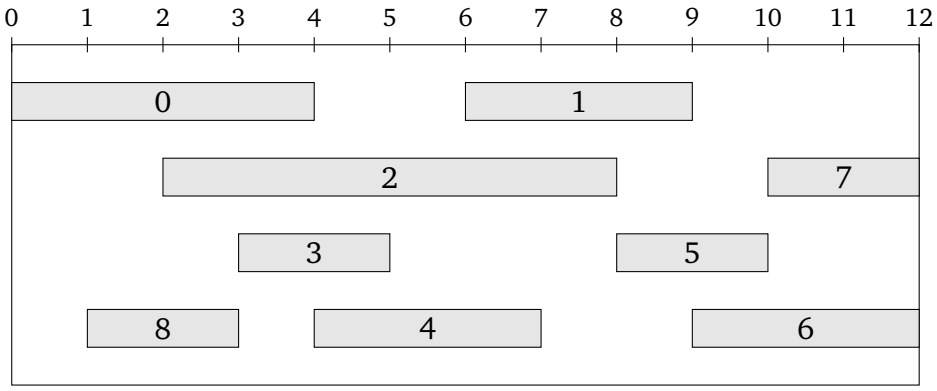
Devoir Surveillé n° 2.
le 14 novembre, 2h.
Attribution de salle

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. Si l'on veut tester toutes les combinaisons de demandes (compatibles ou non), combien y a t'il de cas à considérer pour une liste de n demandes ? Un algorithme basé sur ce principe est-il envisageable ?
2. Montrer qu'il existe toujours une solution optimale, est-elle 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


    return None
```

4. Quelle est sa complexité dans le pire des cas pour une liste de taille n ?
5. Quel autre algorithme de tri peut on utiliser pour améliorer la complexité 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.

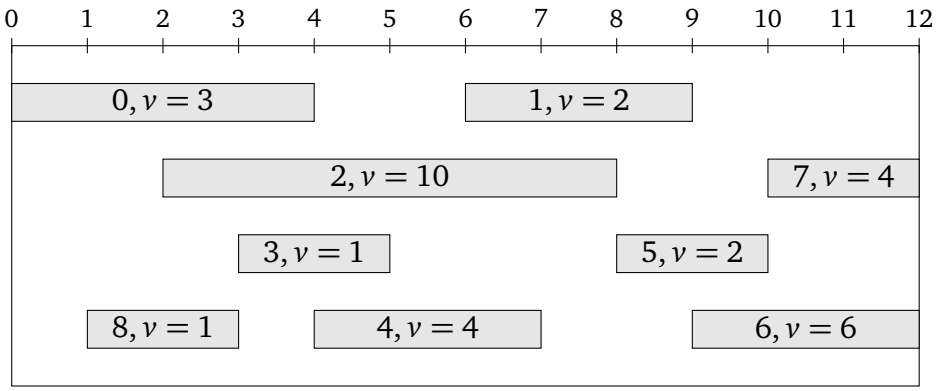
H2 : on commence par choisir une activité qui commence le plus tôt.

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

6. Montrer que les heuristiques H1 et H2 ne permettent pas de construire un algorithme glouton qui fournit dans tous les cas une solution optimale.
7. Dans tous les cas, montrer que 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)]`
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.
10. Quelle est la complexité de cette fonction ?

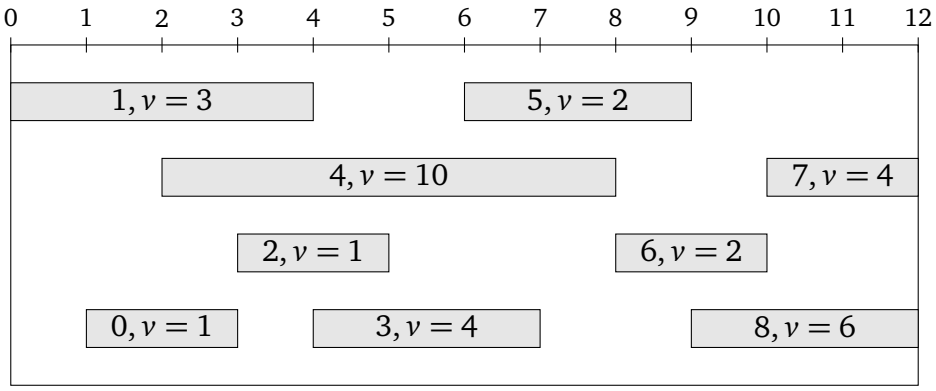
Programmation dynamique

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. Justifier que 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`.
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).
Montrer que : $valmax(-1) = 0$ et pour tout $k \in [\![0; n-1]\!]$:

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

14. Écrire une fonction récursive `val_max_rec(L, k)` qui renvoie $valmax(k)$ pour les activités représentées dans la liste `L` supposée triée. Quelle est sa complexité ?
15. Écrire une fonction `val_max_mem(L)` utilisant la mémoïsation avec un dictionnaire (programmation dynamique) qui renvoie la valeur d'une solution optimale. Quelle est sa complexité ?
16. Écrire une fonction qui renvoie une solution optimale sous la forme d'une liste de triplets.
17. Écrire une version itérative (approche ascendante ou bottom-up) de la fonction `val_max_mem(L)`.