SOLUTIONS SÉRIE 7 (Chapitre 8)

Question # 1

Lequel des algorithmes suivants est le plus efficace pour calculer le coefficient binomial $C(n,k) = \binom{n}{k}$ *?*

A) Utiliser la formule

$$C(n,k) = \frac{n!}{k!(n-k)!}$$

B) Utiliser la formule

$$C(n,k) = \frac{n \times (n-1) \times \ldots \times (n-k+1)}{k!}$$

C) Appliquer récursivement la formule

$$C(n,k) = C(n-1,k-1) + C(n-1,k) \text{ for } n \geq k > 0$$

$$C(n,0) = C(n,n) = 1$$

D) L'algorithme de programmation dynamique. Voir Algorithme 1.

E) Appliquer récursivement la formule

$$C(n,k) = \begin{cases} 1 & \text{si } k = 0 \\ \frac{n}{k} C(n-1,k-1) & \text{sinon} \end{cases}$$

Algorithme 1 : Binomial(n,k)

```
1 pour i = 0..n faire
2     pour j = 0.. min(i,k) faire
3         si j = 0 ou j = i alors
4             C[i,j] ← 1
5         sinon
6             C[i,j] ← C[i-1,j-1] + C[i-1,j]
7 retourner C[n,k]
```

Solution [1] ***:***

A) $\Theta(n)$

B) $\Theta(k)$

C) $\Theta(C(n,k))$

D) $\Theta(nk)$

E) $\Theta(k)$

L'algorithme B) est donc le meilleur choix.

Question # 2

Considérons deux équipes A *et* B *qui jouent une série de parties jusqu'à ce qu'une équipe ait gagné* n *parties. En supposant que la probabilité que l'équipe* A *gagne une partie est la même pour toutes les parties et est égale à* p *et que la probabilité que l'équipe* A *perde une partie est* $q = 1 - p$ *(donc il n'y a jamais de partie nulle). Soit* $P(i,j)$*, la probabilité que* A *gagne la série si* A *a besoin de* i *victoires pour gagner alors que* B *a besoin de* j *victoires pour gagner.*

1. L'analyse est faite en fonction de la valeur de l'entrée et non de la taille de l'entrée.

A) *Construisez une relation de récurrence pour $P(i,j)$ qui peut être utilisée par un algorithme de programmation dynamique.*
Solution :
Soit $P(i,j)$, la probabilité que A gagne la série si A a besoin de i victoires supplémentaires alors que B en a besoin de j. Si A remporte la prochaine partie (ce qui arrive avec une probabilité p), A aura alors besoin de $i-1$ victoires supplémentaires tandis que B en a encore besoin de j. Par contre, si A perd la prochaine partie (ce qui arrive avec une probabilité $q=1-p$), A aura alors besoin d'encore i victoires tandis que B n'aura besoin que de $j-1$ victoires. On obtient alors la récurrence suivante pour le calcul de $P(i,j)$

$$P(i,j) = pP(i-1,j) + qP(i,j-1) \text{ pour } i,j > 0$$

On peut facilement extraire les cas de bases[2] suivants à partir de la définition de $P(i,j)$

$$P(0,j) = 1 \text{ pour } j > 0 \text{ et } P(i,0) = 0 \text{ pour } i > 0$$

B) *Écrivez un algorithme pour résoudre ce problème selon l'approche de programmation dynamique et déterminez son efficacité en temps et en espace mémoire.*
Solution : *Voir Algorithme 2 pour la solution. L'efficacité, tant en temps qu'en espace mémoire, est de $\theta(n^2)$ car chaque entrée du tableau de $n+1$ par $n+1$ (à l'exception de l'entrée $P[0,0]$ qui n'est pas calculée) requiert un temps constant pour être calculée.*

Le tableau 1 contient un exemple d'application de l'algorithme de programmation dynamique lorsque $p = 0.4$. La probabilité que A remporte une série de 7 parties est $P[4,4] = 0.29$.

$i \backslash j$	0	1	2	3	4
0	ND	1	1	1	1
1	0	0.40	0.64	0.78	0.87
2	0	0.16	0.35	0.52	0.66
3	0	0.06	0.18	0.32	0.46
4	0	0.03	0.09	0.18	0.29

TABLEAU 1 – Série de 7 parties

Algorithme 2 : WorldSeries(n, p)

```
q ← 1 − p
pour j = 1..n faire
    P[0, j] ← 1
pour i = 1..n faire
    P[i, 0] ← 0
    pour j = 1..n faire
        P[i, j] ← p × P[i − 1, j] + q × P[i, j − 1]
retourner P[n, n]
```

2. Le cas où $P(0,0)$ n'est pas possible.

Question # 3

De façon générale, comment peut-on utiliser le tableau construit par l'algorithme de programmation dynamique pour savoir s'il y a plus d'une solution optimale pour le problème du « sac à dos » ?

Vous pouvez vous aider de l'instance du problème du « sac-à-dos » donné par le tableau 2 et de la trace de l'algorithme donnée par le tableau 3 lorsque la capacité du sac est de $W = 6$.

item	poids	valeur
1	3	25
2	2	20
3	1	15
4	4	40
5	5	50

TABLEAU 2 – Instance du problème du « sac à dos »

$i \backslash j$	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	25	25	25	25
2	0	0	20	25	25	45	45
3	0	15	20	35	40	45	60
4	0	15	20	35	40	55	60
5	0	15	20	35	40	55	65

TABLEAU 3 – Solutions pour une instance du problème du « sac à dos »

Solution :

Une instance du problème du « sac à dos » a une solution unique si et seulement si l'algorithme pour lire une solution optimale à partir du tableau de programmation dynamique (en retraçant le chemin permettant de calculer $V[n, W]$*) ne rencontre pas d'égalité entre* $V[i-1, j]$ *et* $v_i + V[i-1, j-w_i]$.

Question # 4

Élaborez un algorithme de programmation dynamique pour le problème « rendre la monnaie » (étant donné un montant n *et une quantité suffisante de pièces pour chacune des valeurs* $D[1], D[2], \ldots, D[m]$*, il faut trouver le plus petit nombre de pièces dont la somme est* n *ou indiquer que le problème n'a pas de solution). L'algorithme doit d'abord calculer le nombre de pièces de la solution optimale, puis retourner le nombre de pièces de chaque type utilisé dans cette solution.*

Solution 1 :

Soit $C[i, j]$*, le plus petit nombre de pièces requis pour remettre le montant* j *(*$0 \leq j \leq n$*) en utilisant seulement des pièces des* i *premières dénominations (*$1 \leq i \leq m$*). On utilise* $C[i, j] = \infty$ *pour indiquer qu'il n'y a pas de solution pour cette instance. La récurrence suivante permet de calculer* $C[i, j]$

$$
\begin{aligned}
C[i, 0] &= 0 \text{ pour } 0 \leq i \leq m \\
C[0, j] &= \infty \text{ pour } 1 \leq j \leq n \\
C[i, j] &= C[i-1, j] \text{ si } D[i] > j \\
C[i, j] &= \min(C[i-1, j], 1 + C[i, j - D[i]]) \text{ sinon}
\end{aligned}
$$

Les cas de bases ont la signification suivante :

— $C[i, 0] = 0$ *car pour rendre un montant de* 0 *en utilisant les* i *premières pièces, on a besoin de* 0 *pièce ;*

— $C[0, j] = \infty$ *car il est impossible de rendre un montant non nul en utilisant* 0 *pièce ;*

La récurrence vient du fait que pour rendre un montant j *en utilisant les* i *premières pièces, il faut :*

— *soit rendre* j *en utilisant seulement les* $i - 1$ *premières pièces, auquel cas le montant à rembourser est toujours de* j*, i.e., le cas* $C[i-1, j]$ *;*

— *soit utiliser* 1 *pièce de type* i*, auquel cas le montant à rembourser est maintenant de* $j - D[i]$*, i.e., le cas* $1 + C[i, j - D[i]]$.

Algorithme 3 rempli le tableau de programmation dynamique qui contient le nombre de pièces minimal dans $C[m, n]$, puis remonte le tableau pour déterminer le nombre de pièces de chaque type utilisé dans la solution optimale.

Solution 2 :

Soit $P[i]$, le plus petit nombre de pièces requis pour remettre le montant i en utilisant les pièces D. On utilise $P[i] = \infty$ pour indiquer qu'il n'y a pas de solution à cette instance. La récurrence suivante permet de calculer $P[i]$.

$$P[i] = \begin{cases} 0 & \textit{si i = 0} \\ \min\{1 + P[i-d] \mid d \in D \wedge d \leq i\} \cup \{\infty\} & \textit{sinon} \end{cases}$$

La récurrence vient du fait que pour rendre un montant i, il faut choisir, parmi les pièces qu'il est possible d'utiliser, celle qui minimise le nombre de pièces qu'il restera à rendre. S'il n'y a aucune pièce suffisamment petite pour rendre le montant i, le minimum se fera sur un ensemble contenant uniquement l'infini, indiquant qu'il n'y a pas de solution pour ce montant i.

Algorithme 4 rempli le tableau de programmation dynamique qui contient le nombre de pièces minimal dans $P[n]$, puis remonte le tableau pour déterminer le nombre de pièces de chaque type utilisé dans la solution optimale.

Algorithme 3 : MinimumChange1($D[1, \ldots m], n$)

```
1  pour i = 0..m faire C[i, 0] ← 0 ;
2  pour j = 1..n faire C[0, j] ← ∞ ;
3  pour i = 1..m faire
4      pour j = 1..n faire
5          si D[i] ≤ j alors
6              C[i, j] ← min(1 + C[i, j − D[i]], C[i − 1, j])
7          sinon
8              C[i, j] ← C[i − 1, j]
9  si C[m, n] = ∞ alors
10     retourner «Aucune solution»
11 pour k = 1..m faire L[k] ← 0 ;
12 i ← m
13 j ← n
14 tant que i > 0 ∧ j > 0 faire
15     si C[i, j] = C[i − 1, j] alors
16         i ← i − 1
17     sinon
18         L[i] ← L[i] + 1
19         j ← j − D[i]
20 retourner L
```

Algorithme 4 : MinimumChange2($D[1, \ldots m], n$)

```
1  P[0] ← 0
2  pour i = 1..n faire
3  |   P[i] ← ∞
4  |   pour d ∈ D faire
5  |   |   si d ≤ i alors P[i] ← min(P[i], 1 + P[i − d]) ;
6  si P[n] = ∞ alors
7  |   retourner «Aucune solution»
8  pour j = 1..m faire L[j] ← 0 ;
9  i ← n
10 tant que i > 0 faire
11 |   pour j = 1..m faire
12 |   |   d ← D[j]
13 |   |   si P[i − d] = P[i] − 1 alors
14 |   |   |   L[j] ← L[j] + 1
15 |   |   |   i ← i − d
16 |   |   |   break
17 retourner L
```

Question # 5

Vous avez une image, que l'on peut encoder comme une matrice de pixels $A[1..m, 1..n]$, et vous désirez la compresser à l'aide du recadrage intelligent. Cette technique consiste à retirer un unique pixel sur chacune des m lignes de la matrice A. Pour éviter au maximum la corruption de l'image, les pixels supprimés sur deux lignes consécutives doivent être dans la même colonne ou dans des colonnes adjacentes de A. La suite de pixels retirés est appelée «couture».

Supposons que l'on vous fournit une matrice $P[1..m, 1..n]$ contenant les perturbations entraînées par le retrait de chaque pixel. Alors, vous avez que $P[i, j]$ est la perturbation encourrue si on retire le pixel $A[i, j]$. La perturbation totale est la somme des perturbations de la couture.

A) Construisez une relation de récurrence pour $T[i, j]$, la valeur minimale de la perturbation encourrue en retirant le pixel $A[i, j]$.

Solution :

La première ligne n'est pas influencée par les lignes précédentes, c'est pourquoi $T[1, j]$ représente les cas de base. Ensuite, il faut faire attention aux bornes du tableau. La relation de récurrence est la suivante.

$$
\begin{aligned}
T[1, j] &= P[1, j] && \forall j \in \{1, \ldots, n\} \\
T[i, 1] &= \min(T[i-1, 1], T[i-1, 2]) + P[i, 1] && \forall i \in \{1, \ldots, m\} \\
T[i, n] &= \min(T[i-1, n-1], T[i-1, n]) + P[i, n] && \forall i \in \{1, \ldots, m\} \\
T[i, j] &= \min(T[i-1, j-1], T[i-1, j], T[i-1, j+1]) + P[i, j] && \textit{autrement}
\end{aligned}
$$

B) Élaborez un algorithme pour déterminer la valeur de la perturbation totale minimale pour une image donnée $A[1..m, 1..n]$.

Solution :

Voir Algorithme 5 pour la solution.

C) Élaborez un algorithme qui détermine quels pixels doivent être retirés à partir de la solution retournée en A).

Solution :

Voir Algorithme 6 pour la solution.

Algorithme 5 : MinimalPerturbationValue($A[1 \ldots m, 1 \ldots n], P[1 \ldots m, 1 \ldots n]$)

```
pour j = 1..n faire
    T[1, j] ← P[1, j]
pour i = 2..m faire
    T[i, 1] ← min(T[i − 1, 1], T[i − 1, 2]) + P[i, 1]
    pour j = 2..n − 1 faire
        T[i, j] ← min(T[i − 1, j − 1], T[i − 1, j], T[i − 1, j + 1]) + P[i, j]
    T[i, n] ← min(T[i − 1, n − 1], T[i − 1, n]) + P[i, n]
retourner T
```

Algorithme 6 : Couture($T[1 \dots m, 1 \dots n], P[1 \dots m, 1 \dots n]$)

```
pour i = 1..m faire
    R[i] ← 0
k ← 0
min ← ∞
pour j = 1..n faire
    si T[m, j] < min alors
        k ← j
        min ← T[m, j]
R[m] ← k
pour i = m − 1..1 faire
    si k > 1 ∧ T[i, k − 1] + P[i + 1, k] = T[i + 1, k] alors
        k ← k − 1
    sinon si k < n ∧ T[i, k + 1] + P[i + 1, k] = T[i + 1, k] alors
        k ← k + 1
    R[i] ← k
retourner R
```

Question # 6

Une planche de longueur L doit être coupée aux positions $p = [p_1, p_2, \dots, p_n]$. La planche doit passer dans une machine et ressort coupée à un seul endroit. Le temps pour couper une planche est proportionel à sa longueur.

A) Trouvez la relation de réccurence pour obtenir le temps minimum pour couper la planche.

On doit ajouter 0 au début du vecteur p et L à la fin. Notez que la longueur du vecteur est maintenant $n+2$. $T[i, j]$ est le temps pour couper la planche entre les positions i et j. Comme on cherche le temps pour couper la planche au complet, on cherche la valeur de $T[0, n]$.

Cas de base

Si on veut couper entre deux positions consécutives, il n'y a pas de coupe à faire. Le temps est donc 0. Donc $T[i, i + 1] = 0 \; \forall \; i \; \in \{0, 1, \dots, n\}$

Cas général

On cherche la coupe minimale entre i et j. On doit passer la planche dans la machine pour la couper au moins une fois. On aura donc un coût de $p_j - p_i$.

On doit maintenant ajouter le coût des coupes des deux morceaux qu'on vient de créer.

Si on coupe à la position $i + 1$, on va ajouter le coût du morceau i à $i + 1$ et du morceau $i + 1$ à j, donc $T[i, i + 1]$ et $T[i + 1, j]$.

Si on coupe à la position $i + 2$, on va ajouter le coût du morceau de i à $i + 2$ et du morceau de $i + 2$ à j, donc $T[i, i + 2]$ et $T[i + 2, j]$.

Si on coupe à la position $i + 3$, on va ajouter le coût du morceau de i à $i + 3$ et du morceau de $i + 3$ à j, donc $T[i, i + 3]$ et $T[i + 3, j]$.

Si on coupe à la position $i + k$, on va ajouter le coût du morceau de i à $i + k$ et du morceau de $i + k$ à j, donc $T[i, i + k]$ et $T[i + k, j]$.

On cherche la coupe mininmale, donc on cherche le k qui minimise la coupe.

Notre coût va donc être

$$T[i,j] = p_j - p_i + \min_{i<k<j}(T[i,k] + T[k,j]) \qquad \forall\, i,j \mid j > i+1$$

Au final, nous obtenons

$$T[i,j] = \begin{cases} 0 & \textit{si } i+1 = j \\ p_j - p_i + \min_{i<k<j}(T[i,k] + T[k,j]) & \textit{si } i+1 < j \end{cases}$$

B) *Écrivez un algorithme pour résoudre ce problème selon l'approche de programmation dynamique.*

Algorithme 7 : MinimaleCoupe($[p_1, \ldots, p_n]$)

```
1  p ← [0, p_1, p_2, ..., p_n, L];
2  pour i = 0..n faire
3      T[i, i+1] ← 0
4  pour m ← 2..n+1 faire
5      pour i = 0..n−m+1 faire
6          j ← i + m;
7          T[i,j] ← p_j − p_i + T[i,i+1] + T[i+1,j];
8          pour k = i+2..j−1 faire
9              T[i,j] ← min(T[i,j], p_j − p_i + T[i,k] + T[k,j]);
10 retourner T
```

C) *Élaborez un algorithme pour énumérer l'ordre des coupes.*

Algorithme 8 : TrouverCoupe($T[0..,n+1][0..,n+1], [p_1 \ldots p_n]$)

```
1  p ← [0, p_1, p_2, ..., p_n, L];
2  Soit C[1..n] un vecteur;
3  a ← 1;
4  V ← pile vide;
5  push(V, (0, n+1));
6  tant que V ≠ ∅ faire
7      (i,j) ← pop(V);
8      si i+1 < j alors
9          pour k = i+1..j−1 faire
10             si T[i,j] = p_j − p_i + T[i,k] + T[k,j] alors
11                 C[a] ← k;
12                 a ← a + 1;
13                 push(V, (i,k));
14                 push(V, (k,j));
15                 break;
16 retourner C;
```