Seq. 14 – Rappels Algorithmie

Objectifs

1. Algorithme des k plus proches voisins
2. Recherche dichotomique dans un tableau trié
3. Algorithmes gloutons (problèmes du sac à dos ou du rendu de monnaie)

I. Parcours séquentiel d'une liste

Dans l'ensemble de cette fiche, les exemples utilisent des listes de personnes. Une personne est modélisée par un tuple (nom, age).

A. Algorithmes de cumul

Exemple de moyenne

Voici une fonction qui renvoie la moyenne d'âge, arrondie à l'entier inférieur, d'une liste de personnes.

```
def moyenne(liste_personnes):
    somme = 0
    for (nom, age) in liste_personnes:
        somme = somme + age
    return somme // len(liste_personnes)
```

```
>>> import moyenne
>>> moyenne([('Zoé', 17), ('Léa', 19), ('Léo', 12)])
16
```

Exemple de comptage

Voici une fonction qui compte le nombre de personnes d'une liste qui ont strictement moins de 15 ans.

```
def nb_bebes(liste_personnes):
    compteur = 0
    for (nom, age) in liste_personnes:
        if age < 15:
            compteur = compteur + 1
    return compteur
```

```
>>> import nb_bebes
>>> nb_bebes([('Ano', 14), ('Léa', 19), ('Léo', 12)])
2
```

Autres exemples d'algorithme de la même famille

- Calculer la somme des nombres positifs d'une liste de nombres.
- Compter le nombre de voyelles dans une chaîne de caractères.

B. Recherche d'un extremum (maximum ou minimum)

Voici une fonction qui renvoie le nom de la personne la plus âgée d'une liste.

```
def le_plus_vieux(liste_personnes):
    (nom_vieux, age_max) = liste_personnes[0]
    for (nom, age) in liste_personnes:
        if age > age_max:
            (nom_vieux, age_max) = (nom, age)
    return nom_vieux
```

```
>>> import le_plus_vieux
>>> le_plus_vieux([('Zoé', 17), ('Léa', 19), ('Léo', 12)])
Léa
```

Autres problèmes de la même famille :

- Trouver le nombre le plus proche de zéro dans une liste de nombres ;
- Trouver l'indice du mot qui contient le plus de « a » dans une liste de mots.

C. Algorithmes de vérification

On parcourt la liste en faisant une recherche/vérification à chaque étape. On s'arrête dès qu'on trouve ce que l'on cherche ou un contre-exemple dans le cas d'une vérification.

Voici une fonction qui renvoie l'indice de la première personne qui a au moins 15 ans dans une liste (et None si aucune personne ne correspond au critère).

```
def premier_ado(liste_personnes):
    for i in range(len(liste_personnes)):
        (nom, age) = liste_personnes[i]
        if age >= 15:
            return i
    return None
```

```
>>> import premier_ado
>>> premier_ado([('Léo', 12), ('Léa', 19),('Zoé', 17)])
1
>>> premier_ado([('Léo', 12), ('Zu', 11),('Dora', 14)])
None
```

À NOTER

Ici, nous utilisons return en milieu de boucle pour quitter la fonction. C'est tolérable.

Autres problèmes de la même famille :

- Vérifier qu'une personne dont le nom est passé en paramètre est bien membre d'une liste de personnes ;
- Vérifier qu'une liste de personnes est rangée en ordre décroissant d'âges ;
- Chercher le premier mot de plus de 5 lettres dans une liste de mots.

II. Recherche dichotomique dans une liste triée

A. Problème posé

Nous voulons rechercher l'élément 7 dans la liste [1, 2, 5, 9, 10, 14, 17, 24, 41].
Avec une recherche séquentielle ou recherche par balayage, on parcourt la liste du début à la fin en comparant chaque valeur à l'élément recherché.
Dans le pire des cas, on parcourt la liste en entier. Dans notre exemple, il faut faire 9 comparaisons.
Comme la liste de départ est triée, la recherche dichotomique permet d'améliorer la performance de la recherche.

MOT CLÉ

Le nom dichotomie provient du grec ancien *dikhotomia* qui signifie « couper en deux

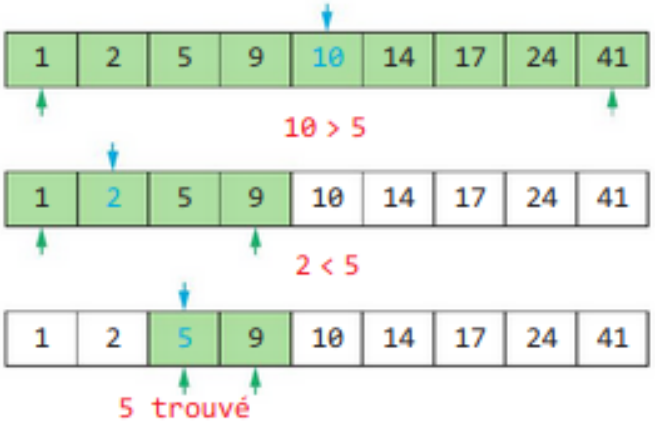
B. Principe de la recherche dichotomique

Voici le principe de la recherche dichotomique avec une liste triée dans l'ordre croissant :

- Si la liste est vide : répondre négativement, la recherche est finie.
- Sinon, trouver la valeur la plus centrale de la liste et comparer cette valeur à l'élément recherché :
  - si la valeur est celle cherchée : répondre positivement, la recherche est finie ;
  - si la valeur est strictement plus petite que l'élément recherché, reprendre la procédure avec la seconde moitié de la liste ;
  - sinon reprendre la procédure avec la première moitié de la liste.

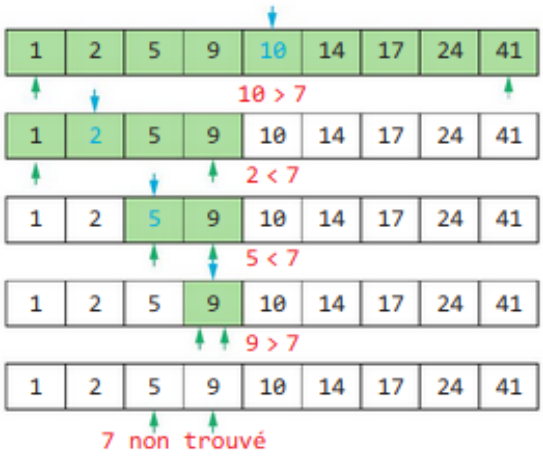
Exemple 1 :

Recherche de l'élément 5 dans la liste triée [1, 2, 5, 9, 10, 14,17, 24, 41]



Exemple 2 :

Recherche de l'élément 7 dans la liste triée [1, 2, 5, 9, 10, 14, 17, 24, 41] :



Il suffit de 4 tours de boucle pour conclure qu'un élément n'est pas dans une liste de taille 9.

C. Nombre de tours de boucle maximum

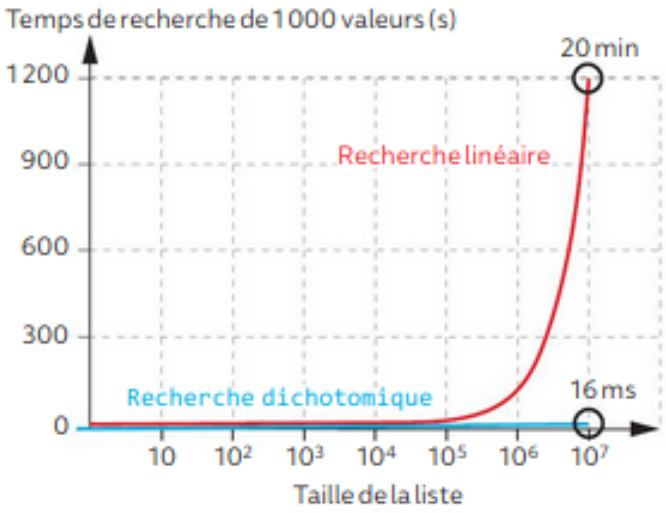
Taille de la liste	0	1	2	4	8	16	32	64	128	N
Recherche séquentielle	0	1	2	4	8	16	32	64	128	N
Recherche dichotomique	0	1	2	3	4	5	6	7	8	$\log_2 N$

Le nombre de tours de boucle de la recherche dichotomique est donc de l'ordre de $\log_2(n)$ où n est la taille de la liste.

Complexité :

1 000 recherche dans une liste de 10 000 000 éléments :

- recherche linéaire → ≈ 20 min ;
- recherche dichotomique → ≈ 16 ms.



D. Exemple de mise en œuvre

Recherche dichotomique d'un élément dans une liste triée :

```
def recherche_dicho(liste, element):
    """ 'liste' doit être triée dans l'ordre croissant
    Renvoie True si element est dans la liste, False sinon
    """
    indice_debut = 0
    indice_fin = len(liste) - 1
    while indice_debut <= indice_fin :
        indice_centre = (indice_debut + indice_fin) // 2
        valeur_centrale = liste[indice_centre]
        if valeur_centrale == element:
            return True
        elif valeur_centrale < element:
            indice_debut = indice_centre + 1
        else:
            indice_fin = indice_centre - 1
    return False
```

Ce document est une adaptation de livre aux éditions Hatier : https://monespace-educ.fr/feuilleter/9782401052307
et du site pixees : https://pixees.fr/informatiquelycee/n_site/nsi_prem.html

III. Introduction à l'algorithme des k plus proches voisins

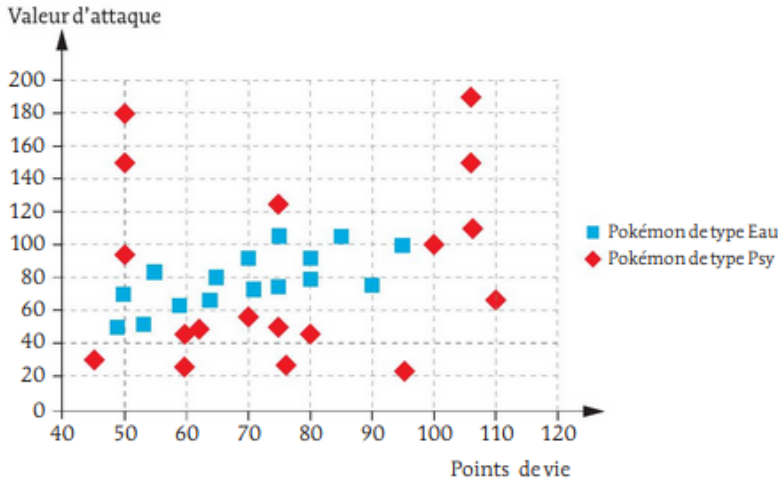
L'algorithme des k plus proches voisins est l'un des algorithmes utilisés dans le domaine de l'intelligence artificielle. Il intervient dans de nombreux domaines de l'apprentissage automatique.

A. Un problème de classification

Voici un problème qui peut être résolu en utilisant l'algorithme des k plus proches voisins.
De façon très simpliste, admettons que les Pokémons ne possèdent que deux caractéristiques : leurs points de vie et leur valeur d'attaque. On suppose qu'ils se répartissent en deux types seulement : Eau et Psy.

Nom	Écayon	Deoxys	Éoko	Groret	Tarpaud
Points de vie	49	50	80	90	90
Attaque	49	95	45	75	75
Type	Eau	Psy	Psy	Psy	Eau

Le fichier de l'échantillon pokemons.csv est disponible sur le site des éditions Hatier : http://hatier-clic.fr/19nsi01
À partir de cet échantillon, on veut pouvoir prédire la classification d'un Pokémon mystère à partir de la donnée de ses points de vie et de sa valeur d'attaque.



B. Algorithme de prédiction

a. À l'aide d'un diagramme

À partir des données représentées sur le diagramme, on veut prédire la classe d'un Pokémon qui a 65 points de vie et 40 en attaque. On trouve dans l'échantillon les 6 plus proches voisins :

Parmi ces 6 voisins, il y a deux Pokémons de type Eau et quatre de type Psy. Il est donc probable que notre Pokémon mystère soit un Pokémon de type Psy.

b. Formulation de l'algorithme de prédiction

Pour automatiser la classification, il faut formuler un algorithme de façon formelle. Pour prédire la classe d'un Pokémon donné, il faut des données :
– un échantillon de Pokémons ;
– un Pokémon_mystère dont on veut prédire la classification ;
– la valeur de k.
Une fois ces données modélisées, la formulation de l'algorithme de prédiction est assez simple.
Algorithme :
1. Trouver, dans l'échantillon, les k plus proches voisins de Pokémon_mystère.
2. Parmi ces proches_voisins, trouver la classification majoritaire.
3. Renvoyer la classification_majoritaire.
Remarque : La valeur k = 6 est ici un choix arbitraire. Cette valeur doit néanmoins être choisie judicieusement : trop faible, la qualité de la prédiction diminue ; trop grande, la qualité de la prédiction diminue aussi. Par exemple, dans l'exemple précédent avec k = 34, la prédiction sera toujours Psy (classe majoritaire dans l'échantillon).

C. Implémentation de l'algorithme des k plus proches voisins

a. Algorithme naïf

Voici un algorithme qui permet de résoudre le problème.
Données :
• une table de données de taille n : table ;
• une donnée cible : cible ;
• un entier k plus petit que n ;
• une règle permettant de calculer la « distance » entre deux données.
Résultat : les k plus proches voisins de la cible ;
Algorithme :
1. Trier les données de la table selon la distance croissante avec la donnée cible.
2. proches_voisin est la liste des k premières données de la table triée.
3. Renvoyer proches_voisins.

b. Implémentation en Python

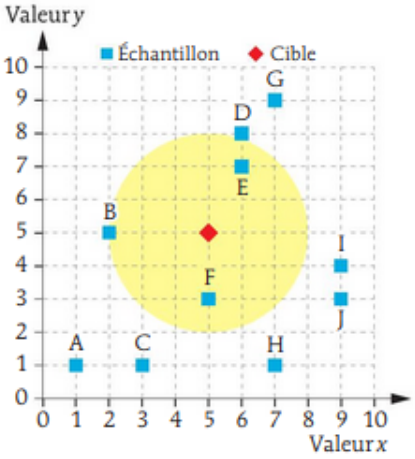
On suppose donnée une fonction distance qui calcule la distance entre deux données. On peut alors implémenter en Python de la façon suivante :

```
def k_plus_proches_voisins(table, cible, k):
    # 1. Trier les données de la table selon la
    # distance croissante avec la donnée cible
    # on définit le critère de tri
    def distance_cible(donnee):
        """ renvoie la distance entre une donnée
        et la cible """
        return distance(donnee, cible)
    # on trie la table selon le critère choisi
    table_triee = sorted(table, key = distance_cible)
    # 2. Prendre les k premières valeurs de la table triée
    proches_voisins = [ ]
    for i in range(k):
        proches_voisins.append(table_triee[i])
    # 3. Renvoyer proches_voisins
    return proches_voisins
```

c. Choix de la distance

Le choix du mode de calcul de la distance entre deux données n'est pas anodin.

En voici l'illustration avec un échantillon de 10 données (points bleu) et d'une cible (point rouge).



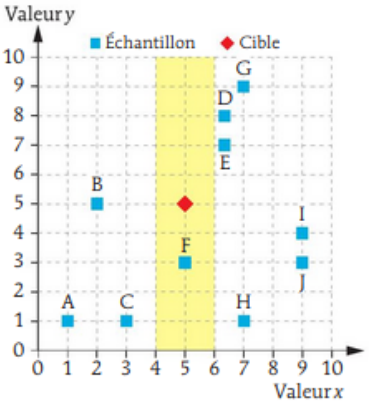
En utilisant la distance « naturelle », c'est-à-dire celle qui est donnée par une règle graduée, les trois plus proches voisins de la cible sont les points B, E et F.
Dans un repère orthonormé, cette distance est donnée par la formule :

$$distance(point\,1, point\,2)=\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$.

Si maintenant on considère que les valeurs sur l'axe des ordonnées n'ont pas
d'intérêt, on utilise une distance qui ne dépend pas de y, par exemple :

$$distance(point\,1, point\,2)=|x_1-x_2|$$

Dans ce cas, les trois plus proches voisins de la cible sont les points D, E et F.



IV. Exemple d'algorithme glouton : le rendu de monnaie

En informatique, on rencontre souvent des problèmes d'optimisation comme celui mis en œuvre dans le rendu de monnaie. Les algorithmes gloutons sont couramment utilisés pour les résoudre.

A. Le rendu de monnaie

On veut programmer une caisse automatique pour qu'elle rende la monnaie de façon optimale, c'est-à-dire avec le nombre minimal de pièces et billets.
La valeur des pièces et billets à disposition sont : 1, 2, 5, 10, 20, 50, 100 et 200 euros. On dispose d'autant d'exemplaires qu'on le souhaite de chaque pièce et billet.
Exemple : Anaïs veut acheter un objet qui coûte 53 euros. Elle paye avec un billet de 100 euros. La caisse automatique doit lui rendre 47 euros.
La meilleure façon de rendre la monnaie est de le faire avec deux billets de 20, un billet de 5 et une pièce de 2 euros.

a. Résolution du problème de rendu de monnaie

La force brute

La solution naïve consiste à énumérer toutes les solutions possibles puis choisir la solution optimale, celle qui minimise le nombre de pièces et de billets. On peut totaliser 47 euros de différentes façons, par exemple :

Rendus de monnaie	Nombre de pièces et billets
47 × 1 €	47
45 × 1 € + 1 × 2 €	46
6 × 1€ + 3 × 2 € + 3 × 5 € + 2 × 10 €	14
7 × 1 € + 4 × 10 €	11
...	...

Cette méthode permet de trouver la solution optimale globale au problème, mais le nombre de possibilités est très important. L'utilisation d'un tel algorithme ici est très coûteux en temps de calcul.

L'algorithme glouton

Le principe de l'algorithme glouton

Utiliser un algorithme glouton consiste à optimiser la résolution d'un problème en utilisant l'approche suivante : on procède étape par étape, en faisant, à chaque étape, le choix qui semble le meilleur, sans jamais remettre en question les choix passés.

Application au rendu de monnaie

Dans l'exemple du problème de rendu de monnaie, on commence par rendre la pièce ou le billet de la plus grande valeur possible, c'est-à-dire dont la valeur est inférieure à la somme à rendre. On déduit alors cette valeur de la somme à rendre, ce qui conduit à un problème de plus petite taille. On recommence ainsi jusqu'à obtenir une somme nulle.

Données : la *somme* à rendre et la *liste* des pièces et billets à disposition.

Résultat : une liste des pièces et billets à rendre.

Algorithme :

• initialiser *monnaie* à la liste vide ;

• initialiser *somme_restante* à *somme* ;

• tant que la *somme_restante* est strictement positive :

  * on choisit dans liste la plus grande valeur qui ne dépasse pas la somme restante,
  * on ajoute cette valeur à *monnaie*,
  * *somme_restante* = *somme_restante* - valeur choisie ;

• renvoyer *monnaie*.

Solution du problème de rendu de monnaie

Dans notre exemple, on a :

```
somme = 47
liste = [1, 2, 5, 10, 20, 50, 100, 200]
```

L'algorithme permet de résoudre le problème étape par étape :

Initialisation	monnaie = []	somme_restante = 47
Étape 1	monnaie = [20]	somme_restante = 27
Étape 2	monnaie = [20, 20]	somme_restante = 7
Étape 3	monnaie = [20, 20, 5]	somme_restante = 2
Étape 4	monnaie = [20, 20, 5, 2]	somme_restante = 0

```
Résultat : [20, 20, 5, 2].
```

Un algorithme glouton permet de trouver une solution optimale locale au problème, mais pas toujours une solution optimale globale.

Par exemple, si notre caisse automatique doit rendre une somme de 63 euros, mais qu'elle ne dispose plus de billets de 5 euros ni de 10 euros.

```
somme = 63
liste = [1, 2, 20, 50, 100, 200]
```

L'algorithme glouton donne comme résultat :

```
resultat = [50, 2, 2, 2, 2, 2, 2, 1]
```

Alors que la solution optimale globale est :

```
solution_optimale_globale = [20, 20, 20, 2, 1]
```

A faire vous-même 1.

Un cambrioleur possède un sac à dos d'une contenance maximum de 30 Kg. Au cours d'un de ses cambriolages, il a la possibilité de dérober 4 objets A, B, C et D. Voici un tableau qui résume les caractéristiques de ces objets :

caractéristiques des objets

objet	A	B	C	D
masse	13 Kg	12 Kg	8 Kg	10 Kg
valeur marchande	700 €	400 €	300 €	300 €

Déterminez les objets que le cambrioleur aura intérêt à dérober, sachant que :

- tous les objets dérobés devront tenir dans le sac à dos (30 Kg maxi)
- le cambrioleur cherche à obtenir un gain maximum.

Appliquons une méthode gloutonne à la résolution du problème du sac à dos :

- Sachant que l'on cherche à maximiser le gain, commençons par établir un tableau nous donnant la "valeur massique" de chaque objet (pour chaque objet on divise sa valeur par sa masse) :

objet	A	B	C	D
valeur massique	54 €/Kg	33 €/Kg	38 €/Kg	30 €/Kg

- On classe ensuite les objets par ordre décroissant de valeur massique : A - C - B -D
- Enfin, on remplit le sac en prenant les objets dans l'ordre et en s'arrêtant dès que la masse limite est atteinte. C'est ici que ce fait "le choix glouton", à chaque étape, on prend l'objet ayant le rapport "valeur-masse" le plus intéressant au vu des objectifs :
    - 1re étape : A (13 Kg)
    - 2e étape : C (13+8=21 Kg)
    - 3e étape : B (13+8+12=33 Kg) => impossible, on dépasse les 30 Kg.

Le sac est donc composé de 2 objets : A et C pour un montant total de 1000 € et une masse totale de 21 Kg.

Vous proposerez ensuite une implémentation en Python de cet algorithme.