td1a_plus_grande_somme_correction

October 20, 2021

1 1A.algo - la plus grande sous-séquence croissante - correction

Un exercice classique : trouver la plus grande sous-séquence croissante. Celle-ci n'est pas nécessairement un bloc contigü de la liste initiale mais les entiers apparaissent dans le même ordre.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

L'algorithme optimal est exposé en dernier, la correction propose un cheminement jusqu'à cette solution en introduisant au fur et à mesure les idées qui aboutissent à cette solution. A partir de la première solution, on élague peu à peu :

1. ce qu'il n'est pas nécessaire de faire car mathématiquement inutile et ne changeant pas la solution,
2. ce qu'il n'est pas nécessaire de conserver d'une itération à l'autre car cette information peut être reconstruite et qu'il coûte plus cher de la stocker que de la reconstruire.

$E[k]$, E_k désignent l'élément d'indice k de l'ensemble E. Sauf précision contraire, il est tenu compte de l'ordre de tous les éléments dans l'ensemble auxquels ils appartiennent.

1.1 Solution simple

Prenons un tableau quelconque :

[2]:
```
E = [10, 15, 7, 19, 2, 5, 7, 16, 3, 9, 15, 0, 1, 15, 6, 11, 0, 14, 7, 9]
E
```

[2]:
```
[10, 15, 7, 19, 2, 5, 7, 16, 3, 9, 15, 0, 1, 15, 6, 11, 0, 14, 7, 9]
```

Avant de passer à l'algorithme dichotomique, on va d'abord suivre un chemin plus facile et plus lent. Supposons qu'on connaît la meilleure séquence croissante de longueur n : $s(1 \rightarrow n)$.
On découpe cette séquence en deux $s(1 \rightarrow k)$ et $s(k+1 \rightarrow n)$. On est sûr que la séquence $s(1 \rightarrow k)$ est la plus grande séquence croissante incluant l'élément s_k. Dans le cas contraire, on aurait trouvé un moyen de fabriquer une plus longue séquence croissante sur E. Et c'est contradictoire avec l'hypothèse de départ.
A chaque indice k, il existe une meilleure séquence $S[k]$ se terminant à la position k : $E[k]$ est le dernier élément de la séquence. On suppose que toutes les meilleures séquences croissantes se terminant à la position i pour $i \in [1:k]$. Comment calculer la meilleure séquence croissante pour la position $i+1$? D'après ce qui précède, il suffit de l'ajouter à toutes les séquences qui précèdent puis de choisir la meilleure. Une fois qu'on a obtenu toutes les meilleures séquences se terminant aux positions i, il suffit de prendre la plus longue.

[3]:
```
def plus_grande_sequence_position_k(E, k=None):
    if k is None:
        k = len(E)-1
```

```
    if k == 0:
        return [[0]]
    else :
        S = plus_grande_sequence_position_k(E, k-1)

        best = []
        for j,s in enumerate(S):
            if len(s) > len(best) and E[k] >= E [s[-1]]:
                best = s
        best = best + [k]
        S.append(best)
        return S

def plus_grande_sequence(E):
    if len(E) == 0:
        return E
    S = plus_grande_sequence_position_k(E)
    best = []
    for s in S:
        if len(s) > len(best):
            best = s
    return best

b = plus_grande_sequence(E)
"E",E,"indice:",b, "valeurs:", [ E[i] for i in b ]
```

[3]:
```
('E',
 [10, 15, 7, 19, 2, 5, 7, 16, 3, 9, 15, 0, 1, 15, 6, 11, 0, 14, 7, 9],
 'indice:',
 [4, 5, 6, 9, 10, 13],
 'valeurs:',
 [2, 5, 7, 9, 15, 15])
```

Le coût de cet algorithme est en $O(n^2)$. L'énoncé de l'exercice suggère qu'on peut faire mieux en utilisant la dichotomie. En coupant l'ensemble E en deux, $A = E(1 \to k)$ et $B = E(1 \to k+1)$, soit la plus grande séquence croissante est dans A, soit dans B, soit elle commence avant la position k et se termine après. Les deux premiers cas sont très simples à traiter par récurrence. Le dernier l'est moins mais on sait deux choses : on cherche la séquence $s(a \to b)$ avec $a \leqslant k \leqslant b$ et la recherche de cette séquence doit avant un coût en $O(n)$ sinon le nouvel algorithme ne sera pas plus rapide que le précédent.

1.2 Solution simple améliorée

Est-il nécessaire de garder l'historique des séquences ? Pour chaque position k, on conserve toute la meilleure séquence se terminant à la position k. Toutefois, il est possible de ne retenir que l'élément précédent : $P[E[k]]$ car la meilleure séquence $S[k]$ peut être décomposée en $S[E[k]] + [E[k]]$. Cette amélioration rentre dans la catégorie 2 : l'algorithme conserve une information non indispensable.
La fonction est plus simple à implémenter de façon non récursive. Elle se sert de deux tableaux :

- $longueur[k]$: conserve la longueur de la meilleure séquence se terminant à la position k
- $precedent[k]$: conserve la position de l'élément précédent dans la meilleure séquence se terminant à la position k (donc précédent k).

[4]:
```
def plus_grande_sequence_2(E):
    if len(E) == 0:
```

```
        return E

    precedent = [-1 for e in E]
    longueur  = [0 for e in E]

    longueur[0] = 1
    for k in range(1, len(E)):

        bestL = 1
        bestP = -1

        for j in range(0,k) :
            if E[k] >= E [ j ] and longueur[j]+1 > bestL:
                bestL = longueur [j]+1
                bestP = j

        precedent[k] = bestP
        longueur[k] = bestL

    # on récupère la longueur de la plus grande séquence
    maxiL = 0
    for i,l in enumerate(longueur):
        if l > longueur[maxiL]:
            maxiL = i

    # on récupère la plus grande séquence
    seq = [maxiL]
    while precedent[seq[-1]] != -1:
        p = precedent[seq[-1]]
        seq.append(p)

    seq.reverse()
    return seq

E = [10, 15, 7, 19, 2, 5, 7, 16, 3, 9, 15, 0, 1, 15, 6, 11, 0, 14, 7, 9]
b = plus_grande_sequence_2(E)
"E",E,"indice:",b, "valeurs:", [ E[i] for i in b ]
```

[4]:
```
('E',
 [10, 15, 7, 19, 2, 5, 7, 16, 3, 9, 15, 0, 1, 15, 6, 11, 0, 14, 7, 9],
 'indice:',
 [4, 5, 6, 9, 10, 13],
 'valeurs:',
 [2, 5, 7, 9, 15, 15])
```

On compare les coûts. La seconde fonction est un peu plus rapide à un facteur multiplicatif près. Le coût des deux fonctions est $O(n^2)$.

[5]:
```
import random
for n in (20,50,100,200) :
    E = [ random.randint(0,n) for i in range(n) ]
    print("n=",n)
    %timeit plus_grande_sequence(E)
    %timeit plus_grande_sequence_2(E)
```

```
n= 20
89.9 µs ± 18.2 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
38.8 µs ± 3.08 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
n= 50
324 µs ± 41.6 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
186 µs ± 21.5 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
n= 100
1.18 ms ± 94 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
653 µs ± 125 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
n= 200
4.86 ms ± 715 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
2.24 ms ± 148 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

1.3 Un peu plus près de la solution optimale

La fonction précédente se termine par deux boucles. La première détermine la longueur de la plus longue séquence, la seconde reconstruit la séquence. Pour chaque k, on conserve la meilleure séquence croissante se terminant à la position k. Mais supposons que nous sommes à l'itération l, et pour les positions $j < k < l$, et que les deux meilleures séquences se terminant aux positions k et j ont même longueur et que $E[k] < E[j]$, est-il vraiment nécessaire de conserver une seconde séquence aussi longue mais dont le dernier élément est plus grand ?

			j	k	...	l
S_j	2	3	5			
S_k	2	3		4		

La réponse est évidemment non. Par extension, à la position l, on peut classer toutes les séquences se terminant avant :

- par longueur décroissante
- par dernier élément croissant

Pour chaque longueur, on va conserver le plus petit dernier élément possible parmi toutes les séquences de cette longueur. L'optimisation est deux types : l'algorithme est plus rapide (son coût est plus faible), il stocke moins d'information.

[6]:
```python
def plus_grande_sequence_2L(E):
    if len(E) == 0:
        return E

    dernier   = [0]
    precedent = [-1 for e in E]

    for k in range(1,len(E)):
        if E[k] >= E [dernier [-1]]:
            # on ajoute à la dernière séquence
            precedent[k] = dernier[-1]
            dernier.append( k )
        else :
            # on s'en sert pour améliorer une séquence existante
            for j in range(len(dernier)-1, -1, -1):
                if E[k] < E [dernier[j]]:
                    if precedent[dernier[j]] == -1:
                        dernier [j] = k
                    elif E[k] >= E[dernier[j-1]]:
                        if j == 0:
                            break
```

```
                        precedent[k] = dernier[j-1]
                            # ce n'est pas exactement precedent[dernier[j]],
                            # mais cette valeur est admissible par construction
                        dernier[j] = k
                        break  # car il ne sert à rien d'aller plus loin


    # on récupère la plus grande séquence
    seq = [dernier[-1]]
    while precedent[seq[-1]] != -1:
        p = precedent[seq[-1]]
        seq.append(p)

    seq.reverse()
    return seq

E = [10, 15, 7, 19, 2, 5, 7, 16, 3, 9, 15, 0, 1, 15, 6, 11, 0, 14, 7, 9]
b = plus_grande_sequence_2L(E)
"E",E,"indice:",b, "valeurs:", [ E[i] for i in b ]
```

[6]:
```
('E',
 [10, 15, 7, 19, 2, 5, 7, 16, 3, 9, 15, 0, 1, 15, 6, 11, 0, 14, 7, 9],
 'indice:',
 [4, 5, 6, 9, 15, 17],
 'valeurs:',
 [2, 5, 7, 9, 11, 14])
```

On compare les coûts. La seconde fonction est un peu plus rapide à un facteur multiplicatif près. Le coût de la première fonction est en $O(n^2)$. La seconde est en $O(nL)$ où L est la longueur de la plus longueur séquence. On majore ce coût par $O(n^2)$ mais dans les faits, c'est plus court.

[7]:
```
import random
for n in (20,50,100,200) :
    E = [random.randint(0,n) for i in range(n)]
    %timeit plus_grande_sequence_2(E)
    %timeit plus_grande_sequence_2L(E)
```

```
37.9 µs ± 5.51 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
19.3 µs ± 2.62 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
162 µs ± 8.65 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
76.5 µs ± 4.98 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
568 µs ± 25.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
137 µs ± 2.8 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
2.27 ms ± 192 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
357 µs ± 57.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

1.4 Solution optimale

Enfin, la solution proposée sur la page wikipedia. Elle consiste simplement à remplacer la boucle imbriquée par une recherche dichotomique. En effet, on cherche le premier élément plus petit qu'un autre dans un tableau trié. Cela peut être aisément remplacé par une recherche dichotomique. Les paragraphes qui suivent reprennent l'explication depuis le début avec les notations de l'article wikipedia.
L'idée consiste à parcourir le tableau de gauche à droite en conservant à chaque itération des séquences optimales de longueur 1 à L en s'assurant que chaque séquence se termine par le plus petit entier possible.

L'implémentation s'appuie sur un tableau $M_{1\to n}$. La variable L mémorise la longueur de la plus grande séquence connue jusque là. A l'itération i, $M[L]$ contient l'indice du dernier élément de la meilleure séquence croissante de longueur L dans l'ensemble $E(1 \to i)$. $M[k]$ pour $1 \leqslant i \leqslant i$ contient l'indice du dernier élément d'une séquence de k nombre compris entre les indices 1 et i.
A chaque itération $i+1$, on met à jour le tableau M en considérant l'élément $E[i+1]$. Tout d'abord, si $E[i+1] \geqslant E[M[L]]$, cela veut dire qu'on peut ajouter l'élément $E[i+1]$ à la plus grande séquence connue, elle sera nécessairement la plus grande. Si maintenant $E[i+1] < E[M[L]]$, il n'y aura pas de meilleure séquence. Pour autant, cet élément remplacera le dernier élément d'une séquence de longueur moindre s'il est plus petit. On peut aisément comprendre cela : si deux séquences ont même longueur, celle se terminant par le plus petit élément a plus de chance de s'agrandir par la suite.
L'algorithme repose sur une propriété du tableau M : la suite $E[M[i]]$ est croissante entre 1 et L. On peut dire cela autrement : il existe une séquence de longueur $L-1$ dont le dernier élément est nécessaire plus petit que le dernier élément de la meilleure séquence de longueur L. Il suffit que prend les $L-1$ premiers éléments de la meilleure séquence de longueur L.

[8]:
```
def plus_grande_sequence_wikipedia(E):
    P = [-1 for _ in E]
    M = [-1 for _ in E]

    L = 0
    for i in range(0, len(E)):
        lo = 1
        hi = L
        while lo <= hi:
            mid = (lo + hi) // 2
            if E[M[mid]] < E[i]:
                lo = mid + 1
            else:
                hi = mid - 1

        newL = lo
        P[i] = M[newL - 1]

        if newL > L:
            M[newL] = i
            L = newL
        elif E[i] < E[M[newL]]:
            M[newL] = i

    S = [-1 for i in range(L)]
    k = M[L]
    for i in range(L-1, -1, -1) :
        S[i] = k
        k = P[k]

    return S

E = [10, 15, 7, 19, 2, 5, 7, 16, 3, 9, 15, 0, 1, 15, 6, 11, 0, 14, 7, 9]
b = plus_grande_sequence_wikipedia(E)
"E",E,"...","indice:",b, "valeurs:", [ E[i] for i in b ]
```

[8]:
```
('E',
 [10, 15, 7, 19, 2, 5, 7, 16, 3, 9, 15, 0, 1, 15, 6, 11, 0, 14, 7, 9],
 '…',
```

```
'indice:',
[4, 5, 6, 9, 15, 17],
'valeurs:',
[2, 5, 7, 9, 11, 14])
```

On compare avec la version précédente et on vérifie qu'elle est plus rapide.

[9]:
```
import random
for n in (20,50,100,200) :
    E = [ random.randint(0,n) for i in range(n) ]
    print("n=",n)
    %timeit plus_grande_sequence_2L(E)
    %timeit plus_grande_sequence_wikipedia(E)
```

```
n= 20
21.9 µs ± 1.46 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
15.8 µs ± 877 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
n= 50
60.3 µs ± 4.66 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
42.9 µs ± 3.38 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
n= 100
127 µs ± 4.52 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
102 µs ± 9.96 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
n= 200
411 µs ± 42.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
228 µs ± 15 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[10]: