Exercice 1. Rendu de monnaie
Étant donné un entier n (de l'argent à rendre) et des entiers $a_1, \dots, a_k$ (des pièces), on veut calculer le nombre minimum $r(n,k)$ de pièces parmi $a_1, ..., a_k$ dont la somme vaut n.
Par exemple, si $k = 3$ et $a_1 = 1, a_2 = 2, a_3 = 5$ alors $r(7,3) = 2$ (car $7 = 2 + 5$ et c'est la façon de rendre 7 qui utilise le moins de pièces).
Remarque : on peut utiliser plusieurs fois la même pièce.

1. Écrire une relation de récurrence sur $r(n,k)$.

   > **Solution** : Si

2. En déduire une fonction `int rendu(int n, int* a, int k)` en C renvoyant $r(n,k)$. On renverra -1 si ce n'est pas possible.
   Quelle est la complexité de `rendu` ?

Exercice 2. Plus grand carré dans une matrice
Étant donnée une matrice carrée remplie de 0 ou 1, on souhaite connaître la taille du plus gros carré de 1 dans cette matrice.
Par exemple, ce nombre est 2 pour la matrice M suivante (correspondant au carré en pointillé) :

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

La case de coordonnés (x, y) est celle sur la ligne x, colonne y.
La case de coordonnées (0, 0) est celle en haut à gauche.
On supposera que les indices en arguments des fonctions ne dépassent pas des tableaux ou matrices correspondants.

1. Écrire une fonction `est_carre` telle que `est_carre m x y k` détermine si la sous-matrice de `m` de taille $k \times k$ et dont la case en haut à gauche a pour coordonnées (`x`, `y`) ne possède que des 1.
   Par exemple, `est_carre M 1 2 2` doit renvoyer `true` (ce qui correspond au carré en pointillé dans la matrice M ci-dessus).

2. Écrire une fonction `contient_carre` telle que `contient_carre m k` renvoie `true` si `m` contient un carré de 1 de taille k, `false` sinon.

3. Écrire une fonction `max_carre1` telle que `max_carre1 m` renvoie la taille maximum d'un carré de 1 contenu dans `m`.

4. Quelle est la complexité de `max_carre1 m` ? Dans la suite, on utilisera une méthode plus efficace par programmation dynamique.

On va construire une matrice `c` telle que `c.(x).(y)` est la taille maximum d'un carré de 1 dans `m` dont la case en bas à droite est `m.(x).(y)` (c'est à dire un carré de 1 qui contient `m.(x).(y)` mais aucun `m.(i).(j)` si $i > x$ ou $j > y$).
Par exemple, `c.(1).(2) = 1` et `c.(2).(3) = 2` pour la matrice M ci-dessus.

5. Écrire une fonction `init` telle que `init m c` remplisse les valeurs de `c.(0).(y)` et `c.(x).(0)`, $\forall\, x, y$.

6. Que vaut `c.(x).(y)` si `m.(x).(y) = 0`?

7. Montrer que, si `m.(x).(y) = 1`, `c.(x).(y)` est égal à 1 plus le minimum de `c.(x-1).(y)`, `c.(x).(y-1)` et `c.(x-1).(y-1)`.

8. En déduire une fonction `remplir` telle que, si `m` et `c` sont des matrices de même taille, `remplir m c` modifie `c` pour qu'elle vérifie: $\forall$ `x`, `y`, `c.(x).(y)` est la taille maximum d'un carré de 1 dans `m` dont la case en bas à droite est `m.(x).(y)`.

9. En déduire une fonction `max_carre2` telle que `max_carre2 m` renvoie la taille maximum d'un carré de 1 contenu dans `m`, ainsi que les coordonnées de la case en haut à gauche d'un tel carré.

10. Quelle est la complexité de `max_carre2 m`, en fonction des dimensions de `m`? Comparer avec `max_carre1 m`.

11. Est-il possible de trouver un algorithme avec une meilleur complexité?

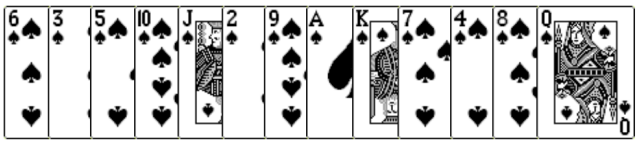
12. **Application** (Google Hash Code) : Colorier Alan Turing de la meilleure façon possible sur https://primers.xyz/0

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 :

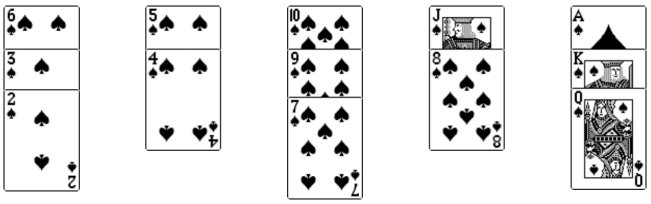
- Soit sur une nouvelle pile
- Soit sur une pile existante, si la valeur de la carte est plus petite que la carte au dessus de la pile

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 ?

   > **Solution** : Les valeurs sont triées par ordre croissant.

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 OCaml 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 :

   ```
   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'où 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.

   Solution : Au moment où on ajoute une carte dans l'algorithme glouton, on conserve un "pointeur" vers la carte du dessus de la pile précédente, de façon similaire au prédecesseurs pour Bellman-Ford. On obtient alors une LIS (à l'envers) en partant d'une carte sur la dernière pile et en remontant les pointeurs jusqu'à revenir sur la 1ère pile.

```
from bisect import bisect_left

def patience(T):
    P = []
    pred = []
    for e in T:
        i = bisect_left(P, e)
        pred[i] = P[-1]
        if i == len(P):
            P.append(e) # créer une pile
        else:
            P[i] = e
    return pred
```

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