

Programmation dynamique

Exercice 1 : Sous-sommes maximales

1. Une simple boucle suffit, comme pour calculer la somme des éléments d'une liste.

```

1 def sous_somme(L,p,q):
2     s=0
3     for i in range(p,q+1):
4         s+=L[i]
5     return s

```

2. On calcule toutes les sommes en utilisant à chaque fois la fonction de la question précédente. Une variable m sert à retenir en mémoire la plus grande valeur calculée.

```

1 def somme_max_naive(L):
2     m=L[0]; n=len(L)
3     for p in range(n):
4         for q in range(p,n):
5             s=sous_somme(L,p,q)
6             if s > m:
7                 m=s
8     return m

```

La complexité est en $O(n^3)$: deux boucles imbriquées qui font appel à une fonction de coût linéaire.

3. L'idée consiste simplement à stocker dans une variable temporaire la valeur $T[p] + \dots + T[q]$ au fur et à mesure des deux boucles. Lorsque q augmente, il suffit d'une addition pour la mettre à jour, et elle est réinitialisée à 0 pour chaque nouvelle valeur de p .

```

1 def somme_max_naive2(L):
2     m=L[0]; n=len(L)
3     for p in range(n):
4         s=0
5         for q in range(p,n):
6             s+=L[q]
7             if s > m:
8                 m=s
9     return m

```

4. La preuve ci-dessous est basée sur les deux égalités

$$\max(a+b, a+c) = \max(a, b) + c$$

$$\text{et} \quad \max(A \cup B) = \max(\max A, \max B)$$

valable pour tous $a, b, c \in \mathbb{R}$, et tous $A, B \subset \mathbb{R}$ finies. Soit $i \in \llbracket 0; n-2 \rrbracket$.

- Par définition, en séparant dans la définition de ℓ_{i+1} le dernier terme $p = i+1$ du reste, il vient

$$\begin{aligned} \ell_{i+1} &= \max_{0 \leq p \leq i+1} (L[p] + \dots L[i+1]) \\ &= \max \left(L[i+1], \max_{0 \leq p \leq i} (L[p] + \dots L[i+1]) \right) \end{aligned}$$

De plus,

$$\begin{aligned} \max_{0 \leq p \leq i} (L[p] + \dots L[i+1]) &= L[i+1] + \max_{0 \leq p \leq i} (L[p] + \dots L[i]) \\ &= L[i+1] + \ell_i \end{aligned}$$

$$\begin{aligned} \text{Ainsi,} \quad \ell_{i+1} &= \max(L[i+1], L[i+1] + \ell_i) \\ &= L[i+1] + \max(\ell_i, 0) \end{aligned}$$

- Par ailleurs, en séparant dans la définition de m_{i+1} les sommes qui font intervenir $q = i+1$ et celles pour lesquelles $q < i+1$, on a directement

$$\begin{aligned} m_{i+1} &= \max_{0 \leq p \leq q \leq i+1} (L[p] + \dots L[q]) \\ &= \max \left(\max_{0 \leq p \leq q \leq i} (L[p] + \dots L[q]), \max_{0 \leq p \leq i} (L[p] + \dots L[i+1]) \right) \\ m_{i+1} &= \max(m_i, \ell_{i+1}) \end{aligned}$$

5. Il suffit maintenant d'une seule boucle pour calculer par récurrence les valeurs $(m_i)_{i \in \llbracket 0; n-1 \rrbracket}$ et $(\ell_i)_{i \in \llbracket 0; n-1 \rrbracket}$. Les mises à jour se faisant en temps constant, la complexité est bien linéaire.

```

1 def somme_max(L):
2     n = len(L); m=L[0]; l=L[0]
3     for i in range(1,n):
4         l=L[i]+max(0,l)
5         m=max(m,l)
6     return m

```

6. Outre la variable m stockant la plus grande somme calculée jusqu'alors, on utilise également trois variables a , b et c pour stocker les indices qui réalisent le maximum dans les définitions de m_i et ℓ_i .

- La variable c est utilisée pour ℓ_i . Elle est mise à jour lorsque ℓ_i devient négatif, car cela correspond au cas où $\ell_{i+1} = L[i+1]$ et donc que la somme maximale faisant intervenir $L[i+1]$ est réalisée en ne prenant que cette valeur dans la somme.
- Les variables a et b sont utilisées pour m_i . La mise à jour s'effectue cette fois lorsque $\ell_{i+1} > m_i$, car il faut dans ce cas mettre à jour m_{i+1} , qui est alors réalisé avec les indices c et $i+1$.

```

1 def indices_max(L):
2     n = len(L); m=L[0]; l=L[0]
3     a=0; b=0; c=0
4     for i in range(1,n):
5         if l<0:
6             c=i
7             l=L[i]+max(0,l)
8             if l>m:
9                 a,b=c,i
10            m=max(m,l)
11    return a,b

```

Exercice 2 : Problème de la scierie

1. Dans un premier temps, on calcule pour tout entier k l'indice i_k tel que

$$\frac{p_{i_k}}{i_k} = \max_{i \in \llbracket 1;k \rrbracket} \frac{p_i}{i}$$

Le calcul de $i_1, \dots, i_n$ peut se faire pour un coût linéaire de la manière suivante :

```

1 def rentabilite(P):
2     I=[0]; rm=0; im=0
3     for i in range(1,len(P)):
4         if P[i]/i > rm:
5             I.append(i); rm=P[i]/i; im=i
6         else:
7             I.append(im)
8     return I

```

Cette fonction permet ensuite de savoir en coût constant quel est la première découpe à faire pour une planche de longueur quelconque inférieure ou égale à n . La fonction de découpage glouton peut alors s'écrire de la manière suivante :

```

1 def decoupage_glouton(P,n):
2     R=rentabilite(P)
3     def decoupe(k):
4         if k==0:
5             return []
6         else:
7             return [R[k]]+decoupe(k-R[k])
8     return decoupe(n)

```

2. Pour un contre-exemple, il suffit d'avoir

$$3p_1 < p_3 \quad p_1 + p_2 < p_3 \quad \text{et} \quad \frac{p_3}{3} < \frac{p_2}{2}$$

par exemple $p_1 = 1$ $p_2 = 6$ et $p_3 = 8$

Les deux premières conditions garantissent que le prix optimal pour vendre une planche de longueur 3 consiste à la laisser intacte. Mais la dernière condition garantit que l'algorithme glouton choisira pourtant de découper d'abord un morceau de longueur 2.

3. Soit $k \in \mathbb{N}$. Cherchons le prix optimal pour une planche de longueur $k + 1$. Pour vendre une telle planche,

- Soit on ne la découpe pas, et on la vend pour un prix p_{k+1} ;
- soit on commence par découper un morceau de longueur $r \in \llbracket 1; k \rrbracket$ que l'on vendra au prix p_r , puis on redécoupe de manière optimale la planche restante de longueur $k + 1 - r$ pour un prix m_{k+1-r} , ce qui donne un prix total égal à $p_r + m_{k+1-r}$.

Le prix optimal est donc obtenu en prenant la plus grande valeur parmi ces quantités.

$$\forall k \in \mathbb{N}, \quad m_{k+1} = \max_{r \in \llbracket 1; k+1 \rrbracket} (p_r + m_{k+1-r})$$

Par suite, les éléments de E_{k+1} s'obtiennent en ajoutant l'entier r à n'importe quel élément de E_{k+1-r} où r est un entier quelconque réalisant le maximum dans la formule précédente.

4. Le calcul de $m_0, \dots, m_n$ se fait de proche en proche en utilisant la relation de récurrence précédente.

```

1  def prix_optimal(p):
2      n=len(p)-1
3      m=[0 for i in range(n+1)]
4      m[1]=p[1]
5      for k in range(1,n):
6          g=p[k+1]; ind=k+1
7          for r in range(1,k+1):
8              if p[r]+m[k+1-r] > g:
9                  ind=r; g=p[r]+m[k+1-r]
10         m[k+1]=g
11     return m[n]
```

5. Il n'y a pas grand chose à changer pour l'adaptation. En plus du calcul de $m_0, \dots, m_n$, on calcule pour chaque valeur de k , un découpage optimal l_k sous la forme d'une liste d'entiers comme indiqué. Le calcul de l_{k+1} utilise la remarque en fin de question 4.

```

1  def decoupage(p):
2      n=len(p)-1
3      m=[0 for i in range(n+1)]
4      l=[[ ] for i in range(n+1)]
```

```

5      m[1]=p[1]; l[1]=[1]
6      for k in range(1,n):
7          g=p[k+1]; ind=k+1
8          for r in range(1,k+1):
9              if p[r]+m[k+1-r] > g:
10                 ind=r; g=p[r]+m[k+1-r]
11         m[k+1]=g; l[k+1]=[ind]+l[k+1-ind].copy()
12     return m[n],l[n]
```

Ecole Polytechnique (Tronc commun) 2006

1. Pour cette fonction, il s'agit de faire la somme des distances parcourues par chaque candidat (à l'aide d'une référence `s` initialisée à 0). On utilise une liste `p` de longueur 2 pour stocker les positions courantes des élèves au fur et à mesure des examens : `p[0]` contient la position du candidat 1, `p[1]` celle du candidat 2. A noter que l'on aurait parfaitement pu créer deux variables mais l'utilisation d'une liste permet d'éviter un test de branchement.

```

1 def cout_seq(a,d):
2     s=0; n=len(d)
3     p=[0,0]
4     for i in range(0,n):
5         s+=abs(a[i+1] - p[d[i]-1])
6         p[d[i]-1]=a[i+1]
7     return(s)

```

2. Le programme précédent s'effectue en $O(n)$. Puisqu'il y a 2^n séquences de déplacements possibles, on a une complexité totale de $O(n \cdot 2^n)$, ce qui est inexploitable du point de vue informatique.
3. S'il n'y a que deux examens à passer, n'importe quel élève peut se présenter au premier. Pour le second, il suffit alors d'envoyer l'élève le plus proche. On écrit donc le code suivant

```

1 def cout_opt2(a):
2     return a[1] + min(a[2], abs(a[1]-a[2]))

```

4. La seule différence avec `cout_seq` est que dans la boucle, la personne se déplaçant n'est plus donné par la quantité `d[i]` mais obtenue par un test comparant les distances de chaque élève au prochain centre d'examen.

```

1 def cout_naif(a):
2     s=0; n=len(d)
3     p=[0,0]
4     for i in range(n):
5         if abs(p[0]-a[i+1]) > abs(p[1]-a[i+1]):
6             c=1
7         else:
8             c=0
9         s+=abs(a[i+1]-p[c])
10        p[c]=a[i+1]
11    return s

```

5. Pour le tableau d'adresse `[10;20;9;1]`, le programme renvoie le coût 37 (qui correspond à la séquences de déplacement `[1;2;2]` par exemple). Toutefois, pour la séquence `[1;1;2]`, on obtient un coût de 32, ce qui montre que la stratégie naïve n'est pas optimale.
6. La quantité $c_{0,1}$ est le coût optimal pour que le candidat 2 ait passé la première épreuve tandis que le candidat 1 n'a rien passé. Elle est donc clairement égale à a_1 . Par symétrie, $a_{1,0}$ est également égale à a_1 .
7. Distinguons les deux cas de l'énoncé :

- Considérons un déplacement `d` optimal pour arriver en configuration (i, j) . Si $i < j - 1$, cela signifie que le candidat 1 n'a pas passé l'épreuve $j - 1$ (il n'a rien fait depuis la i -ième épreuve). Par suite, c'est le candidat 2 qui s'en est chargé. Le déplacement pour aller en position $(i, j - 1)$ s'est nécessairement fait de manière optimale et a donc pour coût $c_{i,j-1}$. Pour obtenir $c_{i,j}$, il faut alors rajouter le coût du dernier déplacement, qui est celui du candidat 2 de l'adresse a_{j-1} à l'adresse a_j .
- Pour le deuxième formule, on procède par double inégalité.

$\boxed{\geq}$ Soit `d` déplacement optimal pour arriver en configuration $(i, i + 1)$. Le candidat 2 a passé l'épreuve $i + 1$. Avant cela, il n'a pas pu passer l'épreuve i , c'est 1 qui s'en est chargé. Soit donc $k_0 \in \llbracket 0; i - 1 \rrbracket$ l'indice de la dernière épreuve qu'il a passé. A l'issue de la i -ième épreuve, on est donc en configuration (i, k_0) . Nécessairement, le déplacement pour aboutir à cette configuration est optimal, donc de coût c_{i,k_0} . Ainsi, `d` est de coût égal à cette quantité à laquelle on ajoute le coût du dernier déplacement pour aller passer la $(i + 1)$ -ième épreuve, soit $|a_{k_0} - a_{i+1}|$. Ainsi,

$$c_{i,i+1} = c_{i,k_0} + |a_{k_0} - a_{i+1}|$$

et en particulier,

$$c_{i,i+1} \geq \min_{k \in \llbracket 0; i-1 \rrbracket} c_{i,k} + |a_k - a_{i+1}|$$

$\boxed{\leq}$ Réciproquement, fixons $k < i$ et considérons un déplacement optimal jusqu'en configuration (i, k) (le premier candidat passe donc la dernière épreuve i). Si à l'issue de ce déplacement, le candidat 2 passe l'épreuve $i + 1$, on aboutit en configuration $(i, i + 1)$ pour un coût total $c_{i,k} + |a_k - a_{i+1}|$. Par définition du coût minimal $c_{i,i+1}$, on a donc

$$c_{i,i+1} \leq c_{i,k} + |a_k - a_{i+1}|$$

et puisque cette inégalité est valable pour tout $k \in \llbracket 0; i-1 \rrbracket$, il vient

$$c_{i,i+1} \leq \min_{k \in \llbracket 0; i-1 \rrbracket} c_{i,k} + |a_k - a_{i+1}|$$

Le calcul de $c_{i,j}$ peut alors se faire dans l'ordre suivant :

- On commence par calculer $c_{0,1}$ (qui vaut a_1), puis $c_{0,2}, \dots, c_{0,n}$ grâce à la première formule.

On en déduit les valeurs de $c_{1,0}, \dots, c_{n,0}$ par symétrie.

- Plus généralement, à la i -ième étape, on commence par calculer $c_{i,i+1}$ grâce à la seconde formule, puis $c_{i,p}$ pour tout $p \geq i+2$ via la première formule.

On complète ensuite les valeurs $c_{p,i}$ pour $p \geq i+1$ par symétrie.

8. Le programme suivant est le résultat du raisonnement de la question précédente. On commence par définir la fonction permettant de calculer l'entier k réalisant le minimum dans la seconde formule de récurrence (en supposant le tableau partiellement rempli).

```

1 def trouve_min(c,a,i):
2     if i==0:
3         return (0,a[1])
4     else:
5         min=0
6         cout_min = c[i][0]+abs(a[i+1]-a[0])
7         for k in range(1,i):
8             cout_k = c[i][k]+abs(a[i+1]-a[k])
9             if cout_k < cout_min:
10                 min=k; cout_min = cout_k
11     return min,cout_min

```

On construit alors une fonction qui calcule le tableau c . Même si ce n'est pas exigé dans cette question, on construit en parallèle un tableau $kopt$. Ce dernier recense les valeurs de l'entier k qui réalise le minimum lors du calcul des valeurs $c_{i,i+1}$ et $c_{i+1,i}$. Il servira pour la reconstitution des déplacements optimaux.

```

1 def calcule_c_kopt(a):
2     n=len(a)-1
3     c=[((-1) for j in range(n+1)) for i in range
4         (n+1)]
5     kopt = [(-1) for i in range(n+1)]
6     for i in range(n):
7         (x,y)=trouve_min(c,a,i)
8         c[i][i+1]=y; c[i+1][i]=y
9         kopt[i]=x
10        for p in range(i+2,n+1):
11            c[i][p]=c[i][p-1]+abs(a[p]-a[p-1])
12            c[p][i]=c[i][p]
13    return c,kopt

```

Pour trouver le coût minimal, il suffit d'extraire la plus grande valeur de la dernière ligne de c (sans prendre en compte la dernière case).

```

1 def cout_opt(a):
2     c = calcule_c_kopt(a)[0]
3     n=len(a)-1
4     return(min(c[n][: -1]))

```

9. Pour contruire le chemin optimal pour aboutir en configuration (i,j) , construisons une fonction récursive qui utilise les deux cas de la formule de récurrence :

- Si $i < j-1$, alors c'est nécessairement 2 qui a passé l'épreuve 2, et on provient forcément de la configuration $(i,j-1)$. On fera donc un appel récursif sur le couple $(i,j-1)$.
- Si $j = i+1$, c'est à nouveau 2 qui a passé l'épreuve $i+1$. On provient d'une configuration de forme (i,k) avec $k < i$. La valeur de k en question a été judicieusement stockée dans le tableaux $choix$. On pourra donc récupérer cette valeur sans calcul et faire un appel récursif sur le couple (i,k) .
- Le cas $j = i-1$ et $i > j$ sont symétriques.

La fonction ci-dessous implémente cette méthode en partant d'une configuration de la forme (n,k) pour laquelle $c_{n,k}$ est égal au coût optimal. Cet entier k est déterminé à l'aide d'une fonction qui trouve la position du minimum parmi les valeurs $l[i], \dots, l[j]$ d'une liste l .

```
1 def indice_min(l,i,j):
2     a=i
3     for k in range(i+1,j+1):
4         if l[k]<l[a]:
5             a=k
6     return a
```

On complète un tableau D au fur et à mesure des appels récursifs, et ce dernier est renvoyé une fois que les appels récursifs sont revenus à la configuration (0,0).

```
1 def trajet_opt(a):
2     n=len(a)-1
3     c,kopt=calcul_c_kopt(a)
4     d=[(-1) for i in range(n)]
5     def aux(i,j):
6         print((i,j))
7         print(d)
8         if (i,j)!=(0,0):
9             if i<j:
10                d[j-1]=2
11                if j>i+1:
12                    aux(i,j-1)
13            else:
14                aux(i,kopt[i])
15        else:
16            d[i-1]=1
17            if i>j+1:
18                aux(i-1,j)
19            else:
20                aux(kopt[j],j)
21    k=indice_min(c[n],0,n-1)
22    aux(n,k)
23    return d
```