\{1, 2, \ldots, n\}$ et supposerons que notre tournée débute à 1. Nous dénoterons la distance du point $i$ au point $j$ par $d_{ij}$.

Soit $D[i, S]$ la distance d'un plus court chemin partant de $i$, passant par tous les points de $S$ et se terminant à 1. Nous pouvons établir la récurrence suivante :

$$D[i, S] = \min_{j \in S}\{d_{ij} + D[j, S \setminus \{j\}]\}$$

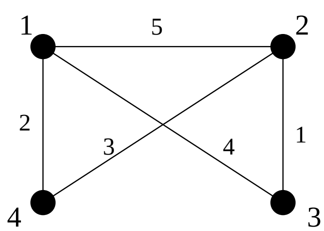
Définissons à cet effet un tableau $D$ de dimensions $(n-1) \times (2^{n-1} - 1)$.

**Exemple 5** *avec* $n = 4$

| $i$ $S$ | $\{\}$ | $\{2\}$ | $\{3\}$ | $\{4\}$ | $\{2,3\}$ | $\{2,4\}$ | $\{3,4\}$ |
|---|---|---|---|---|---|---|---|
| 2 | | | | | | | |
| 3 | | | | | | | |
| 4 | | | | | | | |

- Où est notre réponse ? C'est $D[1, \{2, 3, \ldots, n\}]$.
- Valeurs frontière ? $D[i, \{\}] = d_{i1}, \forall\ 2 \leq i \leq n$
- Dans quel ordre remplir notre tableau ?

**Exemple 6**




| $i$ $S$ | $\{\}$ | $\{2\}$ | $\{3\}$ | $\{4\}$ | $\{2,3\}$ | $\{2,4\}$ | $\{3,4\}$ |
|---|---|---|---|---|---|---|---|
| 2 | *5* | - | *5* | *5* | - | - | $\infty$ |
| 3 | *4* | *6* | - | $\infty$ | - | *6* | - |
| 4 | *2* | *8* | $\infty$ | - | *8* | - | - |

$$D[1, \{2,3,4\}] = \min\{d_{12} + D[2, \{3,4\}], d_{13} + D[3, \{2,4\}], d_{14} + D[4, \{2,3\}]\} = \min\{\infty, 10, 10\} = 10$$

Et si on veut retrouver la tournée ?
On maintient un autre tableau de mêmes dimensions mais contenant *argmin*, l'index $j$ minimisant l'expression dans la récurrence.

**analyse :**
On doit remplir presque tout le tableau et $D[i, S]$ nécessite l'examen de $|S|$ autres cases, ce qui donne un temps dans $\mathcal{O}(n^2\, 2^n)$.

## 6.5 Plus courts chemins

Soit $G = (N, A)$ un graphe orienté avec $N = \{1, \ldots, n\}$ et dont les arcs ont une longueur non négative. Quelle est la longueur du plus court chemin entre chaque paire de nœuds ?
Le principe d'optimalité s'applique :
si $\alpha \cdot \beta$ (où $\alpha = \langle i, \ldots, k\rangle$ et $\beta = \langle k, \ldots, j\rangle$) est un plus court chemin de $i$ à $j$ alors $\alpha$ et $\beta$ doivent aussi être les plus courts.

Définissons un tableau $D[1 \ldots n, 1 \ldots n, 0 \ldots n]$ où $D[i, j, k]$ est la longueur du plus court chemin de $i$ à $j$ dont les sommets intermédiaires sont parmi $1, \ldots, k$.

$D[i, j, k]$ : nous pouvons soit passer par $k$ ($D[i, k, k-1] + D[k, j, k-1]$) ou ne pas y passer ($D[i, j, k-1]$).

$$D[i, j, k] = \min(D[i, k, k-1] + D[k, j, k-1],\ D[i, j, k-1])$$

- Où est notre réponse ? Aux cases $D[*, *, n]$.
- Valeurs frontière ? $D[i, j, 0] =$ longueur de $(i, j)$.
- Peut-on sauver de l'espace ?
  On pourrait éliminer la $3^e$ dimension (k) à condition de garantir que $D[i, k, k]$ et $D[k, j, k]$ sont mis à jour après $D[i, j, k]$. Mais

  $$D[*, k, k] = \min(D[*, k, k-1]+D[k, k, k-1],\ D[*, k, k-1]) = \min(D[*, k, k-1]+0,\ D[*, k, k-1]) = D[*, k, k-1]$$

  et de façon similaire pour $D[k, *, k]$, alors les $k^{\text{ième}}$ ligne et colonne ne changent pas.

- Dans quel ordre remplir notre nouveau tableau $D[1 \ldots n, 1 \ldots n]$ ? Arbitraire.

On obtient l'algorithme de Floyd :

**fonction** Floyd($L[1..n, 1..n]$ :longueurs) : tableau $[1..n, 1..n]$
    tableau $D[1..n, 1..n]$ ;
    $D \leftarrow L$ ;
    **pour** $k = 1$ **à** $n$ **faire**
        **pour** $i = 1$ **à** $n$ **faire**
            **pour** $j = 1$ **à** $n$ **faire**
                $D[i, j] \leftarrow \min(D[i, j], D[i, k] + D[k, j])$ ;
    **retourner** $D$ ;

Afin de retrouver les plus courts chemins une fois l´algorithme terminé, on met à jour un tableau $n \times n$ qui contient, pour chaque paire de noeuds $\langle i, j\rangle$, la dernière itération à laquelle $D[i, j]$ a été modifiée. Cette information nous permet de retracer nos pas.

**analyse :**

**temps de calcul :** $\Theta(n^3)$. On pourrait aussi utiliser Dijkstra $n$ fois : $\Theta(n^3)$ ou $\Theta(an \lg n + n^2 \lg n)$. Notre choix peut dépendre de la densité du graphe.
Et si on veut retrouver un chemin ? $\mathcal{O}(n)$

**espace mémoire :** $\Theta(n^2)$.