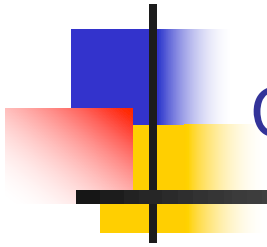
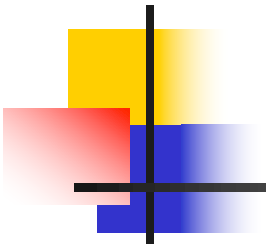


Chapitre 8

La programmation dynamique



Introduction à la programmation dynamique

- La programmation dynamique est une technique de conception d'algorithmes qui consiste à résoudre un problème en solutionnant une séquence de sous problèmes, du plus petit vers le plus grand, en sauvegardant les solutions intermédiaires dans un tableau.
  - Ce tableau, occupant un certain espace mémoire, permet d'obtenir la solution au problème initial
- Ce type d'approche exploite donc un compromis espace-temps
  - Le tableau occupe un espace mémoire additionnel, mais permet d'obtenir plus rapidement la solution
- Le point de départ de cette démarche est, habituellement, une relation de récurrence permettant d'obtenir la solution de l'instance initiale à partir de la solution d'instances plus petites
  - Mais nous utilisons une approche du « bas vers le haut »: nous obtenons d'abord les solutions aux plus petites instances pour construire les solutions aux instances plus grandes

Calcul du coefficient binomial

- Concevons un algorithme de programmation dynamique pour calculer

$$C(n,k) \stackrel{\text{def}}{=} \binom{n}{k} \stackrel{\text{def}}{=} \frac{n!}{k!(n-k)!}$$

sans effectuer de multiplications.

- Le coefficient C(n,k) est appelé coefficient binomial, car il apparaît dans la formule du binôme:

$$(a+b)^n = \sum_{i=0}^{n} C(n,i) a^i b^{n-i}$$

- Il est bien connu que C(n,k) satisfait la récurrence suivante:

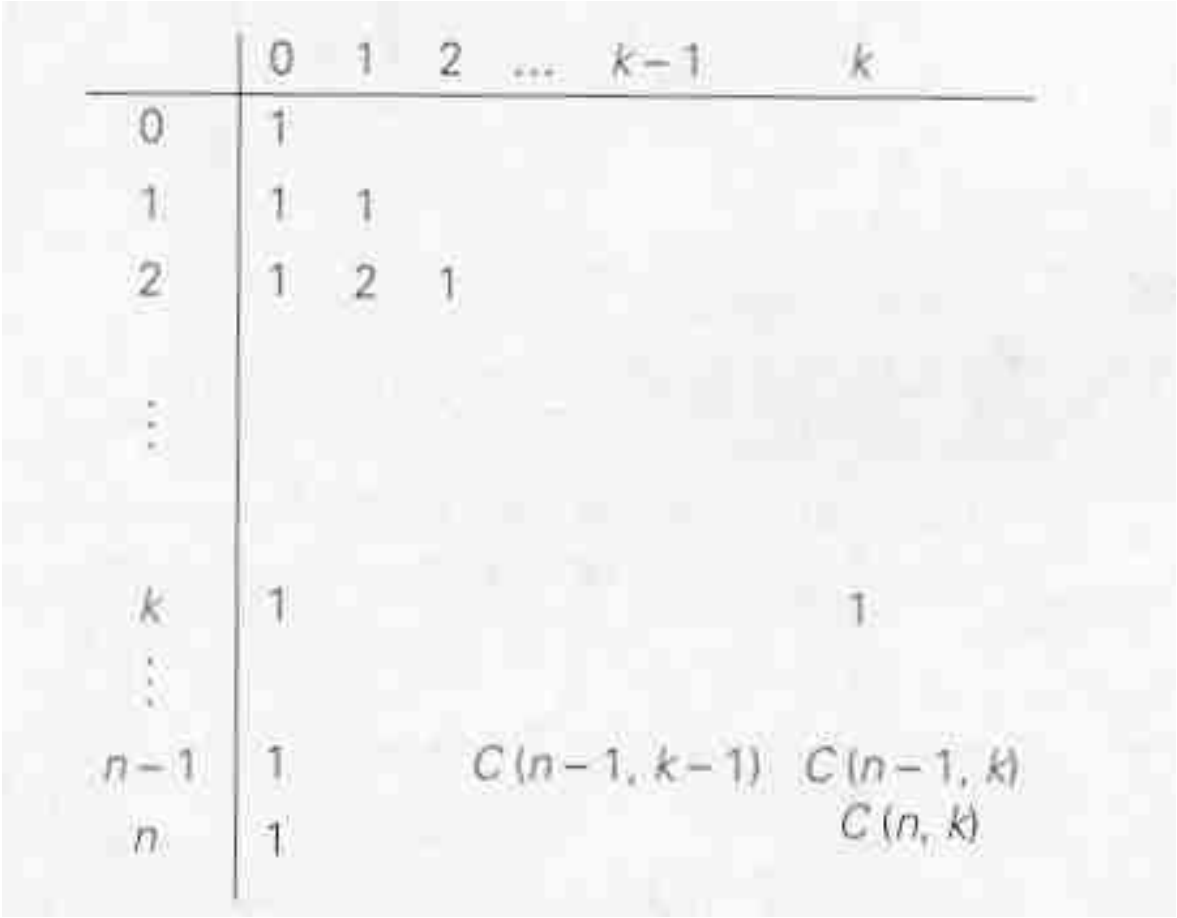
$$C(n,k) = C(n-1,k-1) + C(n-1,k) \quad \text{pour } n > k > 0$$
$$\text{avec } C(n,0) = C(n,n) = 1$$

Une mauvaise idée

```
ALGORITHME BinomRec(n, k)
//Entrée: deux entiers n ≥ k ≥ 0
//Sortie: C(n,k)
    if k = 0 or k = n return 1
    else return BinomRec(n-1,k-1) + BinomRec(n-1,k)
```

- Cette récurrence nous suggère l'algorithme diviser pour régner ci-dessus
- Or, chaque valeur intermédiaire C(i,j) est calculée plusieurs fois
  - Ex: Pour obtenir C(5,3), il faut calculer C(4,2) et C(4,3) et chacun de ces coefficients nécessite le calcul de C(3,2)
- Le résultat final est obtenu en additionnant plusieurs valeurs « 1 »
- Pour obtenir la valeur de C(n,k), cet algorithme effectuera alors Ω(C(n,k)) additions !!
- Nous avions obtenu la même explosion du temps d'exécution pour le calcul de F(n) à l'aide de la récurrence F(n) = F(n-1) + F(n-2)

Une meilleure idée: la programmation dynamique



	0	1	2	...	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
⋮						
k	1					1
⋮						
$n-1$	1				$C(n-1, k-1)$	$C(n-1, k)$
n	1					$C(n, k)$

- Pour éviter de calculer C(i,j) plusieurs fois, calculons chaque valeur de C(i,j) une seule fois du bas vers le haut pour $0 \leq i \leq n$, $0 \leq j \leq k$
- Calculons le tableau C(i,j) rangée par rangée, de gauche à droite, en débutant avec la rangée i = 0 (et en terminant avec i = n). C'est le triangle de Pascal!

L'algorithme Binomial(n,k)

- Voici donc l'algorithme Binomial(n,k) construisant ce tableau C(i,j)

ALGORITHM *Binomial*(n, k)

```
//Computes C(n, k) by the dynamic programming algorithm
//Input: A pair of nonnegative integers n ≥ k ≥ 0
//Output: The value of C(n, k)
for i ← 0 to n do
    for j ← 0 to min(i, k) do
        if j = 0 or j = i
            C[i, j] ← 1
        else C[i, j] ← C[i − 1, j − 1] + C[i − 1, j]
return C[n, k]
```

Analyse de l'algorithme Binomial(n,k)

- Utilisons l'addition pour l'opération de base
- Le temps d'exécution T(n,k) est donc le nombre d'additions effectuées par Binomial(n,k) pour obtenir la valeur du coefficient C(n,k)
  - C'est le même en pire cas et en meilleur cas, car nous avons une seule instance par paire (n,k)
- Chaque calcul de C(i,j) = C(i-1,j-1) + C(i-1,j) nécessite 1 addition
- Le nombre d'additions est alors donné par le nombre d'entrées (i,j) qui existent avec $1 \leq i \leq n$ et $1 \leq j \leq k$ (voir le tableau en page 5)
  - Les rangées i de 1 à k forment un triangle de k(k-1)/2 entrées
  - Les rangées i de k+1 à n forment un rectangle de (n-k)k entrées

- Nous avons donc: $T(n,k) = k(k-1)/2 + (n-k)k = nk - k^2/2 - k/2$

- C'est donc vraiment beaucoup plus efficace que BinomRec(n,k)

Analyse de l'algorithme Binomial(n,k) (suite)

- Nous avons donc $T(n,k) \leq nk$
- Nous avons également:

$$T(n,k) = nk - k^2/2 - k/2 \geq nk - nk/2 - (n/2)\, k/2 = nk/4$$

- Alors: $nk/4 \leq T(n,k) \leq nk$
- Alors **$T(n,k) \in \Theta(nk)$**
- L'espace utilisée par Binomial(n,k) est également $\Theta(nk)$
  - Cependant, il n'est pas nécessaire de stocker tout le tableau
  - Il suffit d'utiliser un vecteur de k éléments pour la ligne courante et de mettre à jour ce vecteur de la gauche vers la droite.
  - Dans cette version, l'espace utilisée sera seulement de $\Theta(k)$

Le problème du sac à dos

- Nous avons n objets (items) avec des poids respectifs $w_1,\ldots, w_n$ et valeurs respectives $v_1,\ldots, v_n$
  - Chaque poids w_i est un entier non négatif et les valeurs v_i sont réelles
- Nous avons un sac à dos pouvant supporter un poids total maximal W
  - W est un entier non négatif
- L'objectif est de trouver le sous-ensemble d'objets de valeur maximale et dont le poids n'excède pas W (la capacité du sac à dos)
  - C'est un problème pertinent pour un voleur ☺

- L'algorithme force brute pour ce problème consiste à énumérer chacun des 2^n sous-ensembles possibles et, pour chaque sous-ensemble, de calculer la somme des valeurs de chaque objet lorsque la somme des poids n'excède pas W
  - Le temps d'exécution de cet algorithme est donc $\Omega(2^n)$

Programmation dynamique et sac à dos

- La première étape de conception d'un algorithme de programmation dynamique consiste à exprimer la solution d'une instance en fonction de la solution d'une instance plus petite

- Considérons une instance constituée des i premiers objets, $0 \leq i \leq n$,
  - avec les poids $w_1,\ldots,w_i$ et valeurs $v_1,\ldots,v_i$
  - et d'un sac à dos de capacité j, $0 \leq j \leq W$,
- Soit V[i,j] = la valeur de la **solution optimale** de cette instance
- Donc V[i,j] = la valeur du sous-ensemble de valeur maximale des i premiers objets dont le poids total n'excède pas j

- Il existe deux types **possibles** de sous ensembles des i premiers objets dont le poids total n'excède pas j:
  - Ceux qui incluent l'objet i
  - Ceux qui n'incluent pas l'objet i

Programmation dynamique et sac à dos (suite)

- Parmi les sous-ensembles (des i premiers objets) qui n'incluent pas l'objet i, la valeur du sous-ensemble optimal est V[i-1,j]
- Parmi les sous-ensembles qui incluent l'objet i (alors, $j - w_i \geq 0$), la valeur du sous-ensemble optimal est $v_i + V[i-1,j-w_i]$
- La valeur maximale du sous-ensemble optimal des i premiers objets est donc le maximum de ces deux dernières valeurs
- Nous obtenons donc la récurrence:

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1,j-w_i]\} & \text{si} \quad j - w_i \geq 0 \\ V[i-1,j] & \text{si} \quad j - w_i < 0 \end{cases}$$

- Nous avons également les conditions initiales suivantes:

$$\begin{aligned} V[0,j] &= 0 \quad \text{pour } j \geq 0 \\ V[i,0] &= 0 \quad \text{pour } i \geq 0 \end{aligned}$$

Programmation dynamique et sac à dos (suite)

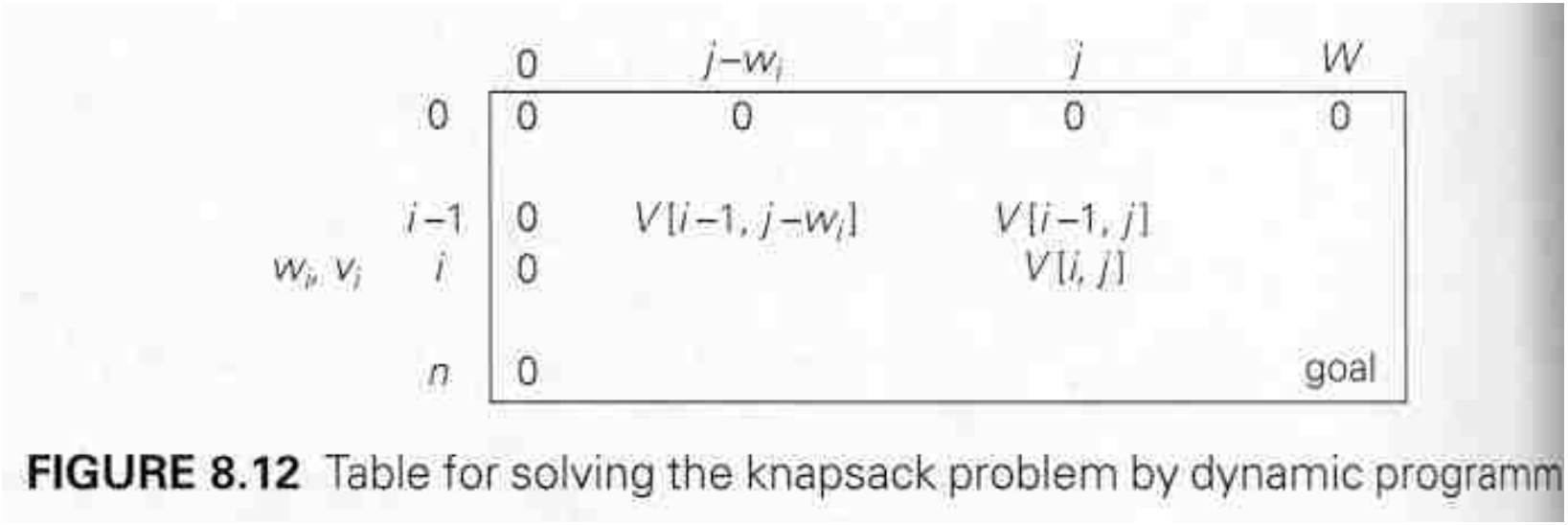


FIGURE 8.12 Table for solving the knapsack problem by dynamic programm

- Pour trouver la valeur V[n,W] de la solution optimale du problème initial il suffit de construire le tableau V[i,j] du haut vers le basà l'aide de la récurrence précédente.

Algorithme DPKnapsack

Algorithm *DPKnapsack*$(w[1..n], v[1..n], W)$
//Solves the knapsack problem by dynamic programming (bottom up)
//Input: Arrays $w[1..n]$ and $v[1..n]$ of weights and values of n items,
// knapsack capacity W
//Output: Table $V[0..n, 0..W]$ that contains the value of an optimal
// subset in $V[n, W]$ and from which the items of an optimal
// subset can be found
for $i \leftarrow 0$ **to** n **do** $V[i, 0] \leftarrow 0$
for $j \leftarrow 1$ **to** W **do** $V[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** W **do**
    if $j - w[i] \geq 0$
      $V[i, j] \leftarrow \max\{V[i-1, j], v[i] + V[i-1, j-w[i]]\}$
    else $V[i, j] \leftarrow V[i-1, j]$
return $V[n, W]$, V

Sac à dos: exemple numérique

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	37

- La valeur maximale est alors de V[4,5] = 37\$
- Pour trouver la composition du sac à dos, faire un retour arrière:
  - Puisque V[4,5] > V[3,5], **l'item 4 fut inclus** dans le sac à dos
  - Puisque w_4 = 2, la capacité résiduelle du sac est de 5-2 = 3
  - Maintenant V[3,3] = V[2,3]. Alors **l'item 3 ne fut pas inclus**
  - V[2,3] > V[1,3]. Alors **l'item 2 fut inclus**.
  - Or w_2 = 1. La capacité résiduelle du sac = 3-1 = 2.
  - V[1,2] > V[0,2]. Alors **l'item 1 fut inclus**.
  - Composition du sac = items 4,2,1

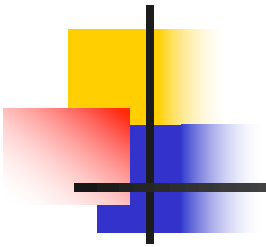
Algorithme pour trouver la composition finale du sac

Algorithm *OptimalKnapsack*$(w[1..n], v[1..n], W)$
//Finds the items composing an optimal solution to the knapsack problem
//Input: Arrays $w[1..n]$ and $v[1..n]$ of weights and values of n items,
// knapsack capacity W, and table $V[0..n, 0..W]$ generated by
// the dynamic programming algorithm
//Output: List $L[1..k]$ of the items composing an optimal solution
$k \leftarrow 0$ //size of the list of items in an optimal solution
$j \leftarrow W$ //unused capacity

for $i \leftarrow n$ **downto** 1 **do**
  if $V[i, j] > V[i-1, j]$
    $k \leftarrow k+1$; $L[k] \leftarrow i$ //include item i
    $j \leftarrow j - w[i]$
return L

Analyse de DPKnapsack et OptimalKnapsack

- Pour DPKnapsack:
  - Chaque évaluation de V[i,j] nécessite un temps constant $\Theta(1)$.
  - Le tableau V possède n+1 rangées et W+1 colonnes
  - **Le temps total requis est alors $\in \Theta(nW)$**
  - **L'espace requis pour ce tableau est également $\in \Theta(nW)$**
- Pour OptimalKnapsack:
  - Pour chaque valeur V[i,j] examinée, on compare d'abord à V[i-1,j].
    - Si V[i,j] > V[i-1,j]: on va en $V[i-1,j-w_i]$
    - On recommence ensuite avec V[i-1,j] ou $V[i-1,j-w_i]$
  - Alors, pour chaque rangée, nous examinons toujours V[i-1,j] et, parfois, on examine également $V[i-1,j-w_i]$
    - On examine alors au plus 2 valeurs de V par rangée
  - Le nombre de valeurs de V[i,j] examinées au total sera alors au plus de 2n. **Le temps d'exécution est alors $\in$ O(n).**

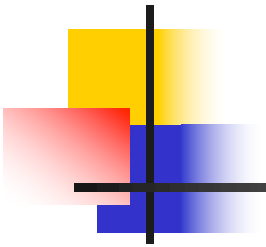


Distance d'édition

- Ces deux chaînes de caractères sont-elles identiques?
  - « 123 boulevard René-Lévesque #6 »
  - « 123 boul. Rene Levesque apt. 6 »
- La réponse est non. Dans la deuxième chaîne, le mot boulevard est abrégé, il n'y a pas d'accents, le trait d'union est manquant et les symboles pour désigner l'appartement diffèrent.
- Les données obtenues par diverses sources divergent souvent en plusieurs points. Un programme informatique devrait cependant traiter ces deux chaînes de caractères comme identiques.
- Nous reformulons donc la question: « Ces deux chaînes de caractères sont-elles presque identiques? »
- La réponse dépend de notre définition du mot « presque » et c'est là que la notion de « distance d'édition » entre en jeu.

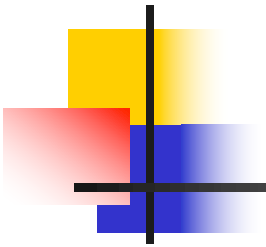
Définition: la distance d'édition

- La distance d'édition est le nombre minimal d'insertions, de suppressions et de substitutions de caractères qu'il faut appliquer à la première chaîne de caractères pour obtenir la deuxième chaîne de caractères.
- Ainsi, la distance entre les chaînes
  « 123 boul**evard** Ren**é-**L**é**vesque **#**4» et
  « 123 boul**.** Ren**e** L**e**vesque **apt.** 4»
  est de 13.
  - 1 substitution du caractère «e» par le caractère «.»
  - 4 suppressions des caractères «vard»
  - 2 substitutions des caractères é en caractères e
  - 1 substitution du trait d'union par l'espace
  - 1 substitution du caractère # en caractère a
  - 4 additions des caractères «pt._»



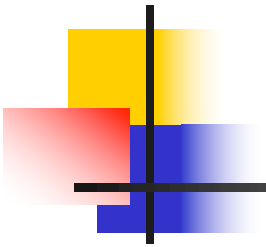
Définition du tableau

- Nous désirons calculer la distance d'édition entre deux chaînes de caractères A et B.
- Pour tout algorithme de programmation dynamique, il faut d'abord trouver une récurrence qui définit le problème en fonction d'instances plus petites. Comment pouvons-nous réduire la taille de l'instance?
- Nous pouvons considérer que les i premiers caractères de la première chaîne et les j premiers caractères de la deuxième chaîne.
- Nous définissons le tableau suivant. Soit D[i,j] la distance d'édition entre deux chaînes: la première étant formée des i premiers caractères de A et la deuxième des j premiers caractères de B.



Cas de base

- Si la chaîne B est vide (j = 0), alors les i caractères de la chaîne A sont supprimés.
  - Nous avons donc D[i, 0] = i.
  - Ex.: Il faut supprimer 7 caractères dans la chaîne «bonjour» pour obtenir la chaîne vide «».
- Si la chaîne A est vide (i = 0), alors les j caractères sont insérés dans la chaîne B.
  - Nous avons donc D[0,j] = j
  - Ex.: Il faut ajouter 7 caractères à la chaîne vide «» pour obtenir la chaîne «bonjour».



Récurrence

- Supposons que le dernier caractère de la chaîne A est identique au dernier caractère la chaîne B, c'est-à-dire A[i] = B[j]. Nous pouvons dire que ces deux caractères n'ont pas été modifiés.
- Conséquemment, la distance d'édition se calculera sur les i-1 premiers caractères de la chaîne A et les j-1 premiers caractères de la chaîne B.
  - Nous avons: D[i,j] = D[i-1, j-1]
- Ex.:
  - La distance d'édition entre A[1..i]=«abab» et B[1..j]=«abbb» est la même qu'entre A[1..i-1]=«aba» et B[1..j-1]=«abb».

Récurrence (suite)

- Si les derniers caractères de chaque chaîne diffèrent (A[i] ≠ B[j]), il y a eu une insertion, une suppression ou une substitution.
  - Dans le cas d'une insertion, nous avons D[i,j] = D[i, j-1] + 1
    - Ex.: La distance d'édition entre A[1..i] = «ab» et B[1..j] = «abc» est un de plus que la distance d'édition entre A[1..i] = «ab» et B[1..j-1] = «ab».
  - Dans le cas d'une suppression, nous avons D[i,j] = D[i-1, j] + 1
    - Ex.: La distance d'édition entre A[1..i] = «abc» et B[1..j] = «ab» est un de plus que la distance d'édition entre A[1..i-1] = «ab» et B[1..j] = «ab».
  - Dans le cas d'une substitution nous avons D[i,j] = D[i-1,j-1] + 1
    - Ex: La distance d'édition entre A[1..i] = «abc» et B[1..j] = «abb» est un de plus que la distance d'édition entre A[1..i-1] = «ab» et B[1..j-1] = «ab».
- Entre les trois possibilités, nous prenons celle qui minimise la distance d'édition: D[i,j] = min(D[i,j-1], D[i-1,j], D[i-1,j-1]) + 1

Récurrence (suite)

- Nous obtenons donc la récurrence suivante.

$$D[i,j] = \begin{cases} i & \text{si } j = 0 \\ j & \text{si } i = 0 \\ D[i-1, j-1] & \text{si } A[i] = B[j] \\ \min(D[i, j-1], D[i-1, j], D[i-1, j-1]) + 1 & \text{si } A[i] \neq B[j] \end{cases}$$

Le pseudo-code

Algorithme 1 : DistanceEdition($A[1..n], B[1..m]$)

// Retourne la distance d'édition entre les chaînes A et B ainsi
// que le tableau qui a été créé pour calculer cette distance
// Entrée : Deux chaînes A et B de n et m caractères
// Sortie : La distance d'édition et le tableau utilisé pour calculer cette distance
Crée un tableau $D[0..n, 0..m]$ de dimensions $n+1 \times m+1$
pour $i = 0..n$ **faire** $D[i, 0] \leftarrow i$
pour $j = 1..m$ **faire** $D[0, j] \leftarrow j$
pour $i = 1..n$ **faire**
  pour $j = 1..m$ **faire**
    si $A[i] = B[j]$ **alors**
      $D[i, j] \leftarrow D[i-1, j-1]$
    sinon
      $D[i, j] \leftarrow \min(D[i, j-1], D[i-1, j], D[i-1, j-1]) + 1$
retourner $D[n, m]$, D

Exemple

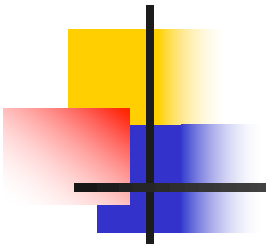
		M	O	N	S	I	E	U	R
	0	1	2	3	4	5	6	7	8
M	1	0	1	2	3	4	5	6	7
O	2	1	0	1	2	3	4	5	6
N	3	2	1	0	1	2	3	4	5
C	4	3	2	1	1	2	3	4	5
E	5	4	3	2	2	2	2	3	4
A	6	5	4	3	3	3	3	3	4
U	7	6	5	4	4	4	4	3	4

Analyse de l'algorithme DistanceÉdition

- La première boucle s'exécute en temps $\Theta(n)$.
- La deuxième boucle s'exécute en temps $\Theta(m)$.
- En prenant A[i] = B[i] comme instruction baromètre, les deux boucles imbriquées s'exécute en temps

$$\sum_{i=1}^{n}\sum_{j=1}^{m} 1 = \sum_{i=1}^{n} m = nm \in \Theta(nm)$$

- En appliquant la règle du maximum, on en déduit que l'algorithme DistanceÉdition s'exécute en temps $\Theta(nm)$.



Lister les éditions

- Pour découvrir la séquence minimale d'éditions nécessaire pour passer de la chaîne A à la chaîne B, on peut remonter le tableau D en commençant par la cellule D[i,j] pour i=n et j=m et en analysant comment la valeur de cette cellule a été calculée.
  - Si A[i] = B[j], alors il n'y a eu aucune édition (donc le caractère A[i] a été copié dans B[j]).
  - Si D[i,j] = D[i,j-1]+1 alors le caractère B[j] a été inséré.
  - Si D[i,j] = D[i-1,j]+1 alors le caractère A[i] a été supprimé.
  - Si D[i,j] = D[i-1,j-1]+1 alors le caractère A[i] a été substitué par B[i].
- Après avoir analysé comment D[i,j] a été calculé, on fixe i et j aux coordonnées de la cellule ayant servi au calcul de D[i,j].

Pseudo-code

Algorithme 2 : ImprimeÉditions($A[1..n], B[1..m], D[0..n, 0..m]$)

```
// Imprime, en ordre inverse, les éditions appliquées à la chaîne A pour obtenir la chaîne B.
// Entrée : Les chaînes de caractères A et B ainsi que le tableau D produit par la fonction DistanceÉdition.
// Sortie : La liste des éditions en ordre inverse.
i ← n
j ← m
tant que i > 0 ∨ j > 0 faire
    si i > 0 ∧ j > 0 ∧ A[i] = B[j] alors
        Imprimer « Copie de A[i] »
        i ← i − 1, j ← j − 1
    sinon
        si j > 0 ∧ D[i, j] = D[i, j − 1] + 1 alors
            Imprimer « Insertion de B[j] »
            j ← j − 1
        sinon si i > 0 ∧ D[i, j] = D[i − 1, j] + 1 alors
            Imprimer « Suppression de A[i] »
            i ← i − 1
        sinon
            Imprimer « Substitution de A[i] par B[j] »
            i ← i − 1, j ← j − 1
```

Exemple

		M	O	N	S	I	E	U	R
	0	1	2	3	4	5	6	7	8
M	1	0	1	2	3	4	5	6	7
O	2	1	0	1	2	3	4	5	6
N	3	2	1	0	1	2	3	4	5
C	4	3	2	1	1	2	3	4	5
E	5	4	3	2	2	2	2	3	4
A	6	5	4	3	3	3	3	3	4
U	7	6	5	4	4	4	4	3	4

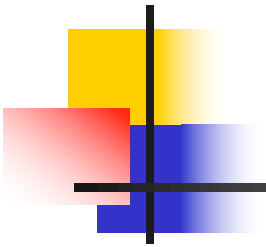
- Insertion de R
- Copie de U
- Suppression de A
- Copie de E
- Insertion de I
- Substitution de C par S
- Copie de N
- Copie de O
- Copie de M

Analyse de l'algorithme ImprimeÉditions

- En pire cas, une seule des variables i et j est décrémentée à chaque itération. L'algorithme s'exécute donc en temps $C_{WORST}(n,m) = n + m \in \Theta(n+m)$.
- En meilleurs cas, les deux variables i et j sont décrémentées à chaque itération jusqu'à ce que l'une d'entre elles devienne nulle. Il y a donc un maximum de min(n, m) itérations où les deux variables sont décrémentées.
- L'algorithme complète ensuite son exécution en décrémentant l'autre variable ce qui se produit max(n, m) – min(n,m) fois.
- En meilleurs cas, l'efficacité de l'algorithme est donc la suivante.

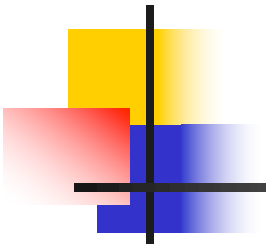
$$C_{BEST}(n,m) = \min(n,m) + (\max(n,m) - \min(n,m)) = \max(n,m) \in \Theta(n+m)$$

- Conclusion: en pire et en meilleurs cas, l'algorithme ImprimeÉditions s'exécute en temps $C(n,m) \in \Theta(n+m)$



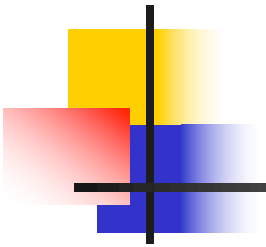
Applications de la distance d'édition

- La distance d'édition est utilisée dans l'agrégation de données provenant de différentes sources.
  - C'est souvent le cas lorsqu'on demande à des internautes de remplir un formulaire en ligne.
- L'algorithme est aussi utilisé pour comparer deux séquences d'ADN
  - Exemple:
    - AGTCAGTCAGTC
    - AGCCAGTCAAGTC
  - Dans cet exemple, la deuxième séquence a subi une mutation: le nucléotide T a été muté en nucléotide C et le nucléotide A a été inséré dans la séquence.
- La distance d'édition est utilisée dans les outils de vérification orthographique afin de proposer un mot du dictionnaire qui s'apparente le plus au mot écrit par l'utilisateur.



Applications de la distance d'édition

- La distance d'édition est utilisée pour comparer deux versions d'un code de programmation.
- Dans les systèmes de sauvegarde, la distance d'édition permet de ne sauvegarder que les modifications apportées à un document plutôt que d'enregistrer au complet la nouvelle version du document.
- Mise à jour de logiciel: les systèmes d'exploitation utilisent la distance d'édition pour calculer les différences entre deux versions d'un même programme. Lors de la mise à jour, les utilisateurs ne téléchargent pas un nouveau programme, mais seulement les différences entre l'ancienne version et la nouvelle du programme. Le logiciel de mise à jour s'occupe d'appliquer les modifications à l'ancienne version du programme afin d'obtenir la nouvelle version.



Lecture (Levitin)

- Chapitre 8
  - 8.1 Three Basic Examples
  - 8.2 The Knapsack Problem and Memory Functions