Quelques exemples de programmation dynamique

1 Rendu de monnaie

Dans cette partie, on cherche à réaliser un programme de rendu de monnaie. On souhaite que l'automate rende l'appoint de manière optimale, dans le sens où il minimise le nombre de pièces rendues (/billets). On se place pour le moment avec le système européen

$$\mathscr{P} = \{1; 2; 5; 10; 20; 50; 100; 200\}$$

> Un algorithme **glouton** est un algorithme localement optimal. Il ne donne pas toujours le meilleur résultat, mais « parfois » un résultat satisfaisant.

Exercice 1 Écrire une fonction qui reçoit une somme entière et renvoie ou affiche la répartition des pièces selon l'algorithme glouton.

A présent, on considère le système de monnaie britannique d'avant 1971 qui utilisait les multiples suivants du penny : $\mathscr{P} = \{1; 3; 6; 12; 24; \mathbf{30}\}$... Avec ce système, l'algorithme glouton décompose

- 48 pennies en : 48 = 30 + 12 + 6
- alors que la décomposition optimale est : 48 = 24 + 24.

> Une nouvelle stratégie s'impose...

Si on note $N(s)$ le nombre minimal de pièces nécessaires pour obtenir la somme s :

- $N(0) = 0$
- $N(s) = 1 + \min\limits_{p \in \mathscr{P} \mid p \leqslant s} N(s-p)$

Exercice 2 Une solution récursive :

1. Écrire cette fonction récursive.
2. L'améliorer pour obtenir en plus la répartition de pièces.

Exercice 3 Pour éviter les redondances de calculs, proposer une solution Top-Down avec mémoïsation.

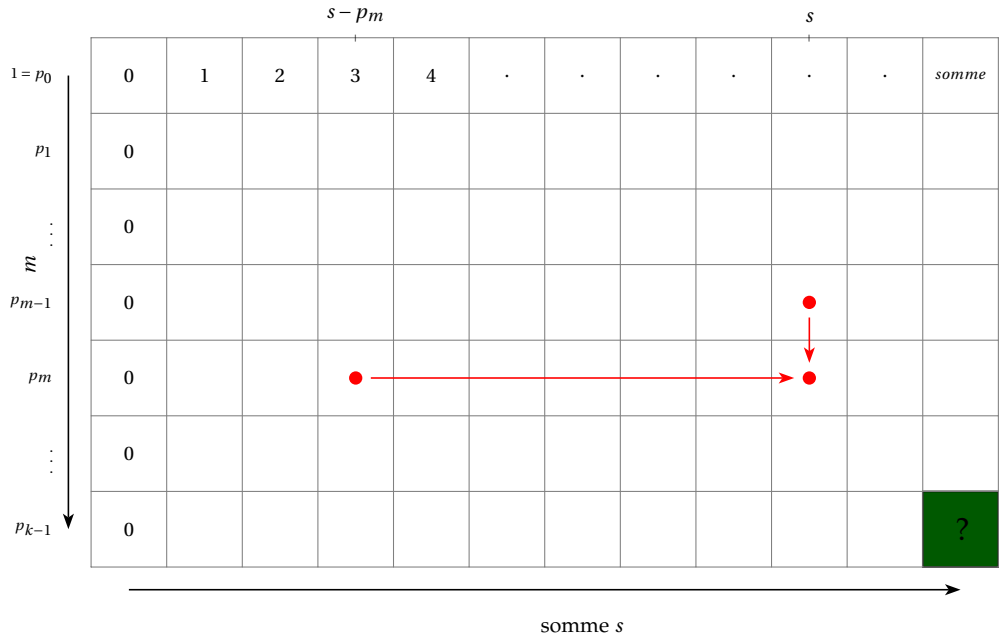
> A la recherche d'une solution dynamique...

Notons,

- p_0, p_1, p_{k-1} les éléments de $\mathscr{P}$ toujours classés par ordre croissant.
- $N_{m,s}$ le nombre minimal de pièces pour obtenir la somme s, mais en n'utilisant que des pièces du sous-ensemble $\{p_0, p_1, ..., p_m\}$

On cherche $N(s) = N_{k-1,somme}$ où k est le nombre de pièces différentes et $somme$ la somme souhaitée. On a alors :

$$N_{m,s} = \begin{cases} 0 \text{ si } s = 0 \\ N_{m-1,s} \text{ si } s < p_m \\ \min \begin{cases} \underbrace{N_{m-1,s}}_{\text{On ne prend pas la pièce } p_m} ; & \underbrace{1 + N_{m,s-p_m}}_{\text{On prend la pièce } p_m} \end{cases} \end{cases}$$



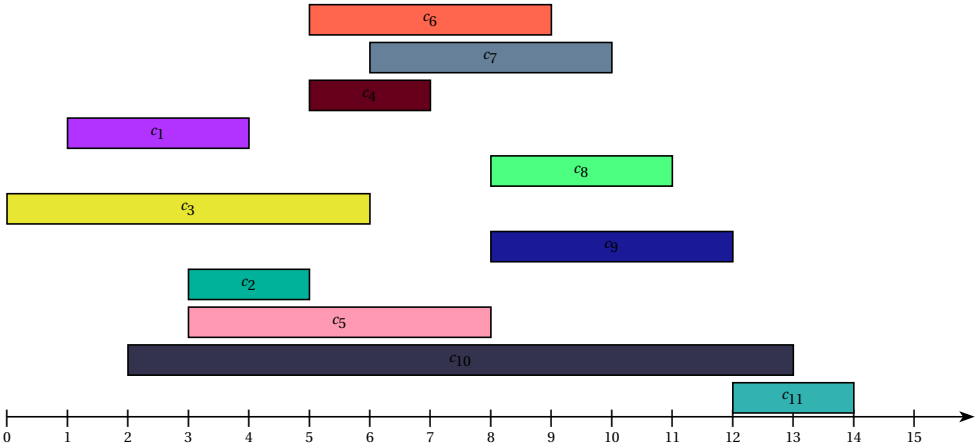
Exercice 4 Avec ce principe, imaginer une fonction recevant la somme à donner et la liste des pièces disponibles et qui renvoie le nombre minimal de pièces.

Exercice 5 Améliorer la fonction précédente pour qu'elle renvoie le nombre minimal de pièces ainsi que la répartition de celles-ci.

Exercice 6 Proposer une dernière amélioration qui n'utilise qu'un tableau à une dimension et limite de fait la complexité spatiale.

2 Organisation d'un festival.

Lors d'un festival, n concerts c_1, c_2, ... c_n sont organisés. Chaque concert c_i est défini par son heure de début d_i et de fin f_i. On convient qu'il est possible d'assister aux concerts c_i et c_j dès lors que $[d_j; f_i[\cap [d_j; f_j[= \emptyset$, c'est à dire que les intervalles ne se chevauchent pas.

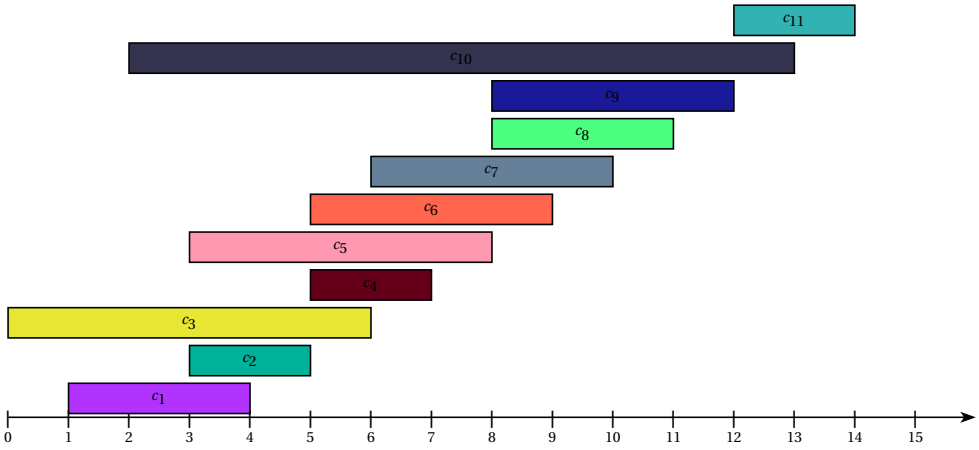


Comment choisir ses concerts pour en suivre un maximum ?

Exercice 7 La programmation des concerts est modélisée par une liste où chaque élément est un concert c_i représenté par une liste $[d_i, f_i]$. Dans notre exemple `C = [[1,4], [3,5], [0,6], [5,7], [3,8], [5,9], [6,10], [8,11], [8,12], [2,13], [12,14]]`

1. Écrire une fonction qui reçoit la programmation ainsi que deux entiers i et j et renvoie un booléen indiquant si on peut ou non assister aux concerts c_i et c_j.
2. Comment savoir si on peut assister c_1, c_8 et c_5 ?

En triant une fois pour toutes en amont les concerts par heure de fin, on peut améliorer la fonction précédente :



3. Programmer une fonction qui reçoit la programmation $[c_k]$ ainsi qu'une liste I d'indices et renvoie un booléen indiquant si on peut suivre l'intégralité des concerts c_i pour $i \in I$.

Python propose un module **`itertools`** qui permet de générer facilement des **itérateurs** pour donner une liste exhaustive des cas répondant à certaines situations.

- Permutations (tous les mélanges possibles)

```
import itertools

E = ['A', 'B', 'C']
for cas in itertools.permutations(E) :
    print (cas,end=' ; ')
```

```
('A', 'B', 'C') ; ('A', 'C', 'B') ; ('B', 'A', 'C') ;
('B', 'C', 'A') ; ('C', 'A', 'B') ; ('C', 'B', 'A') ;
```

- p-listes : (tirages avec remise)

```
import itertools

E = [1, 2, 3, 4, 5]
for cas in itertools.product(E,repeat=2) :
    print (cas,end=' ; ')
```

```
(1, 1) ; (1, 2) ; (1, 3) ; (1, 4) ; (1, 5) ; (2, 1) ; (2, 2) ;
(2, 3) ; (2, 4) ; (2, 5) ; (3, 1) ; (3, 2) ; (3, 3) ; (3, 4) ;
(3, 5) ; (4, 1) ; (4, 2) ; (4, 3) ; (4, 4) ; (4, 5) ; (5, 1) ;
(5, 2) ; (5, 3) ; (5, 4) ; (5, 5) ;
```

- Arrangements : (tirage sans remise)

```
import itertools

E = [1, 2, 3, 4]
for cas in itertools.permutations(E,3) :
    print (cas,end=' ; ')
```

```
(1, 2, 3) ; (1, 2, 4) ; (1, 3, 2) ; (1, 3, 4) ; (1, 4, 2) ; (1, 4, 3)
(2, 1, 3) ; (2, 1, 4) ; (2, 3, 1) ; (2, 3, 4) ; (2, 4, 1) ; (2, 4, 3)
(3, 1, 2) ; (3, 1, 4) ; (3, 2, 1) ; (3, 2, 4) ; (3, 4, 1) ; (3, 4, 2)
(4, 1, 2) ; (4, 1, 3) ; (4, 2, 1) ; (4, 2, 3) ; (4, 3, 1) ; (4, 3, 2)
```

- Combinaisons : (tirages simultanés, ordre non pris en compte)

```python
import itertools

E = [1, 2, 3, 4, 5]
for cas in itertools.combinations(E,3) :
    print (cas,end=' ; ')
```

```
(1, 2, 3) ; (1, 2, 4) ; (1, 2, 5) ; (1, 3, 4) ; (1, 3, 5) ;
(1, 4, 5) ; (2, 3, 4) ; (2, 3, 5) ; (2, 4, 5) ; (3, 4, 5) ;
```

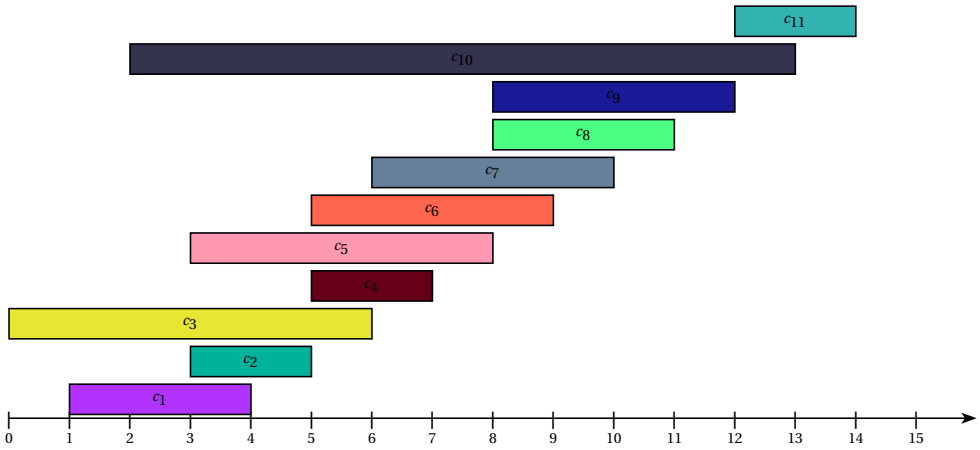
Exercice 8 A partir de ces informations, on peut se lancer dans un algorithme **brute force** pour répondre à la question. Que dire de sa complexité ?

A présent,

- On suppose comme précédemment que les concerts sont rangés par ordre croissant d'heure de fin : $f_0 \leqslant f_1 \leqslant .... \leqslant f_n$.
- On note C_{ij} l'ensemble des concerts se déroulant (intégralement) entre les heures f_i et d_j.
- Enfin, on note m_{ij} le nombre maximal de concerts auxquels on peut assister parmi ceux de C_{ij}
- Pour simplifier les problèmes aux bords, on ajoute deux concerts virtuels :
  - c_0 qui commence et finit au temps $-\infty$
  - c_{n+1} au temps $+\infty$

On cherche donc $m_{0,n+1}$

Exercice 9 On redonne le schéma,

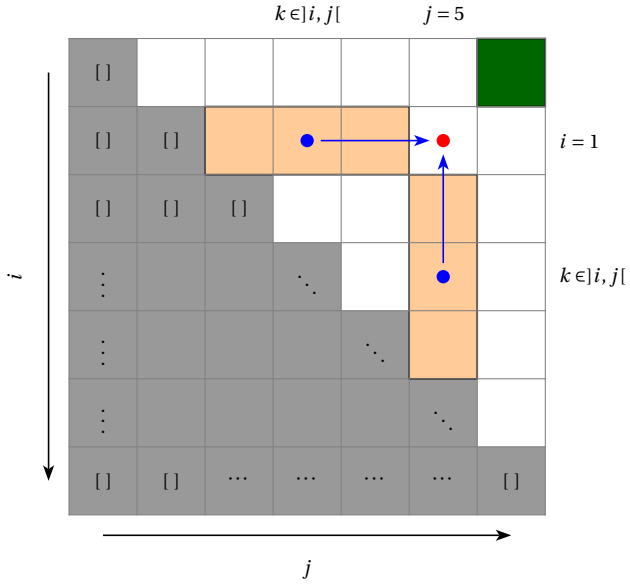


1. Que vaut S_{ij} lorsque $i \geqslant j$?
2. Si $i < j$, montrer que $m_{ij} = \begin{cases} 0 \text{ si } C_{ij} = \emptyset \\ \max\limits_{c_k \in C_{ij}} \{m_{ik} + 1 + m_{kj}\} \end{cases}$

Nous allons donc créer un tableau A de taille $(n+2)\times(n+2)$ destiné à contenir des listes de concerts de C_{ij} tels que $|A_{ij}| = m_{ij}$.

Diagramme
des dépendances

$$m_{ij} = \begin{cases} 0 \text{ si } C_{ij} = \emptyset \\ \max\limits_{c_k \in C_{ij}} \{m_{ik} + 1 + m_{kj}\} \end{cases}$$



Exercice 10 Écrire une fonctions basée sur le principe de programmation dynamique pour compléter le tableau et ainsi renvoyer la réponse souhaitée.

3 Les ensembles super-premiers

Un problème ouvert, inspiré du problème numéro 60 du site projecteuler.net/.

On dit qu'un ensemble est super-premier de taille n si :

- il contient n nombres,
- chacun de ces nombres est premier et possède strictement moins de n chiffres,
- tous les nombres fabriqués en "collant" deux nombres de cet ensemble sont aussi premiers.

Par exemple $\{3;7\}$ est un ensemble super-premier de taille 2 car 3, 7, 37 et 73 sont tous des nombres premiers et de la même manière $\{3,7,109,673\}$ est un ensemble super-premier de taille 4....

Question :

Existe-il des ensembles super-premiers de taille 3 ? 4 ? et 5 ? Si oui combien ?

Pistes de correction

Correction 1 On peut par exemple choisir de renvoyer un dictionnaire :

```
def rendu(s,) :
    pieces = [200, 100, 50, 20, 10, 5, 2, 1]
    rep = {}
    for p in pieces :
        rep[p], s = s // p, s % p
    return rep, sum(rep.values())
```

Un exemple :

```
>>> rendu(73)
({1: 1, 2: 1, 5: 0, 10: 0, 20: 1, 50: 1, 100: 0, 200: 0}, 4)
```

Correction 2

1. Proposition de Philippe PICART qui est une traduction littérale de la formule :

```
def N(S,P):
    if S==0:
        return 0
    else :
        return 1+min([N(S-p,P) for p in P if p<=S])
```

Une seconde solution plus algorithmique et peut-être plus formatrice pour les élèves :

```
def renduR(s, pieces) :
    if s == 0 :
        return 0
    nb_mini = float('inf') # On peut mettre s+1
    for p in pieces :
        if p <= s :
            nb = renduR(s-p, pieces)
            if nb < nb_mini :
                nb_mini = nb
    return 1+nb_mini
```

2. Une seconde version où l'on garde aussi la répartition des pièces :

```
def renduR2(s, pieces) :
    if s == 0 :
        return 0, []
    nb_mini = float('inf')
    piece_mini = 0
    for p in pieces :
        if p <= s :
            nb, liste = renduR2(s-p, pieces)
            if nb < nb_mini :
                nb_mini = nb
                piece_mini = p
                liste_mini = [p]+liste
    if piece_mini == 0 :    # Pas utile si pieces contient 1.
        raise ValueError("Impossible, petit malin, va !")
    return 1+nb_mini, liste_mini
```

Correction 3 On va utiliser un dictionnaire pour mémoriser les calculs déjà effectués ainsi qu'une fonction auxiliaire :

```
def renduTD(s, pieces) :
    dico = {0:0}
    def renduTDaux(s) :
        if s in dico :
            return dico[s]
        nb_mini = float('inf')
        for p in pieces :
            if p <= s :
                nb = renduTDaux(s-p)
                if nb < nb_mini :
                    nb_mini = nb
        dico[s] = 1+nb_mini
        return 1+nb_mini
    return renduTDaux(s)
```

Correction 4 Il suffit d'appliquer la formule :

```
def rendu_dyn(somme, pieces) :
    N = [ [0]*(somme+1) for i in range(len(pieces))]
    for i in range(somme+1) : N[0][i] = i
    for m in range(1, len(pieces)) :
        for s in range(somme+1) :
            if s < pieces[m] :
                N[m][s] = N[m-1][s]
            else :
                N[m][s] = min( N[m-1][s], 1+N[m][s-pieces[m]])
    return N[len(pieces)-1][somme]
```

Correction 5 Cette fois, on stocke dans les cases du tableau, le nombre de pièces et la répartition :

```
def rendu_dyn2(somme, pieces) : # Version avec le détail à rendre
    N = [ [0, []]*(somme+1) for i in range(len(pieces))]
    for i in range(somme+1) : N[0][i] = i, [1]*i
    for m in range(1, len(pieces)) :
        for s in range(somme+1) :
            if s < pieces[m] or N[m-1][s][0] < 1+N[m][s-pieces[m]][0] :
                N[m][s] = N[m-1][s]
            else :
                N[m][s] = 1+N[m][s-pieces[m]][0], [pieces[m]] + N[m][s-pieces[m]][1]
    return N[len(pieces)-1][somme]
```

Correction 6 Une solution :

```
def rendu_dyn3(somme, pieces) : # Version avec un tableau 1D
    N = [(i,[1]*i) for i in range(somme+1)]
    for m in range(1, len(pieces)) :
        for s in range(somme+1) :
            if s < pieces[m] or N[s][0] < 1+N[s-pieces[m]][0] :
                N[s] = N[s]
            else :
                N[s] = 1+N[s-pieces[m]][0], [pieces[m]] + N[s-pieces[m]][1]
    return N[somme]
```

Correction 7

1. $[d_i; f_i[\cap [d_i; f_i[= \emptyset \iff f_i \leqslant d_j$ ou $f_j \leqslant d_i$:

```
def possible2(C, i, j):
    return C[i][1] <= C[j][0] or C[j][1] <= C[i][0]
```

2. Pour savoir si on peut assister aux 3 concerts c_1, c_8 et c_5, il faut a priori les tester 2 à 2 :

```
possible2(C, 1, 8) and possible2(C, 1, 5) and possible2(C, 5, 8)
```

   Plus généralement, si on a une liste de n concerts, il y a $\binom{n}{2} = \frac{n(n-1)}{2}$ couples possibles, donc un algorithme en $\theta(n^2)$.

3. On obtient une solution linéaire (une fois les concerts classés !) :

```
def possible(C, L) : # L = liste d'indices dans l'ordre croissant
    for i in range(1, len(L)) :
        if C[L[i]][0] < C[L[i-1]][1] :
            return False
    return True
```

Correction 8 On va utiliser l'instruction **`combinations`** pour tester des sous-groupes de taille de plus en plus petite en partant de l'intégralité des concerts. Dès que l'on trouve une solution convenable, on la renvoie, stoppant ainsi la boucle.

```
def forcebrute(C) :
    indices = [i for i in range(len(C))]
    for long in range(len(C), 0, -1) :
        for choix in itertools.combinations(indices,long) :
            if possible(C, choix) :
                return(choix, long)
    return "Bizarre ton truc...."
```

En terme de complexité, s'il y a n concerts au total, il y a 2^n sous-ensembles possibles et pour chacun il faut tester si c'est possible, on a donc une solution en $O(n2^n)$ ce qui est très mauvais...

Correction 9

1. Si $i \geqslant j$, $d_j < f_j \leqslant f_i$, il n'y a donc aucun concert possible : $m_{ij} = 0$.
2. Si il n'y a aucun concert dans S_{ij}, alors $m_{ij} = 0$. Dans les autres cas :

$$m_{ij} = \max_{c_k \in C_{ij}} \{ \underbrace{m_{ik}}_{\text{avant } c_k} + \underbrace{1}_{c_k} + \underbrace{m_{kj}}_{\text{après } c_k} \}$$

Correction 10 Il suffit de traduire la formule et de faire attention à l'ordre des calculs :

```
def dynamique(C) :
    C = [[-float('inf'),-float('inf')]] + C + [[float('inf'),float('inf')]]
    A = []
    n = len(C)
    for i in range(n) :
        A.append([[]]*n)

    for i in range(n-1,-1,-1) :      # Calculs du bas vers le haut
        for j in range(1, n) :       # colonnes de gauche à droite
            for k in range(i+1,j) : # i < k < j
                # On peut assister au concert k ?
                if C[k][0] >= C[i][1] and C[k][1] <= C[j][0] :
                    # Et c'est mieux ?
                    if len(A[i][j]) < len(A[i][k]) + 1 + len(A[k][j]) :
                        A[i][j] = A[i][k] + [k] + A[k][j]
    return A[0][-1]
```

Pour information, on obtient ce tableau à la fin :

[]	[]	[]	[]	[1]	[]	[1]	[1]	[1, 4]	[1, 4]	[]	[1, 4, 8]	**[1, 4, 8, 11]**
[]	[]	[]	[]	[]	[]	[]	[]	[4]	[4]	[]	[4, 8]	[4, 8, 11]
[]	[]	[]	[]	[]	[]	[]	[]	[4]	[4]	[]	[4, 8]	[4, 8, 11]
[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[7]	[7, 11]
[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[8]	[8, 11]
[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[8]	[8, 11]
[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[11]
[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[11]
[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[11]
[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[11]
[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]
[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]
[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]