Devoir n°1

Exercice : dynamique unidimensionnelle

On considère la suite de Fibonacci (a_n) défnie par $a_0 = a_1 = 1$ et la relation de récurrence

$$\forall n \geqslant 2\,,\ a_n = a_{n-1} + a_{n-2}$$

1. Calculer a_n pour $n = 0, \ldots, 10$.
2. Programmation récursive.
   (a) La fonction récursive est naturelle à écrire :

```
def a(n):
    if n == 0 or n == 1:
        return 1
    return a(n-1) + a(n-2)
```

   Si $n \in \mathbb{N}$, on note $U(n)$ le nombre d'appels nécessaires pour calculer a_n.

   On a $U(0) = 1$ et $U(1) = 1$ puisque l'on obtient la valeur de a_0 et a_1 en n'effectuant qu'un seul appel initial, la valeur étant renvoyée dès le premier test d'arrêt.

   (b) Montrer que si $n \geqslant 2$, on a
   $$U_n = 1 + U_{n-1} + U_{n-2}$$
   On pose $V_n = U_n + 1$. Vérifier que $V_n = V_{n-1} + V_{n-2}$.
   Remarques ? En déduire que
   $$\forall n \in \mathbb{N}\,,\ U(n) = \frac{2}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right] - 1$$
   Montrer alors que
   $$U(n) \sim \frac{2}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}$$
   Quelle est la nature de la complexité en appels récursifs ? Conclusion ?
3. Programmation dynamique avec mémorisation partielle.

   Pour calculer a_n, nous avons seulement besoin de connaître les deux termes précédent. Compléter (et vérifier le fonctionnement) la fonction suivante qui programme cette idée :

```python
def a_dyn(n):
    x, y = 1, 1   # représentent a_0 et a_1
    for i in range(?):
        x, y = y, x + y  # affectation en parallèle
    return ?
```

Quelle est la complexité d'une telle fonction ?
À votre avis, peut-on l'amméliorer ?

4. Programmation dynamique avec mémorisation totale.

On considère la suite de Catalan définie par $C_0 = 1$ et la relation de récurrence

$$\forall n \in \mathbb{N}^*, C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$$

(a) Calculer C_1, C_2, C_3 et C_4.

(b) Sachant que $C_n \sim \frac{4^n}{\sqrt{\pi} n^{3/2}}$, quelle sera la complexité en appels récursifs d'une programmation récursive ?

(c) On propose alors une programmation dynamique avec mémorisation. Compléter alors la fonction suivante qui programme le calcul de C_n de façon dynamique :

```python
def C_dyn(n):
    C = [0]*(n+1)
    C[0] = 1
    for i in range(1, n+1):
        s = 0
        for k in range(?):
            s += C[?]*C[?]
        C[?] = ?
    return ?
```

Déterminer la complexité de cette fonction et conclure.

Exercice : dynamique bidimensionnelle

Un élève de CPGE souhaite gagner de l'argent de poche. Il peut y consacrer un maximum de H heures par semaine. Il a repéré n emplois. Mais le salaire offert pour chaque emplois n'est pas proportionnel au nombre d'heure travaillées. Par exemple, la tableau suivant agrège les salaires $s(h, j)$ de l'emploi j si on travaille h heures ($T = 4$, $n = 4$) :

heures de travail : h \ emploi : j	1	2	3	4
0	0	0	0	0
1	26	24	16	20
2	38	34	32	34
3	47	42	48	48
4	50	49	64	52

1. La méthode gloutonne est directe (et donc pas très intéressante à programmer ici), elle consiste sur l'exemple à choisir les 4 heures qui payent le plus, soit l'emploi 3. Ce choix est-il pertinent ?

2. On note $P(h,j)$ le salaire maximum perçu possible en travaillant, de façon donc optimale, h heures parmi les emplois de 1 à i.
On pose aussi $P(0,j)=0$ si $1 \leqslant j \leqslant n$.

   (a) Quelle valeur de $P(h,j)$ doit-on calculer pour résoudre le problème ?

   (b) Que vaut $P(h,1)$ pour $0 \leqslant h \leqslant H$?

   (c) Soit $2 \leqslant j \leqslant n$. Montrer que si on travaille h heures parmi les emplois de 1 à j de façon optimisée, alors en notant k le nombre d'heures travaillées parmi les emplois parmi 1 à $j-1$, le salaire de ces k heures parmi les emplois de 1 à $j-1$ est aussi optimisé.

   (d) En déduire la formule si $2 \leqslant j \leqslant n$, $0 \leqslant h \leqslant H$,

$$P(h,j) = \max_{k=0\cdots h} P(k,j-1) + s(h-k,j)$$

   (e) Oubliant les problèmes dus à la récursivité, on écrit une fonction récursive permettant d'appliquer la formule de récursivité ci-dessus.

      i. Compléter et commenter la fonction suivante, pour quelle fonctionne avec s défini par

```
s = [[0,0,0,0],[26,24,16,20], [38,34,32,34],
[47,42,48,48], [50,49,64,52]]
```

      c'est à dire que les emplois sont numérotés 0 à 3.

```
def p_rec(s, h, j):
    if j == 0:
        return ?
    m = s[0][j]  # k=0
    for k in range(1, h+1):
        p = p_rec(s, ?, j-1) + s[?][?]
        if p > m :
            m = ?
    return ?
```

      Pour l'exemple, on exécute donc

```
p_rec(s, 4, 3)
```

      pour obtenir le résultat.

      ii. Pour évaluer la complexité de cette fonction récursive, on note $C(h,j+1)$ le nombre d'appels à la fonction récursive pour calculer $P(h,j)$. Nous avons

$$C(h,1)=1 \qquad C(0,j)=1$$

      Montrer que si $j \geqslant 2$,

$$C(h,j) = 1 + \sum_{k=1}^{h} C(k,j-1) = \sum_{k=0}^{h} C(k,j-1)$$

      En déduire que :

$$C(T,2) = T+1 \qquad C(T,3) = \frac{(T+1)(T+2)}{2} \qquad C(T,4) = \frac{(T+1)(T+2)(T+3)}{6}$$

On conjecture alors la formule (si $n = 1$, produit vide égal à 1)

$$C(T,n) = \frac{(T+1)\cdots(T+n-1)}{(n-1)!} = \binom{T+n-1}{T} = \binom{T+n-1}{n-1}$$

et une preuve par récurrence sur n conduit, pour prouver l'héridité, à montrer la formule

$$\sum_{k=0}^{T} \binom{k+n-1}{n-1} = \binom{T+n}{n}$$

iii. Montrer que la formule ci-dessus est vraie, par récurrence sur T.

(f) La méthode par récursitié n'est pas efficace. Nous procédons alors par programmation dynamique, à l'aide d'un tableau bi-dimensionnel. Compléter et commenter la fonction suivante, qui renvoie le salaire maximisé :

```
def p_dyn(s):
    H = len(s) - 1
    n = len(s[0])
    p = [[0]*n for _ in range(H+1)]
    for h in range(H+1):
        p[h][0] = ?
    for j in range(1, n):
        for h in range(H+1):
            m = -1
            for k in range(h+1):
                pp = p[k][j-1] + s[h-k][j]
                if pp > m :
                    m = ?
            p[h][j] = ?
    return ?
```

(g) Appliquer la méthode à l'exemple. On obtiendra le tableau p suivant qu'il faudra établir complètement :

heures de travail : h \ emploi : j	1	2	3	4
0	0	0	0	0
1	26	26	26	26
2	?	?	?	?
3	?	?	?	?
4	50	?	?	86

(h) Déterminer la complexité de la fonction p_dyn. On considèrera que le corps de la boucle en k est à temps constant.

(i) Peut-on à l'aide du tableau p déterminer les emplois avec leurs nombres d'heures qui permettent d'optimiser le salaire ?
Que faut-il ajouter dans la fonction p_dyn pour pouvoir effectuer un backtracing ?
Compléter la fonction pour qu'elle renvoie la liste des emplois avec le nombre d'heures pour optimiser le salaire.
Appliquer la méthode à l'exemple.

Exercice : dynamique unidimensionnelle

On considère la suite de Fibonacci (a_n) défnie par $a_0 = a_1 = 1$ et la relation de récurrence

$$\forall n \geqslant 2\,,\ a_n = a_{n-1} + a_{n-2}$$

1. On calcule a_n pour $n = 0, \dots, 10$:

n	0	1	2	3	4	5	6	7	8	9	10
a_n	1	1	2	3	5	8	13	21	34	55	89

2. Programmation récursive.

   (a) La fonction récursive est naturelle à écrire :

```
def a(n):
    if n == 0 or n == 1:
        return 1
    return a(n-1) + a(n-2)
```

   Si $n \in \mathbb{N}$, on note $U(n)$ le nombre d'appels nécessaire pour calculer a_n.

   On a $U(0) = 1$ et $U(1) = 1$ puisque l'on obtient la valeur de a_0 et a_1 en n'effectuant qu'un seul appel, la valeur étant renvoyée dès le premier test d'arrêt.

   (b) Soit $n \geqslant 2$, pour calculer a_n, on a besoin de a_{n-1} donc U_{n-1} appels et a_{n-2} soit aussi en plus U_{n-2} appels et ainsi

$$U_n = 1 + U_{n-1} + U_{n-2}$$

   en comptant l'appel initial. Alors

$$V_n - 1 = 1 + V_{n-1} - 1 + V_{n-2} - 1$$

   soit encore $V_n = V_{n-1} + V_{n-2}$.
   C'est la même récurrence que la suite (a_n) ! Nous avons une suite récurrente linéaire à coeffients constant d'ordre 2, d'équation caractéristique $r^2 - r - 1 = 0$, de discriminant $\Delta = 1 + 4 = 5$, de racines $\frac{1\pm\sqrt{5}}{2}$, et ainsi il existe α et β réels de sorte que

$$\forall n \in \mathbb{N}\,,\ V_n = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^n + \beta\left(\frac{1-\sqrt{5}}{2}\right)^n$$

   Les conditions initiales permettent d'accéder à α et β :

$$\begin{cases} V_0 = 2 & = \alpha + \beta \\ U_1 = 2 & = \alpha\left(\frac{1+\sqrt{5}}{2}\right) + \beta\left(\frac{1-\sqrt{5}}{2}\right) = \frac{\alpha+\beta}{2} + \frac{\sqrt{5}}{2}(\alpha - \beta) \end{cases}$$

   et ainsi

$$\alpha + \beta = 2 \qquad \alpha - \beta = \frac{2}{\sqrt{5}}$$

puis par demi-somme et demi-différence

$$\alpha = \frac{\sqrt{5}+1}{\sqrt{5}} \qquad \beta = \frac{\sqrt{5}-1}{\sqrt{5}}$$

On en déduit

$$\forall n \in \mathbb{N}\,,\, V_n = \frac{2}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right]$$

puis

$$\forall n \in \mathbb{N}\,,\, U_n = \frac{2}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right] - 1$$

Or $4 < 5 < 9$, $2 < \sqrt{5} < 3$, et ainsi $-2 < 1-\sqrt{5} < -1$ d'où $-1 < \frac{1-\sqrt{5}}{2} < 0$, et $\frac{1+\sqrt{5}}{2} > \frac{1+2}{2} > 1$, et ainsi $\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}$ tend vers 0 et $\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}$ vers $+\infty$, on a

$$U(n) \sim \frac{2}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}$$

Nous avons une complexité en appels récursifs exponentielle. La programmation récursive n'est pas efficace.

3. Programmation dynamique avec mémorisation partielle.

Pour calculer a_n, nous avons seulement besoin de connaître les deux termes précédents. La fonction suivante programme cette idée :

```
def a_dyn(n):
    x, y = 1, 1   # représentent a_0 et a_1
    for i in range(n-1):
        x, y = y, x + y  # affectation en parallèle
    return y
```

Nous avons une complexité en $\mathcal{O}(n)$, le corps de la boucle étant en temps constant et la boucle étant de taille n.

À votre avis, peut-on l'amméliorer ? On pourrait penser que non, mais en fait oui, par une méthode matricielle et calcul de puissances dichotomique. C'est surprenant et efficace, en complexité $\ln(n)$.

4. Programmation dynamique avec mémorisation totale.

On considère la suite de Catalan définie par $C_0 = 1$ et la relation de récurrence

$$\forall n \in \mathbb{N}^*\,,\, C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$$

(a) On calcule C_1, C_2, C_3 et C_4 :

n	0	1	2	3	4
C_n	1	1	2	5	14

(b) Comme pour l'exemple précédent, la calcul du nombre d'appels récursifs sera approximativement en fait $C_n \sim \frac{4^n}{\sqrt{\pi} n^{3/2}}$, en donne donc un algorithme peu efficace.

(c) On propose alors une programmation dynamique avec mémorisation :

```python
def C_dyn(n):
    C = [0]*(n+1)
    C[0] = 1
    for i in range(1, n+1):
        s = 0
        for k in range(i):
            s += C[k]*C[i-1-k]
        C[i] = s
    return C[n]
```

Le corps de la boucle interne en k est de complexité constante, et ainsi la boucle en k est de complexité $\mathcal{O}(i)$, et ainsi la boucle en i de complexité

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2} \sim \frac{n^2}{2} = \mathcal{O}(n^2)$$

Nous avons donc une complexité globale en $\mathcal{O}(n^2)$, ce qui est évidemment plus efficace que la récursivité.

Exercice : dynamique bidimensionnelle

Un élève de CPGE souhaite gagner de l'argent de poche. Il peut y consacrer un maximum de H heures par semaine. Il a repéré n emplois. Mais le salaire offert pour chaque emplois n'est pas proportionnel au nombre d'heure travaillées. Par exemple, la tableau suivant agrège les salaires $s(h, j)$ de l'emploi j si on travaille h heures ($T = 4$, $n = 4$) :

heures de travail : h \ emploi : j	1	2	3	4
0	0	0	0	0
1	26	24	16	20
2	38	34	32	34
3	47	42	48	48
4	50	49	64	52

1. La méthode gloutonne est directe (et donc pas très intéressante à programmer ici), elle consiste sur l'exemple à choisir les 4 heures qui payent le plus, soit l'emploi 3, avec un salaire de 64 €. Mais, par exemple en travaillant 3 heures dans l'emploi 1, et une heure dans l'emploi 4, on gagne 47+20=67 €, et mieux encore, 3 heures dans l'emploi 4 avec une heure dans l'emploi 1 donne 48+26=74 €. En prenant 1 heure dans chaque emploi, on obtient 86 €.

2. On note $P(h, j)$ le salaire maximum perçu possible en travaillant, de façon donc optimale, h heures parmi les emplois de 1 à i. On pose aussi $P(0, j) = 0$ si $1 \leqslant j \leqslant n$.

   (a) Quelle valeur de $P(h, j)$ doit-on calculer pour résoudre le problème ? C'est $P(H, n)$ que l'on cherche.

   (b) On a $P(h, 1) = s(h, 1)$ pour $0 \leqslant h \leqslant H$ (uniquement l'emploi 1).

   (c) Soit $2 \leqslant j \leqslant n$. Si on travaille h heures parmi les emplois de 1 à j de façon optimisée, alors en notant k le nombre d'heures travaillées parmi les emplois parmi 1 à $j-1$, le salaire de ces k heures parmi les emplois de 1 à $j-1$ est aussi forcément optimisé car sinon, on pourrait faire mieux parmi les emplois de 1 à $j-1$, et en ajoutant l'emploi j, on obtient un choix meilleur parmi les emplois de 1 à j, ce qui est contradictoire : pour résumer, nous avons ici un raisonnement par sous-problème optimal.

   (d) Si on travaille h heures parmi les emplois de 1 à j de façon optimisée, alors en notant k le nombre d'heures travaillées parmi les emplois parmi 1 à $j-1$, le salaire de ces k heures parmi les emplois de 1 à $j-1$ est donc aussi forcément optimisé, on a $0 \leqslant k \leqslant h$; pour une valeur de k, quelqonque, le salaire

global est $P(k, j-1) + s(h-k, j)$, et doit donc choisir la valeur de k pour laquelle cette quantité est maximale, soit si $2 \leqslant j \leqslant n$, $0 \leqslant h \leqslant H$,

$$P(h, j) = \max_{k=0\cdots h} P(k, j-1) + s(h-k, j)$$

(e) Oubliant les problèmes dus à la récursivité, on écrit une fonction récursive permettant d'appliquer la formule de récursivité ci-dessus.

i. Compléter et commenter la fonction suivante, pour quelle fonctionne avec s défini par

```
s = [[0,0,0,0],[26,24,16,20], [38,34,32,34],
[47,42,48,48], [50,49,64,52]]
```

c'est à dire que les emplois sont numérotés 0 à 3.

```
def p_rec(s, h, j):
    if j == 0:
        return s[h][0]
    m = s[0][j]  # k=0
    for k in range(1, h+1): # recherche du max
        p = p_rec(s, k, j-1) + s[h-k][j]
        if p > m :
            m = p
    return m
```

Pour l'exemple, on exécute donc

```
p_rec(s, 4, 3)
```

pour obtenir le résultat (c'est 86 pour l'exemple).

ii. Pour évaluer la complexité de cette fonction récursive, on note $C(h, j+1)$ le nombre d'appels à la fonction récursive pour calculer $P(h, j)$. Nous avons

$$C(h, 1) = 1 \qquad C(0, j) = 1$$

Si $j \geqslant 2$, 1 appel, puis pour chaque k entre 1 et h, $C(k, j-1)$ sous-appels d'où

$$C(h, j) = 1 + \sum_{k=1}^{h} C(k, j-1) = \sum_{k=0}^{h} C(k, j-1)$$

Alors

$$C(T, 2) = \sum_{k=0}^{T} 1 = T + 1$$

$$C(T, 3) = \sum_{k=0}^{T} k + 1 = 1 + 2 + \cdots + (T+1) = \frac{(T+1)(T+2)}{2}$$

$$\begin{aligned} C(T, 4) &= \sum_{k=0}^{T} \frac{(k+1)(k+2)}{2} = \frac{1}{2} \sum_{k=0}^{T} (k+1)(k+1+1) \\ &= \frac{1}{2} \sum_{k=0}^{T} (k+1)^2 + \frac{1}{2} \sum_{k=0}^{T} (k+1) \\ &= \frac{(T+1)(T+2)(2T+3)}{12} + \frac{(T+1)(T+2)}{4} = \frac{(T+1)(T+2)}{12}(2T+3+3) \\ &= \frac{(T+1)(T+2)(T+3)}{6} \end{aligned}$$

On a donc

$$C(T,2) = T+1 \qquad C(T,3) = \frac{(T+2)(T+2)}{2} \qquad C(T,4) = \frac{(T+1)(T+2)(T+3)}{6}$$

On conjecture alors la formule (si $n = 1$, produit vide égal à 1)

$$C(T,n) = \frac{(T+1)\cdots(T+n-1)}{(n-1)!} = \binom{T+n-1}{T} = \binom{T+n-1}{n-1}$$

et une preuve par récurrence sur n conduit, pour prouver l'héridité, à montrer la formule

$$\sum_{k=0}^{T} \binom{k+n-1}{n-1} = \binom{T+n}{n}$$

iii. Montrons que la formule ci-dessus est vraie, par récurrence sur T.
Pour $T = 0$, nous avons pour tout $n \geqslant 1$,

$$\sum_{k=0}^{0} \binom{k+n-1}{n-1} = \binom{n-1}{n-1} = 1 = \binom{n}{n} = \binom{T+n}{n}$$

Soit T. On suppose que si $n \geqslant 1$, on a

$$\sum_{k=0}^{T} \binom{k+n-1}{n-1} = \binom{T+n}{n}$$

Alors pour $T+1$, on a

$$\begin{aligned}\sum_{k=0}^{T+1} \binom{k+n-1}{n-1} &= \sum_{k=0}^{T} \binom{k+n-1}{n-1} + \binom{T+n}{n-1} \\ &= \binom{T+n}{n} + \binom{T+n}{n-1} = \binom{T+n+1}{n} = \binom{(T+1)+n}{n}\end{aligned}$$

ce qui achève la récurrurence sur T.

(f) La méthode par récursitié n'est pas efficace. Nous procédons alors par programmation dynamique, à l'aide d'un tableau bi-dimensionnel.

```
def p_dyn(s):
    H = len(s) - 1
    n = len(s[0])
    p = [[0]*n for _ in range(H+1)]
    for h in range(H+1):
        p[h][0] = s[h][0]
    for j in range(1, n):
        for h in range(H+1):
            m = -1
            for k in range(h+1):
                pp = p[k][j-1] + s[h-k][j]
                if pp > m :
                    m = pp
            p[h][j] = m
    return p[H][n-1]
```

(g) Appliquer la méthode à l'exemple. On obtiendra le tableau p suivant (merci python) :

heures de travail : h \ emploi : j	1	2	3	4
0	0	0	0	0
1	26	26	26	26
2	38	50	50	50
3	47	62	66	70
4	50	72	82	86

(h) On considère que le corps de la boucle en k est à temps constant. La boucle en k est donc en $\mathcal{O}(h)$. La boucle en h est alors de complexité

$$1 + 2 + \cdots + H = \frac{H(H+1)}{2} = \mathcal{O}(H^2)$$

indépendant de j, qui est donc la complexité du corps de boucle en j, et la boucle en j étant de taille n, nous obtenons une complexité totale en $\mathcal{O}(nH^2)$.

(i) Peut-on à l'aide du tableau p déterminer les emplois avec leurs nombres d'heures qui permettent d'optimiser le salaire ? Non.
Pour pouvoir effectuer un backtracing, il ajouter dans la fonction p_dyn la mémorisation dans un tableau de la valeur de k qui a permi l'obtention du maximum. Ensuite, on peut effectivement faire un retour en arrière pour déterminer une répartition optimale :

```
def p_dyn_bt(s):
    H = len(s) - 1
    n = len(s[0])
    p = [[0]*n for _ in range(H+1)]
    pk = [[0]*n for _ in range(H+1)] # tableau des valeurs de k
    for h in range(H+1):
        p[h][0] = s[h][0]
        pk[h][0] = h
    for j in range(1, n):
        for h in range(H+1):
            m = -1
            for k in range(h+1):
                pp = p[k][j-1] + s[h-k][j]
                if pp > m :
                    m = pp     # mise à jour
                    kk = h - k     # mise à jour
            p[h][j] = m
            pk[h][j] = kk      # la valeur de k
    # backtracing
    h, j = H, n-1      # position initiale
    liste = []
    while j >= 0:      # emploi j (décalé de 1)
        k = pk[h][j]  # nombre d'heures
        liste = ['emploi numero '+str(j+1), 'heures :'
            + str(k)] + liste
        h = h - k
        j -= 1
    return p, pk, liste, p[H][n-1]
```

On donne le tableau des valeurs de k pour l'exemple :

heures de travail : h \ emploi : j	1	2	3	4
0	0	0	0	0
1	1	0	0	0
2	2	1	0	0
3	3	1	1	1
4	4	2	2	1

Suivons pas à pas le backtracing :
• étape 1 : travail 4 : 1 heures

heures de travail : h \ emploi : j	1	2	3	4
0	0	0	0	0
1	1	0	0	0
2	2	1	0	0
3	3	1	1	1
4	4	2	2	[1]

• étape 2 : on regarde ensuite quelle case ? il reste 4-1=3 heures, pour les emplois de 1 à 3 :

heures de travail : h \ emploi : j	1	2	3	4
0	0	0	0	0
1	1	0	0	0
2	2	1	0	0
3	3	1	[1]	1
4	4	2	2	1

Nous avons pour l'emploi 3 : 1 heure ;
• étape 3 : on regarde ensuite quelle case ? il reste 4-1-1=2 heures, pour les emplois de 1 à 2 :

heures de travail : h \ emploi : j	1	2	3	4
0	0	0	0	0
1	1	0	0	0
2	2	[1]	0	0
3	3	1	1	1
4	4	2	2	1

soit 1 heure pour l'emploi 2 et ensuite il reste 1 heure pour l'emploi 1 :

heures de travail : h \ emploi : j	1	2	3	4
0	0	0	0	0
1	[1]	1	1	1
2	2	1	2	2
3	3	1	1	1
4	4	2	2	1

soit 1 heure pour l'emploi 1. Und es ist fertig.