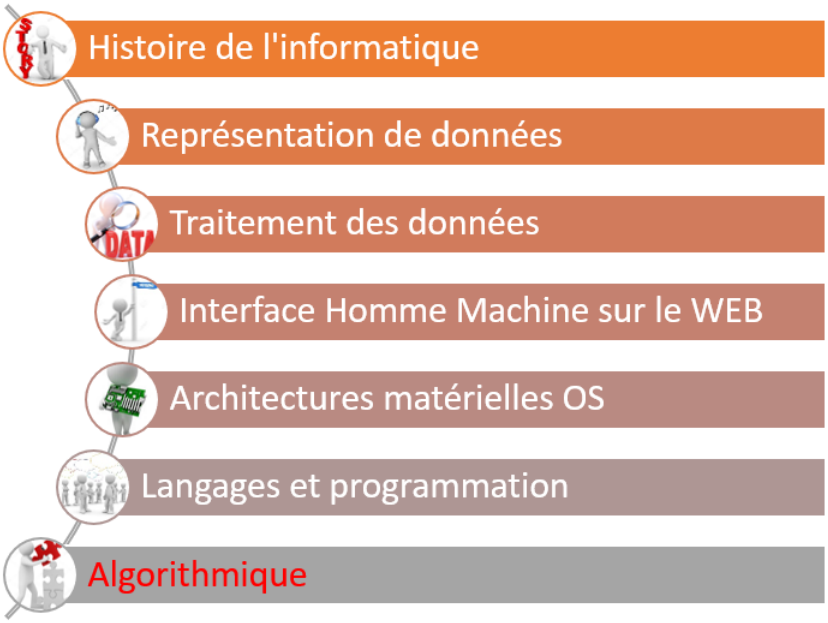

Les trois problèmes abordés dans ce chapitres , le problème du voyageur de commerce, le problème du rendu de monnaie et le problème du sac à dos sont des problèmes informatiques pour lesquels on ne dispose pas d'un algorithme permettant de les résoudre en un temps machine "réaliste".

1 Situation déclenchante : le problème du voyageur de commerce

Voici une situation concrète, un voyageur de commerce part de Nancy, et doit se rendre dans chacune des quatre villes suivantes : Metz, Paris, Reims, Troyes avant de revenir à Nancy. On souhaite trouver le trajet dont la distance à parcourir est la plus faible (il s'agit d'un problème de minimisation). On donne ci-dessous le tableau des distances (en km) séparant ces différentes villes :

	Nancy	**Metz**	**Paris**	**Reims**	**Troyes**
Nancy		55	303	188	183
Metz	55		306	176	203
Paris	303	306		142	153
Reims	188	176	142		123
Troyes	183	203	153	123	

Pour résoudre ce **problème d'optimisation** de la distance à parcourir, on peut ici énumérer tous les trajets possibles. Théoriquement il y en a 24 ; en effet on a 4 choix possibles pour la première ville visitée, puis 3 choix possibles pour la deuxième puis 2 choix possibles pour le troisième et un seul choix pour la dernière soit : $4 \times 3 \times 2 \times 1 = 24$.
Ce calcul s'appelle la factorielle de 4, noté 4!
En fait chaque itinéraire apparait deux fois : une fois dans un sens (par exemple Nancy - Metz - Paris - Reims - Troyes - Nancy) et une fois dans l'autre (Nancy - Troyes - Reims - Paris - Metz - Nancy), on doit donc évaluer la longueur de 12 trajets possibles différents.

Travail à faire :
Déterminer les 12 trajets possibles et la distance totale de chacun.
En déduire alors le trajet le plus court.

Cette technique simple au demeurant, n'est pas généralisable. En effet si l'on souhaite faire de même avec 10 villes (sans la ville de départ et d'arrivée), on aura déjà 181 440 trajets à évaluer $\left(\frac{10!}{2} = \frac{10 \times 9 \times \ldots \times 2 \times 1}{2} = 10 \times 9 \times \ldots \times 3 = 181\,440\right)$ et avec 20 villes on est à plus de un milliard de milliard de trajets possibles !

Ainsi face à de tels problèmes d'optimisation, il est impossible de déterminer **la solution optimale**, c'est à dire la solution qui minimise ou maximise le problème posé, en explorant tous les cas possibles. On peut face à ces situations mettre en place **une stratégie gloutonne**.

2 Algorithmes gloutons

On considère un problème d'optimisation tel que :

- le nombre se solutions possibles est très grand (penser au problème du voyageur de commerce)
- le problème peut se résumer à une succession de choix, produisant et précisant au fur et à mesure une solution partielle
- on dispose d'une fonction mathématique permettant d'évaluer la qualité de chaque solution (dans le cas du voyageur de commerce ; il s'agit de trouver la plus petite somme des distances parcourues)
- on cherche une solution bonne et pas nécessairement **la** solution optimale (dans le cas du voyageur de commerce un trajet de longueur "raisonnable" mais pas forcément le trajet le plus court)

2.1 Stratégie gloutonne

Face à un tel problème, un algorithme glouton est basé sur des **stratégies naturelles**, il a pour objectif d'être **simple et efficace**, et de donner une **bonne approximation de la solution optimale** du problème et parfois même la solution optimale.

Le principe général est le suivant :

- **à chaque étape** où une décision doit être prise, on fait **systématiquement le meilleur choix sur le moment**, en espérant ainsi arriver à une "bonne" solution ;
- **sans jamais remettre en cause un choix fait précédemment**.

2.2 Résolution approchée du problème du voyageur de commerce par un algorithme glouton

Appliquons une stratégie gloutonne pour proposer une solution à l'exemple initial :

- Partant de Nancy, la ville la plus proche est Metz ; le trajet partiel est :
  Nancy - Metz ; la distance alors parcourue est 55 km.
- Partant de Metz, la ville restante la plus proche est Reims à 176 km ; le trajet partiel est :
  Nancy - Metz - Reims ; la distance alors parcourue est 231 km.
- Partant de Reims, la ville restante la plus proche est Troyes à 123 km ; le trajet partiel est :
  Nancy - Metz - Reims - Troyes ; la distance alors parcourue est 354 km.
- Partant de Troyes, la ville restante la plus proche est Paris à 153 km ; le trajet partiel est :
  Nancy - Metz - Reims - Troyes - Paris ; la distance alors parcourue est 507 km.
- Partant de Paris, il ne reste plus qu'à rentrer à Nancy soit 303 km ; le trajet partiel est :
  Nancy - Metz - Reims - Troyes - Paris ; la distance alors parcourue est 810 km.

Le trajet obtenu par notre algorithme glouton n'est pas le trajet de longueur minimale (vous avez dû trouver 709 km pour le trajet : Nancy - Metz - Reims - Paris - Troyes - Nancy), la distance parcourue est supérieure de 100 km, mais c'est mieux que certains trajets qui dépassaient les mille kilomètres et surtout on a évalué un trajet et pas tous les trajets.

2.3 Mise en œuvre

Le script Python ci-dessous, est une implémentation de l'algorithme ci-dessus.

```
# voyageur glouton

def plus_proche(ville, dist, visitees):
    """Renvoie le numéro de la ville non encore visitée la
       plus proche de la ville courante, en supposant qu'il
       en existe au moins une.
    """
    pp = None
    for i in range(len(visitees)):
        if visitees[i]: continue
        if pp == None or dist[ville][i] < dist[ville][pp]:
            pp = i
    return pp

def voyageur(villes, dist, depart):
    """Affiche les étapes du parcours glouton depuis la
       ville de départ.
    """
    n = len(villes)
    visitees = [False] * n
    courante = depart
    for _ in range(n-1):
        visitees[courante] = True
        suivante = plus_proche(courante, dist, visitees)
        print("on va de", villes[courante], \
              "à", villes[suivante], \
              "en", dist[courante][suivante], "km")
        courante = suivante
    print("on revient de", villes[courante], \
          "à", villes[depart], \
          "en", dist[courante][depart], "km")
```

 voyageur_commerce_algorithme_glouton_2.py

Les `n` villes à visiter sont numérotées de `0` à `n-1` et les distances d'une ville à une autre sont stockées dans le tableau : `dist`. `dist[i][j]` donnant la distance séparant la ville `i` et la ville `j`.

La fonction principale `voyageur` prend en paramètre la liste des villes, la table des distances et le numéro de la ville de départ. Elle utilise un tableau `visitees` de booléens indiquant pour chaque ville (donnée par son numéro en indice) si elle a été visitée ou non, ainsi qu'une variable`courante` mémorisant le numéro de la ville actuelle. La fonction est ensuite constituée d'une boucle qui, `n-1` fois, marque comme visitée la ville courante puis sélectionne la suivante.

La fonction `voyageur` délègue le choix de la ville la plus proche à une fonction auxiliaire `plus_proche`, qui prend en paramètres la table des distances, le numéro de la `ville` courante et le tableau des villes déjà visitées pour sélectionner la ville la plus proche de la ville courante parmi les villes non encore visitées et en mémorisant dans une variable `pp` le numéro de la ville non visitée la plus proche vue jusque là. Cette variable est initialisée avec la valeur spéciale `None` puisqu'on ne connait a priori pas le numéro de la première ville éligible.

Travail à faire :
Écrire les doc-string des deux fonctions `plus_proche` et `voyageur`.
Entrer les variables `dist` et `ville` correspondant à l'exemple initial, vérifier que la fonction `voyageur` renvoie bien la solution trouvée par notre glouton au 2.2.

3 Problème du rendu de monnaie

3.1 Introduction du problème

Considérons le problème d'un commerçant devant rendre la monnaie à ses clients, il souhaite utiliser le moins de pièces ou de billets possibles. On suppose que l'on manipule les coupures et pièces habituelles en euros et on ignore les centimes. On suppose de plus que le commerçant dispose d'une réserve suffisante de chaque pièce.
Si la somme qui doit être rendue est 9, les différentes combinaisons possibles sont :

Combinaison	Pièces
9×1 €	9
7×1 € $+ 1 \times 2$ €	8
5×1 € $+ 5 \times 2$ €	7
3×1 € $+ 3 \times 2$ €	6
1×1 € $+ 4 \times 2$ €	5
4×1 € $+ 1 \times 5$ €	5
2×1 € $+ 1 \times 2$ € $+ 1 \times 5$ €	4
2×2 € $+ 1 \times 5$ €	3

La dernière combinaison est la solution optimale.

3.2 Rendu de monnaie glouton

Pour aborder le problème du rendu de monnaie avec une stratégie gloutonne, on va donc sélectionner les pièces et billets à rendre un à un, et faire décroitre progressivement la somme restant à rendre. Chaque choix doit être celui qui parait le meilleur au vu de la situation présente, c'est à dire de la somme restant à rendre. Pour limiter le nombre de pièces rendues, on choisit de faire décroitre cette somme aussi vite que possible, c'est à dire de sélectionner à chaque fois la plus grande valeur disponible qui ne soit pas strictement supérieure à la somme à rendre

```
# rendu de monnaie

euros = [1, 2, 5, 10, 20, 50, 100, 200]

def monnaie(s):
    """combien de pièces faut-il pour obtenir la somme s"""
    i = len(euros) - 1
    p = 0
    while s > 0:
        if s >= euros[i]:
            s -= euros[i]
            p += 1
        else:
            i -= 1
    return p
```

 rendu_monnaie.py

Travail à faire :
Écrire la doc-string de la fonction `monnaies` et et commenter le script `rendu_monnaie.py`

4 Exercices

Exercice 1 :

Écrire un programme qui détermine la somme `s`, en dessous de 200 €, qui si on devait rendre cette somme `s` demanderait le plus grand nombre de pièces. Dit autrement on cherche un entier `s` plus petit que 200 qui maximise le résultat de `monnaie(s)`.

Exercice 2 : Problème du sac à dos

Arsène L. a devant lui un ensemble d'objets de valeurs et de poids variés. Il dispose d'un sac à dos dans lequel prendre une partie des objets, en essayant de maximiser la valeur totale emportée. Cependant, il ne pourra emporter le sac qui si le poids total est inférieur ou égal à 10 kilogrammes. Dans chacune des situations suivantes, indiquer les différentes combinaisons qui peuvent être formées et les valeurs correspondantes.

Situation 1	Masse en kg	Valeur en €
Objet A	6	4 800
Objet B	5	3 500
Objet C	4	3 000
Objet D	1	500

Situation 2	Masse en kg	Valeur en €
Objet A	8	4 800
Objet B	5	4 000
Objet C	4	3 000
Objet D	1	500

Situation 3	Masse en kg	Valeur en €
Objet A	9	8 100
Objet B	6	7 200
Objet C	5	5 500
Objet D	4	4 000
Objet E	1	800

Situation 4	Masse en kg	Valeur en €
Objet A	7	9 100
Objet B	6	7 200
Objet C	4	4 800
Objet D	3	2 700
Objet E	2	2 600
Objet F	1	200

Exercice 3 : Algorithmes gloutons pour le sac à dos

Arsène L. ne voulant pas arriver en retard à son rendez-vous avec la comtesse C. il va devoir choisir très rapidement les objets à emporter. Appliquer chacune des trois stratégies gloutonnes suivantes aux situations de l'exercice précédent.

1. Choisir les objets par ordre de valeur décroissante parmi ceux qui ne dépassent pas la capacité restante.
2. Choisir les objets par ordre de poids croissant.
3. Choisir les objets par ordre de rapport valeur/poids décroissant parmi ceux qui ne dépassent pas la capacité restante

Exercice 4 : piéger les stratégies gloutonnes

Pour chacune des stratégies gloutonnes de l'exercice 3, trouver des situations dans lesquelles la valeur emportée est aussi éloignée que possible de la valeur optimale.

Exercice 5 :

Supposons avoir une liste d'activité, chacune associée à un créneau horaire défini par une heure de début et une heure de fin. Deux activités sont compatibles si leurs créneaux horaires ne se chevauchent pas. On souhaite sélectionner un nombre maximal d'activités toutes compatibles entre elles.

1. On se donne des activités avec les créneaux horaires suivants :8h - 13h, 12h - 17h, 9h - 11h, 14h - 16h, 11h - 12h. Combien de ces activités peuvent-elles être conciliées sur une seule journée ?
2. On propose une stratégie gloutonne pour sélectionner des activités en commençant par le début de la journée : choisir l'activité dont l'heure de fin arrive le plus tôt (parmi les activités dont l'heure de début est bien postérieure aux créneaux des activités déjà choisies). Appliquer cette stratégie à la situation précédente.
   Remarque : cette stratégie gloutonne donne toujours un nombre d'activités optimal.

Exercice 6 :

On reprend le problème de l'exercice précédent et on suppose avoir `n` activités numérotées de `0` à `n-1`, et deux tableaux `début` et `fin` de taille `n` tels que `debut[i]` et `fin[i]` contiennent respectivement l'heure de début et l'heure de fin de l'activité numéro `i`.

1. Écrire une fonction `prochaine(debut, fin, h)` qui sélectionne, parmi les activités dont l'heure de début n'est pas antérieure à `h`, une s'arrêtant le plus tôt. On demandera à la fonction de renvoyer `None` s'il n'y a aucun créneau compatible.
2. En déduire une fonction `selection(debut, fin)` qui, en supposant que toutes les heures sont positives, sélectionne autant d'activités que possible en suivant la stratégie gloutonne. On demandera à la fonction d'afficher les numéros des activités sélectionnées.