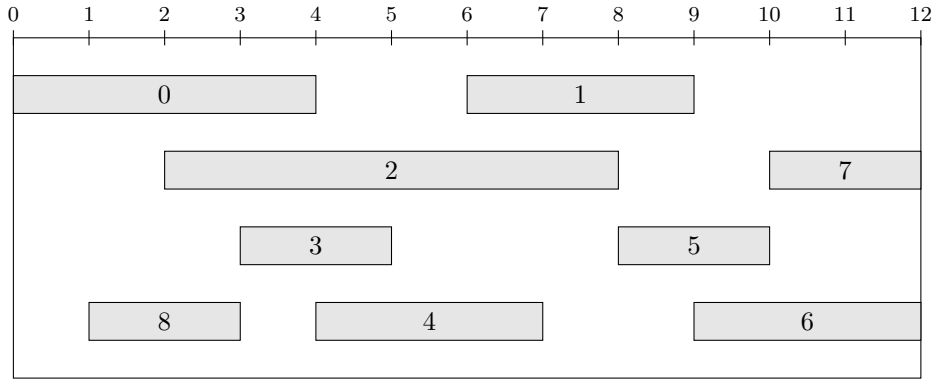
Corrigé du DS 2 informatique commune

On propose une salle pour pratiquer différentes activités, mais la salle ne peut être occupée que pour une activité à la fois. Des demandes sont faites sous la forme d'un intervalle de temps $]d_i; f_i[$ où d_i et f_i désignent l'heure de début et de fin de l'activité pour demande i.
On modélise cette donnée par une liste de couples : `[(d0, f0), (d1, f1), ... ]`.

Par exemple les demandes suivantes :



sont représentées par la liste :
`[(0, 4), (6, 9), (2, 8), (3, 5), (4, 7), (8, 10), (9, 12), (10, 12), (1, 3)]`

Une solution optimale, c'est à dire qui permet un nombre maximal d'activités, dans ce cas est de choisir les activités : 0, 4, 5, 7 représentées par : `[(0, 4), (4, 7), (8, 10), (10, 12)]`.

Force brute

1. Une combinaison est une partie de l'ensemble des n demandes,

il y a 2^n combinaisons possibles.

(je compte ici la combinaison vide).
Un algorithme qui testerait chaque combinaison serait donc de complexité au moins exponentielle, ce qui n'est pas envisageable : trop lent.

2. L'ensemble des combinaisons compatibles est une partie de l'ensemble des demandes, c'est donc un ensemble fini et non vide (la combinaison vide est compatible).
L'ensemble dont les éléments sont les cardinaux de ces combinaisons compatibles est donc une partie finie de $\mathbb{N}$, il admet donc un maximum : il existe une combinaison dont le nombre d'activité est maximal, c'est à dire qu'il existe une solution optimale.
Il n'y a pas unicité des solutions optimales, par exemple pour les demandes : `[[0, 3], [1, 4]]`, les deux demandes sont incompatibles, il y a 2 solutions optimales constituées d'une seule activité : `[[0, 3]]` et `[[1, 4]]`.

Il y a toujours une solution optimale, mais elle n'est pas toujours unique.

Pré-traitement

3. Recopier et compléter la fonction suivante qui modifie la liste de couples passée en argument pour la trier dans l'ordre croissant suivant la valeur du deuxième coefficient du couple.
Par exemple, pour la liste : `demandes = [(0, 4), (6, 9), (2, 8), (3, 5), (4, 7), (8, 10), (9, 12), (10, 12), (1, 3)]`,
après exécution de

```
tri_selection(demandes)
```

la variable `demandes` sera associée à la liste :
`[(1, 3), (0, 4), (3, 5), (4, 7), (2, 8), (6, 9), (8, 10), (10, 12), (9, 12)]`.

```
def tri_selection(L):
    n = len(L)
    for i in range(n - 1):
        min = i
        for j in range(i, n):
            if L[j][1] < L[min][1]:
                min = j
        L[i], L[min] = L[min], L[i]
    return None
```

4. Les opérations élémentaires sont : `len`, les affectations, opérations algébriques sur les nombres, lecture dans une liste.
Il y a $n-1$ tours de la boucle sur i au $i^{\text{ième}}$ tour $n-i$ tours de boucle sur la boucle sur j. La complexité est :

$$1+\sum_{i=0}^{n-2}\left(4+\sum_{j=i}^{n-1}6\right)=O\left(n^2\right)$$

la complexité de la fonction `tri_selection` est quadratique en la taille n de la liste dans tous les cas.

5. Le tri fusion permet de trier une liste avec une complexité quasi-linéaire ($O(n\ln n)$) dans tous les cas.
attention : le tri rapide est de complexité quasi-linéaire en moyenne, mais quadratique dans le pire des cas.

Algorithme glouton

On considère les trois heuristiques suivantes sur lesquelles on peut espérer baser un algorithme glouton pour résoudre le problème d'attribution de salle :

H1 : on commence par choisir une activité de durée minimale.

H2 : on commence par choisir une activité qui commence le plus tôt.

H3 : on commence par choisir une activité qui finit le plus tôt.

6. En appliquant l'heuristique H1 à la liste de demandes : `[[0, 5], [4, 6], [5, 10]]` *(faire un dessin)* on obtient : `[[4, 6]]` *(la première activité choisie est incompatible avec les autres)* qui n'est pas optimale, puisqu'on peut choisir 2 activités : `[[0, 5], [5, 10]]`

De même en appliquant l'heuristique H2 à la liste de demandes : `[[0, 10], [1, 2], [2, 3]]`, on obtient `[[0, 10]]` une seule activité alors qu'on peut choisir les deux autres : `[[1, 2], [2, 3]]`.

> les heuristiques H1 et H2 ne permettent pas de construire un algorithme glouton qui fournit dans tous les cas une solution optimale.

7. Soit `a0` une activité qui finit le plus tôt.
D'après la première question, on sait qu'il existe une solution optimale : `[a1, a2, ..., a_k]` où les activités sont dans l'odre. On distingue 2 cas :

cas 1 : `a0 = a1`, et il existe bien une solution optimale qui contient `a0` ;

cas 2 : `a0` $\neq$ `a1`. Par définition de `a0`, elle finit plus tôt que `a1` (*au sens large*) et `a1` est compatible avec `a2`, donc la fin de `a0` est avant le début de `a2` et `[a0, a2, a3, ..., ak]` est compatible, c'est donc une solution optimale qui contient `a0`.

> Dans tous les cas, si une activité finit le plus tôt, alors il existe une solution optimale qui la contient.

8. On propose l'algorithme glouton suivant : à partir de la liste triée obtenue dans la partie Pré-traitement, on choisit la première activité, puis on parcourt la liste en ajoutant une activité à la solution si elle est compatible avec celles déjà présentes.
Quelle solution obtient-on alors pour l'exemple proposé ?
C'est à dire pour la liste :
`[(1, 3), (0, 4), (3, 5), (4, 7), (2, 8), (6, 9), (8, 10), (10, 12), (9, 12)]`

> Le résultat est : `[(1, 3), (3, 5), (6, 9), (10, 12)]`

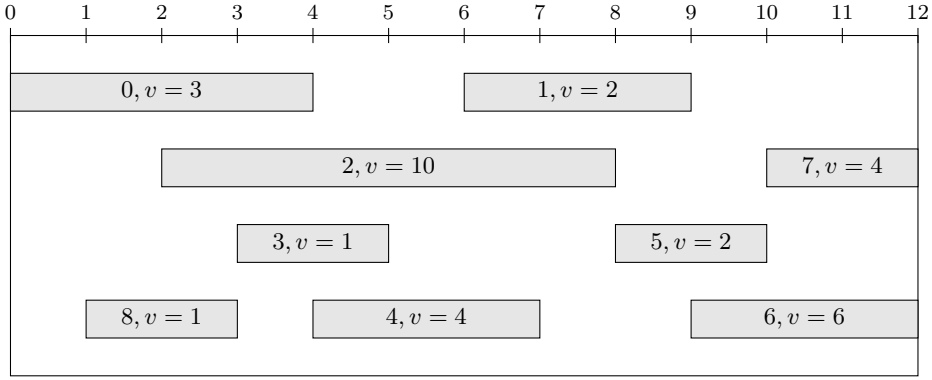
9. Écrire une fonction `solution_glouton` de paramètre une liste de couples triée suivant le deuxième coefficient du couple et qui renvoie une solution sous la forme d'une liste de couples.

```
def solution_glouton(L):
    res = [L[0]]
    fin = L[0][1]
    for k in range(1, len(L)):
        if L[k][0] >= fin:
            res.append(L[k])
            fin = L[k][1]
    return res
```

10. On considère les mêmes opérations élémentaires que précédemment ainsi que `append`.
Soit n la taille de la liste considérée. Il y a $n-1$ tours de boucles et au plus 8 opérations par tours.
concla complexité de `solution_glouton` est linéaire (dans tous les cas).

Programmation dynamique

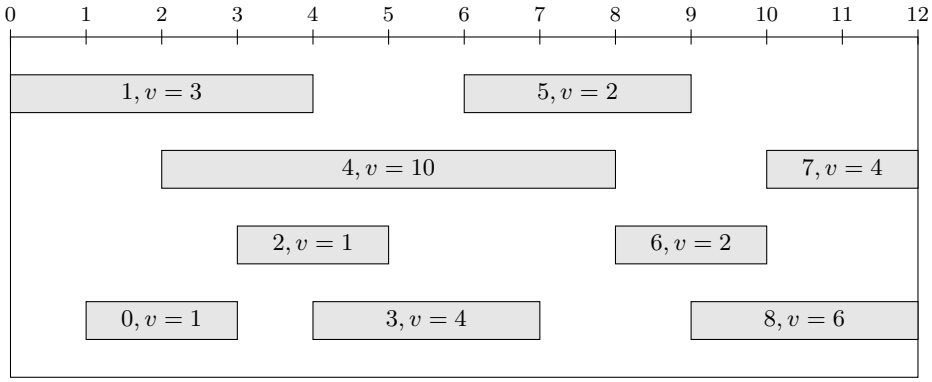
On souhaite ajouter une valeur à chaque demande. Une seule demande peut alors avoir plus de valeur que toutes les autres réunies. On considère par exemple les demandes :



représentées par la liste de triplets `(début, fin, valeur)` :
`[(0, 4, 3), (6, 9, 2), (2, 8, 10), (3, 5, 1), (4, 7, 4), (8, 10, 2), (9, 12, 6), (10, 12, 4), (1, 3, 1)]`

Une solution optimale est alors de choisir les activités : 2, 6 qui a une valeur de 16 ; alors que le choix de : 0, 4, 5, 7 a une valeur de 13.

11. La fonction écrite à la question 3 nous trie selon la valeur du deuxième élément qui est bien présent pour un tuple de longueur 3. Ainsi la fonction de tri de la question 3 nous permet également de trier les listes de triplets suivant la valeur du deuxième coefficient.
On obtient ainsi la liste :
`demandes_pond = [(1, 3, 1), (0, 4, 3), (3, 5, 1), (4, 7, 4), (2, 8, 10), (6, 9, 2), (8, 10, 2), (10, 12, 4), (9, 12, 6)]`
Ce qui correspond à :



12. Écrire une fonction `dernier_avant(L, k)` où `L` est une liste triée de demandes pondérées et `k` est un indice ; et qui renvoie l'indice `j` de la dernière activité de la liste `L` qui finit avant

le début de l'activité `k` (c'est à dire celle qui est représenté dans la liste en position `k`).
Par exemple `dernier_avant(demandes_pond, 8)` renvoie `5` car `demandes_pond[8] = (9, 12, 6)` et la dernière activité qui finit avant le temps `9` est celle en position `5`, c'est à dire `(6, 9, 2)`
et `dernier_avant(demandes_pond, 1)` renvoie `-1` car aucune activité ne finit avant le temps `0`.

il y a de nombreuses façon de faire, en voici 3.

```
def dernier_avant(L, k):
    res = -1
    for i in range(k):
        if L[i][1] <= L[k][0]:
            res = i
    return res


def dernier_avant2(L, k):
    for i in range(k + 1):
        if L[i][1] > L[k][0]:
            return i - 1


def dernier_avant3(L, k):
    deb = L[k][0]
    for i in range(len(L)-1, -1, -1):
        if L[i][1] <= deb:
            return i
    return -1
```

13. On considère une liste ainsi triée et on note $valmax(k)$ la valeur d'une solution optimale si on considère les éléments de la liste des demandes pondérées jusqu'à celui d'indice k ($k = -1$ correspond à ne prendre aucun élément).

$valmax(-1)$ correspond à ne prendre aucune activité, le maximum d'activités compatible est alors 0.

Soit $k \in [\![0; n-1]\!]$, d'après la question 1, on sait qu'il existe une solution optimale S.

- Si l'activité k n'est pas dans S, alors S est une solution optimale en considérant les $(k-1)$ premières activités, dans ce cas $valmax(k) = valmax(k-1)$;
- Si l'activité k est dans S, alors les autres activités finissent toutes avant le début de l'activité k, c'est à dire que leur rang dans la liste est inférieur à $j = dernier\_avant(k)$ et elles constituent une solution optimale pour les j premières activités. Dans ce cas $valmax(k) = v_k + valmax(dernier\_avant(k))$.

Par optimalité du choix, on a donc :

$$valmax(k) = \max\Big(v_k + valmax(dernier\_avant(k)), valmax(k-1)\Big)$$

14. Écrire une fonction récursive `val_max_rec(L, k)` qui renvoie $valmax(k)$ pour les activités représentées dans la liste `L` supposée triée.

```
def val_max_rec(L, k):
    if k == -1:
        return 0
    else:
        return max(L[k][2] +
                   val_max_rec(L, dernier_avant(L, k)),
                   val_max_rec(L, k - 1))
```

On note $C(k)$ la complexité. La complexité de `dernier_avant(L, k)` est linéaire en k (k tours de boucle et un nombre borné d'opérations à chaque tour).
Ainsi, pour $kge1, C(k) = C(k-1) + C(j) + Mk + N$ avec j la valeur renvoyée par `dernier_avant(L, k)` et M, N des constantes.
Dans le pire des cas $j = k - 1$ (toutes les activités sont compatibles).
Dans le pire des cas : $C(k) = 2C(k-1) + MK + N$, donc

dans le pire des cas, la complexité de la fonction récursive est exponentielle.

15. Écrire une fonction `val_max_mem(L)` utilisant la mémoïsation avec un dictionnaire (programmation dynamique) qui renvoie la valeur d'une solution optimale.

```
def val_max_mem(L):
    mem = {-1: 0}

    def inter(k):
        if k in mem:
            return mem[k]
        else:
            new = max(L[k][2] + inter(dernier_avant(L, k)),
                      inter(k - 1))
            mem[k] = new
            return new
    return inter(len(L) - 1)
```

Dans le pire des cas on remplit $n+1$ valeurs dans le dictionnaire, donc au plus n valeurs sont ajoutés avec une complexité majorée par $M \times n$ avec M une constante. Les autres appels récursifs sont alors de complexité bornée et en nombre majoré par 2 fois le nombre d'appels sur de nouvelles valeurs.
La complexité est donc dans le pire des cas :

$$M \times n^2 + 2n$$

Donc :

la complexité de la fonction `val_max_mem` est quadratique.

16. (bonus) Écrire une fonction qui renvoie une solution optimale sous la forme d'une liste de triplets.

```
def dico_mem(L):
    mem = {-1: 0}

    def inter(k):
        if k in mem:
            return mem[k]
        else:
            new = max(L[k][2] + inter(dernier_avant(L, k)),
                      inter(k - 1))
            mem[k] = new
            return new
    inter(len(L) - 1)
    return mem


def solution_mem(L):
    mem = dico_mem(L)
    res = []
    k = len(L) - 1
    while k >= 0:
        if mem[k] > mem[k - 1]:  # on teste si l'activité k est
            choisie
            res.append(L[k])
            k = dernier_avant(L, k)
        else:
            k = k - 1
    return res
```

17. Écrire une version itérative (approche ascendante ou bottom-up) de la fonction `val_max_mem(L)`.

```
def val_max_iter(L):
    mem = {-1: 0}
    for k in range(len(L)):
        mem[k] = max(L[k][2] + mem[dernier_avant(L, k)],
                     mem[k - 1])
    return mem[len(L) - 1]
```