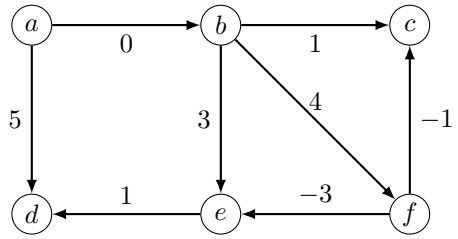
Exercice 1. Exercice de programmation
Écrire, dans votre langage de programmation préféré, une fonction pour résoudre le problème du sac à dos, par programmation dynamique.
À rendre au plus tard mercredi 6 octobre sur https://github.com/fortierq/oc-m1-2021 (instructions détaillées en bas de cette page).

Exercice 2. Bellman-Ford

1. Appliquer l'algorithme de Bellman-Ford pour trouver les distances depuis a vers n'importe quel autre sommet :



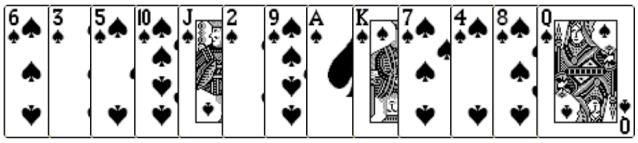
2. Comment modifier légèrement l'algorithme de programmation dynamique pour trouver les plus courts chemins, en plus des distances ?

Exercice 3. Jeu du solitaire et LIS plus rapide
Dans le jeu du solitaire (*patience*), on reçoit des cartes unes par unes. Chaque carte doit être mise :

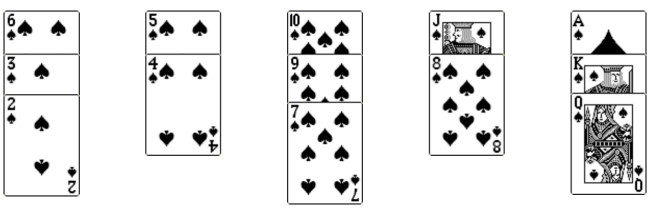
- Soit sur une nouvelle pile
- Soit sur une pile existante, si la valeur de la carte est plus petite que la carte au dessus de la pile

L'objectif est de minimiser le nombre total de piles utilisées.
On considère un algorithme glouton où chaque nouvelle carte est ajoutée sur la pile la plus à gauche possible. Si ce n'est pas possible, on créé une nouvelle pile que l'on met tout à droite.

Par exemple, en recevant ces cartes, de gauche à droite :



On obtient les piles suivantes avec l'algorithme glouton :



1. Que peut-on dire des valeurs des cartes au dessus de chaque pile ?

On peut modéliser ce problème avec un tableau T correspondant aux valeurs des cartes, dans l'ordre ($T[0]$ est la première carte reçue, $T[1]$ la deuxième...). Soit m le nombre minimum de piles nécessaires pour un jeu de solitaire avec les cartes de T.

2. Écrire en Python l'algorithme glouton. On utilisera une liste `P` pour stocker la valeur de la carte du dessus de chaque pile. Par exemple, `P[0]` sera la valeur au dessus de la 1ère pile.

   Solution :

```python
from bisect import bisect_left

def patience(T):
    P = []
    for e in T:
        i = bisect_left(P, e)
        if i == len(P):
            P.append(e)  # créer une pile
        else:
            P[i] = e
    return len(P)
```

3. Quelle est la complexité de votre algorithme précédent ? Si ce n'est pas le cas, expliquer comment obtenir une complexité $O(n \log(n))$, où n est la taille de P.

   Solution : La boucle for exécute n fois `bisect_left` qui est une recherche par dichotomie en $O(\ln(n))$. D'om une complexité $\boxed{O(n \ln(n))}$.

4. Donner une LIS (*Longest Increasing Subsequence*) de T, pour l'exemple avec les cartes au début de l'exercice.

   Solution : 3, 5, 7, 8, 12 (dame), par exemple

5. Montrer que n'importe quelle sous-suite croissante de T est de longueur inférieure à m.

   Solution : On remarque que les éléments au sein d'une même pile forment une sous-suite décroissante. Une sous-suite croissante ne peut donc pas contenir 2 éléments d'une même pile, donc sa taille est au plus le nombre de piles.

6. À partir des m piles obtenues par l'algorithme glouton, montrer qu'on peut obtenir une sous-suite croissante de taille m.

Le dernier résultat montre que l'on peut trouve une LIS en $O(n \log(n))$, à partir d'une solution au jeu de solitaire par l'algorithme glouton.

Exercice 4. distance de Levenshtein
Soient S_1 et S_2 deux chaînes de caractères. La distance de Levenshtein entre S_1 et S_2 est le nombre minimum de caractères qu'il faut insérer, supprimer ou substituer pour passer de S_1 à S_2.

1. Quelle est la distance de Levenshtein entre nuit et bruit ?
2. Soit $T[i][j]$ le nombre minimum d'insertion, suppression, substitution pour passer de $S_1[:i]$ à $S_2[:j]$.
   Donner une équation de récurrence sur $T[i][j]$.
3. Écrire un pseudo-code pour trouver la distance de Levenshtein entre 2 chaînes de caractères. Quelle est sa complexité?

Exercice 5. ***Weighted interval scheduling***

On considère des intervalles I_1, ..., I_n pondérés et ordonnés par temps de fin croissante. On veut sélectionner des intervalles sans aucune intersection dont le poids total est maximum.

1. Montrer sur un exemple que l'algorithme glouton sélectionnant un intervalle s'il n'intersecte aucun des intervalles précédemment choisis n'est pas optimal.
2. Écrire un algorithme par programmation dynamique permettant de résoudre le problème.