

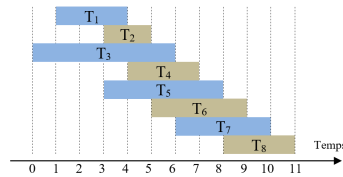
# Programmation dynamique - Ordonnancement de tâches pondérées - Enoncé

## Définition du problème - Weighted Interval Scheduling

On dispose d'une liste de tâches  $T_i$  pour  $1 \leq i \leq n$ .


Chaque tâche est définie par :

- $d_i$  l'instant de début de la tâche  $T_i$  ;
- $f_i$  l'instant de fin de la tâche  $T_i$  ;
- $v_i$  la valeur associée à la tâche  $T_i$ .



**Hypothèse** :  $f_1 \leq f_2 \leq \dots \leq f_n$  (si nécessaire, on trie la liste des tâches selon ce critère).

**Critère de compatibilité entre deux tâches** : deux tâches compatibles ne se chevauchent pas.

Objectif  : déterminer le sous-ensemble de tâches compatibles pour lequel la valeur totale est maximum.

Applications

- Planning d'activités plus ou moins valorisées en maximisant la valorisation totale (une seule activité à la fois).
- Planification de réservation d'une ressource par différents utilisateurs en maximisant la durée d'utilisation (une seule réservation à la fois).

## Bibliothèques

```
from _validation import *
from random import randint, seed
import numpy as np
import matplotlib.pyplot as plt
```

## Analyse de l'exemple - Ebauche d'un algorithme

Si on aborde le problème avec une stratégie descendante (i.e. en partant de la tâche  $T_8$ ), deux cas se présentent :

- soit  $T_8$  fait partie de la solution optimale ;
- soit  $T_8$  ne fait pas partie de la solution optimale.

Dans le premier cas, la tâche suivante à considérer est  $T_5$  ( $T_6$  et  $T_7$  sont incompatibles avec  $T_8$  car elles terminent après le début de  $T_8$ ) et on itère le procédé avec  $T_5$ .

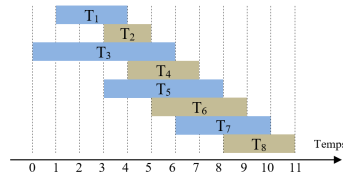
Dans le second cas, on itère ce même raisonnement avec  $T_7$  (tâche précédent  $T_8$ ).

L'analyse amène donc à définir un critère de compatibilité entre tâches.

## Définition

On note  $c(i)$  le nombre de tâches compatibles avec la tâche  $T_i$ .

$c(i)$  est donc le nombre de tâches se terminant avant que la tâche  $i$  ne commence ou encore  $c(i)$  est le plus grand index  $k < i$  tel que la tâche  $f_k \leq d_i$ .



Sur l'énoncé papier de ce TP, remplir la ligne  $c(i)$  dans le tableau ci-dessous (correspondant à l'exemple ci-dessus) en commençant par  $i = 8$  conformément à la stratégie descendante.

$i$	1	2	3	4	5	6	7	8
$c(i)$								

**Fonction  $c(i)$  : nombre de tâches compatibles avec la tâche  $T_i$**

Compléter le code de la fonction  $c(i)$  (lignes 10, 13, 14 et 15) en considérant qu'une variable globale nommée `taches` a été définie sous la forme d'une liste de tuples de la forme :  
`taches = [(d1, f1, v1), (d2, f2, v2), ..., (dn, fn, vn)]` **supposée triée par valeurs  $f$  croissantes.**

```
def c(i):
    """
    Paramètres :
        i : tâche numéro i (int)

    Renvoie :
        k : plus grand index k < i tel que Tk soit compatible avec Ti (i.e fk <= di)
    """
    global taches
    di = 0
    # On cherche la plus grande valeur de k tq fk <= di donc boucle avec k croissant
    k = 0
    while k < i:
        # Initialisation de k
        # Facultatif
        # Début de la tâche Ti
        # Test c(i) avec exemple ci-dessus (valeurs vi fixées arbitrairement à None pour ce test)
        taches = [(1, 4, None), (3, 5, None), (0, 6, None), (4, 7, None), (3, 8, None), (5, 9, None), (6, 10, None), (7, 10, None)]
        print([c(i)+1 for i in range(len(taches))]) # c(i)+1 pour correspondre avec la numérotation T1 à T8 (0 <=
```

## Génération aléatoire des tâches et tri

On cherche à construire une liste de tâches (liste de tuples) de la forme :  
`taches = [(0, 0, 0), (d1, f1, v1), (d2, f2, v2), ..., (dn, fn, vn)]` (notations définies ci-dessus).  
 avec  $d_i < f_i$  entiers (compris entre 0 et 24).


### Tâches aléatoires

randint

Cf. TP Sac à dos module random

*# Penser à l'aide en ligne*

help(randint)

 Compléter le code (lignes 5, 8, 11, 13 et 15).

```
def tirage(n, vmax):
    taches = [(0, 0, 0)]      # Initialisation
    # Tirages aléatoires pour  $T_i$  :  $d_i$ ,  $f_i$  et  $v_i$ 
    for i in range(n):
        di, fi =
        # En cas d'égalité  $d_i = f_i$ , on renouvelle le tirage de  $f_i$ 
        while di == fi:
            fi =
        # On doit avoir  $f_i > d_i$ 
        if di > fi: # Ne pas renouveler de tirage, solution simple !
            di, fi =
        # Valeur de la tâche  $T_i$  avec  $1 \leq v_i \leq vmax$ 
        vi =
        # Une tâche  $T_i$  est définie comme un tuple  $T_i = (d_i, f_i, v_i)$ 
        taches.append(
        )
    return taches

# Test
n = 8      # Nombre de tâches
vmax = 5   # Valeur maximum pour une tâche
taches = tirage(n, vmax)
print(taches)
```

### Tri des tâches par instant final croissant

 L.sort et sorted(L)

Cf. Tp Sac à dos

*# Penser à l'aide en ligne*

`help(list.sort)` # `help(sort)` provoque un message d'erreur car `sort` est une méthode => préciser `list.sort`

`help(sorted)` # Le nom de la fonction suffit sans autre précision

 lambda

Cf. Tp Sac à dos

On souhaite que la liste des tâches soit triée par instant final  $f_i$  croissant.

 Compléter le code (ligne 2).

```
# Fonction renvoyant l'instant final correspondant à une tâche  $T_i$ 
instant_final = lambda Ti:

# Tri de la liste taches en prenant comme critère de tri l'instant final
taches.sort(key=instant_final) # Tri EN PLACE (i.e. la liste est MODIFIEE)

# Test
print(taches)
```

# 1 Algorithme : récursivité descendante avec choix binaire

**Rappel objectif** : déterminer le sous-ensemble de tâches compatibles pour lequel la valeur totale est maximum.

Cette valeur maximale ou *optimale* recherchée est notée  $opt(n)$  où  $n$  est le nombre total de tâches.

## Notation

On note  $opt(i)$  la valeur optimale (i.e. maximum ici) du problème restreint aux tâches  $T_1$  à  $T_i$ .

On considère le sous-problème restreint aux tâches  $T_1$  à  $T_i$ .

Deux cas se présentent pour le calcul de  $opt(i)$ .

### 1er cas : la valeur optimale est obtenue avec la tâche i

⇒ les tâches  $n^{\circ}c(i+1)$ ,  $c(i+2)$ , sont incompatibles (par définition,  $c(i)$  est l'index de la dernière tâche compatible avec  $T_i$ ).


⇒ la valeur optimale est donc  $v_i$  + valeur optimale pour les tâches compatibles précédentes, c'est-à-dire  $v_i + opt(c(i))$  où  $c(i) < i$  est le nombre de tâches compatibles avec la tâche  $i$ .

### 2nd cas : la valeur optimale est obtenue sans la tâche i

⇒ la valeur optimale doit donc être recherchée pour l'ensemble de toutes les tâches précédentes (autrement dit, c'est la solution optimale du sous-problème restreint aux tâches  $T_1$  à  $T_{i-1}$ ); cette valeur est  $opt(i-1)$ .

Finalement :  $opt(i) = \begin{cases} 0 & \text{si } i = 0 \\ \max(v_i + opt(c(i)), opt(i-1)) & \text{sinon} \end{cases}$

## 1.1 Fonction $opt(i)$ « naïve » récursive

 Ecrire la fonction  $opt(i)$  à partir de cet algorithme

$$opt(i) = \begin{cases} 0 & \text{si } i = 0 \\ \max(v_i + opt(c(i)), opt(i-1)) & \text{sinon} \end{cases}$$

```
def opt(i):
```

```
    """
```

```
    Paramètres :
```

```
        i : tâche numéro i (int)
```

```
    Renvoie :
```

```
        la plus grande valeur pour la somme des  $v_i$  pour les tâches compatibles
```

```
    """
```

```
    global taches
```

```
# Test opt(i)
```

```
n = 4
```

```
vmax = 3
```

```
taches = tirage(n, vmax)
```

```
taches.sort(key=instant_final) # Tri EN PLACE (i.e. la liste est MODIFIEE)
```

```
print(taches)
```

```
print(opt(n))
```

  Arborescence des appels - Exécuter et observer.

```
from rcviz import viz
```

```
# Pour information, le symbole @ fait référence à un décorateur (hors programme)
```

```
@viz
```

```
def opt(i):
```

```
    global taches
```

```
    if i == 0: return 0
```

```
    return max(taches[i][2] + opt(c(i)), opt(i-1))
```

```
taches = [(0, 0, 0), (3, 6, 3), (14, 18, 1), (12, 20, 1), (9, 22, 1)]
```

```
opt(n)
```

```
# Le décorateur nommé "viz" permet de modifier la fonction opt (ici en lui ajoutant la méthode callgraph())
```

```
opt.callgraph()
```

## Questions

1. Cet algorithme possède-t-il la propriété de sous-structure optimale?
2. Les sous-problèmes se chevauchent-ils?

## # Réponses

"""

1/


2/

"""

# Valider cette cellule

```
def opt(i): # Redéfinie pour supprimer le décorateur qui permettait de visualiser les appels
    global taches
    if i == 0: return 0
    return max(taches[i][2] + opt(c(i)), opt(i-1))
```

## 2 fonction `opt_mem(i)` récursive descendante utilisant la mémorisation

 Ecrire une fonction `opt_mem(i)` récursive descendante utilisant la mémorisation.

Rappel de la stratégie :

- on utilise un dictionnaire de la forme  $\{j : opt(j)\}$  avec  $0 \leq j \leq i$  pour stocker les valeurs calculées pour chaque tâche  $T_i$ ;
- avant de calculer un terme, on vérifie qu'il n'existe pas déjà dans le dictionnaire;
- à chaque nouveau calcul, la résultat est stocké dans le dictionnaire.

Remarque : cette fonction renvoie la valeur maximum correspondant à une sélection de tâches parmi toutes les tâches mais ne fournit pas la sélection en question.

$$opt(i) = \begin{cases} 0 & \text{si } i = 0 \\ \max(v_i + opt(c(i)), opt(i-1)) & \text{sinon} \end{cases}$$

```
def opt_mem(i): # Fonction "d'encapsulation"
    """
    Paramètres :
        i : tâche numéro i (int)

    Renvoie :
        la plus grande valeur pour la somme des v_i pour les tâches compatibles
    """
    memo = # Dictionnaire pour la fonction opt_mem_aux
    def opt_mem_aux(i):
        global taches
        # Si le resultat cherché est déjà présent, on utilise le dictionnaire
        if
            return
        # Sinon, on fait le calcul et on mémorise la valeur dans le dictionnaire
        memo[i] =
        return
    return opt_mem_aux(i)

# Test opt_mem

taches = [(0, 0, 0), (17, 19, 7), (5, 17, 1), (20, 24, 4), (11, 22, 2), (12, 18, 4)]
taches.sort(key=instant_final) # Tri EN PLACE (i.e. la liste est MODIFIEE)
n = len(taches) - 1
print(taches)

print(f'Fonction "naïve", vmax = {opt(n)}')
print(f'Mémorisation, vmax : {opt_mem(n)}')
```

### 3 Fonction `opt_asc(i)` itérative ascendante

 Proposer une fonction `opt_asc(i)` utilisant un algorithme itératif ascendant avec stockage des valeurs dans un dictionnaire.

Rappel de la stratégie : la fonction remplit un dictionnaire de la forme  $\{j : opt(j)\}$ .

```
def opt_asc(i):
    """
    Paramètres :
        i : tâche numéro i (int)

    Renvoie :
        la plus grande valeur pour la somme des vi pour les tâches compatibles
    """
    # Initialisation du dictionnaire
    memo = {}
    # Construction ascendante du dictionnaire
    for k in range(i):
        memo[k] = opt_asc(k)
    # Dernière valeur = valeur cherchée
    return memo[i]

# Test opt_asc

taches = [(0, 0, 0), (17, 19, 7), (5, 17, 1), (20, 24, 4), (11, 22, 2), (12, 18, 4)]
taches.sort(key=instant_final) # Tri EN PLACE (i.e. la liste est MODIFIEE)
n = len(taches) - 1
print(taches)

print(f'Fonction "naïve", vmax = {opt(n)}')
print(f'Mémoïsation, vmax : {opt_mem(n)}')
print(f'Itérative, vmax : {opt_asc(n)}')
```

**Remarque :** ces algorithmes permettent d'obtenir la valeur maximale mais pas les tâches sélectionnées pour atteindre cette valeur.

### 4 Tâches sélectionnées


La fonction `taches_selectionnees` détermine l'ordonnancement des tâches à effectuer, elle renvoie la liste des index des tâches choisies dans la liste des tâches, par ordre croissant.

La première partie de l'algorithme repose sur celui de la fonction `opt_asc` dans lequel on mémorise dans second un dictionnaire, nommé `dico_choix`, la décision concernant la tâche  $k$  (optimise la valeur ou non) : `dico_choix[k] = True` ou `False`.

On pourrait penser qu'il suffit de sélectionner les tâches associées à la valeur `True` dans le dictionnaire mais ce raisonnement est incorrect car on ne sait pas encore à l'étape  $i$  si une étape ultérieure ne conduit pas à une valeur supérieure (autrement dit, une tâche sélectionnée à l'étape  $i$  peut être abandonnée au cours des évaluations ultérieures si elle devient moins intéressante).

Dans la seconde partie, l'algorithme parcourt donc le tableau à partir de la fin, en sautant de choix optimal en choix optimal.

$$opt(i) = \begin{cases} 0 & \text{si } i = 0 \\ \max(v_i + opt(c(i)), opt(i-1)) & \text{sinon} \end{cases}$$

 Compléter les lignes 16, 17, 19, 20, 24, 25, 28 et 30 du script ci-dessous.

```
def taches_selectionnees(i):
    """
    Paramètres :
        i : tâche numéro i (int)

    Renvoie :
        L : liste des index des tâches choisies dans la liste des tâches, par ordre croissant
    """
```

```

# ----- Première partie (basée sur opt_asc)
memo = {0:0}      # Dictionnaire pour le calcul de la valeur optimale
dico_choix = {}    # Dictionnaire pour la mémorisation des décisions
# Remplissage des dictionnaires
for k in range(1, i+1):
    s = taches[k][2] + memo[c(k)]
    if s > memo[k-1]:      # La tâche k est sélectionnée
        memo[k] =
        dico_choix[k] =
    else:                  # La tâche k n'est pas sélectionnée
        memo[k] =
        dico_choix[k] =

# ----- Seconde partie (parcours des choix optimaux à partir du dernier)
L = []               # Initialisation de la liste des tâches optimales
k =                 # Initialisation boucle while : index de la dernière tâche optimale
while k > 0:         # Parcours du dictionnaire des choix
    if dico_choix[k] == : # La tâche k est sélectionnée
        L.append(k)      # Ajout de la à la liste des tâches choisies
        k =             # SAUT à la tâche compatible précédente
    else:             # La tâche k n'est pas sélectionnée
        k               # On consulte la tâche précédente
L.reverse()         # Car la liste L est remplie de la dernière tâche à la première
return L

```

## Remarques

1. Il est possible de n'utiliser qu'un seul dictionnaire de la forme {k : (opt(k), True ou False)}.
2. Pour éviter L.reverse(), au lieu de L.append(k) faire L = [k] + L (ajout par l'avant).
3. Pour comprendre le fonctionnement, il peut être intéressant de visualiser dico\_choix :
  - soit instruction print(dico\_choix) juste avant return ;
  - soit return L, dico\_choix.

```

# Test taches_selectionnees

n = 10
vmax = 4
#seed(2)
taches = tirage(n, vmax)
taches.sort(key=instant_final) # Tri EN PLACE (i.e. la liste est MODIFIEE)

vmax = opt_asc(n)
L = taches_selectionnees(n)

print(taches)
print(f'Itérative, vmax : {vmax}')
print("Index des tâches : ", L)
print("Tâches : ", [taches[i] for i in L])

```


## 5 Tests - Représentation graphique des résultats

 Exécuter le code suivant afin de pouvoir effectuer les représentations graphiques.

```

def graphe(n, vmax):
    ...

```

 Après la première exécution, modifier la valeur de la graine (seed(valeur)) du générateur pseudo-aléatoire afin pour obtenir des exécutions différentes.

```

seed(2)
n = 10      # Nombre de tâches
vmax = 4    # Valeur maximum pour une tâche

graphe(n, vmax)

```