

## F E U I L L E D E T D N° 1 2

Le **tri rapide** (Quicksort) est un algorithme de tri très efficace basé sur un principe de "diviser pour mieux régner".

La fonction quicksort est une fonction récursive qui fonctionne comme suit :

1. Si la liste L est de taille inférieure ou égale à 1, renvoyer la liste L.
2. Choisir un élément pivot dans la liste L (on prend un élément tiré au hasard, ou le premier élément).
3. Séparer la liste privée en trois sous-listes L1, L2 et L3 comprenant respectivement les éléments inférieurs au pivot, les éléments supérieurs au pivot, et les éléments égaux au pivot.
4. Créer une liste LT qui est la concaténation de quicksort(L1), de L3 et de quicksort(L2).
5. Renvoyer LT

**Exercice 1 (Tri rapide).**

1. Créer une fonction **Separation** qui prend en entrée une liste L et une valeur  $k$ , et qui renvoie trois listes L1, L2 et L3, contenant tous les éléments de L inférieurs à  $k$ , tous ceux strictement supérieurs à  $k$  d'autre part, et enfin tous ceux égaux à  $k$ .
2. Appliquer l'algorithme de tri rapide à la liste  $L = [8, 3, 9, 6, 5]$ , en choisissant à chaque fois un élément au hasard comme pivot. On représentera l'arbre de tous les appels récursifs nécessaires.  
Pour choisir au hasard une valeur entre 0 et 1, on utilisera la fonction **random** du module **random**.
3. Écrire une fonction **quicksort** qui effectue le tri rapide d'une liste.
4. Tester l'algorithme sur les listes définies précédemment.

**Remarque :** La complexité en moyenne du tri rapide est  $O(n \ln(n))$ . Il est ainsi plus efficace que le tri par sélection et que le tri à bulles.

Cependant, le pire des cas pour le tri rapide se produit lorsque la liste est déjà triée ou presque triée, entraînant une complexité de  $O(n^2)$ .

## VARIABLES ALÉATOIRES

**Exercice 2.** Un joueur joue à pile ou face : s'il gagne, il emporte  $g$  euros, et s'il perd, il doit donner  $p$  euros. Il joue  $n$  parties. Représenter sur un graphe l'évolution de ses gains au cours du temps pour différentes valeurs de  $g$ ,  $p$  et  $n$ . On créera pour cela une fonction **partie(g,p,n)** qui prend en entrée  $g$ ,  $p$  et  $n$  et qui renvoie en sortie la liste des gains du joueur au cours des  $n$  parties (on prendra 0 comme gain initial).

On utilisera la fonction **randint** du module **random** pour le côté aléatoire.

## NOMBRES COMPLEXES

Le module **numpy** permet aussi de manipuler des nombres complexes. Une fois celui-ci chargé, le nombre complexe  $i$  est obtenu en écrivant  $1j$ . Ainsi,  $2 - 3i$  s'écrira en python  $2+(-3)*1j$

On utilisera les commandes suivantes pour tracer une courbe à l'aide d'une liste **seg** de nombres complexes :

```
import matplotlib.pyplot as plt
X=[s.real for s in seg] #Liste des abscisses de chaque sommet
Y=[s.imag for s in seg] #Liste des ordonnées de chaque sommet
plt.plot(X,Y)
plt.axis("equal")
plt.axis("off")
plt.show()
```

**Exercice 3 (Le jeu du chaos).**

Le jeu du chaos consiste, à partir d'un triangle équilatéral  $ABC$  et d'un point  $M_0$  choisi au hasard à l'intérieur de ce triangle, de construire une suite de points  $M_1, M_2, \dots, M_n$  tels que pour tout  $i \in 1, \dots, n-1$ ,  $M_{i+1}$  est le milieu de  $M_i$  et d'un sommet de  $ABC$  choisi au hasard uniforme (parmi  $A$ ,  $B$  et  $C$ ).

1. Écrire une fonction **chaos(S,n)** qui prend en entrée un triangle équilatéral  $S$  sous forme d'une liste de nombres complexes et un nombre entier  $n$ , et qui renvoie la liste des nombres complexes associés à  $M_0, M_1, \dots, M_n$ .  
On s'aidera des fonctions **random** et **randint** du module **random**.  
Pour construire  $M_0$ , on pourra par exemple prendre le point d'affixe

$z = \frac{pz_A + qz_B + rz_C}{p+q+r}$ , où  $p, q, r$  sont des réels choisis aléatoirement entre 0 et 1.

2. Tracer quelques figures pour différentes valeurs de  $n$  et différents essais. Utiliser en un premier temps la fonction `plot` du module `matplotlib.pyplot`, puis ensuite la fonction `scatter`. Quelle figure obtient-on ?
3. Procéder de même en remplaçant le triangle initial par un pentagone régulier.

## ALGORITHMES GLOUTONS

Un problème d'optimisation est un problème algorithmique dans lequel l'objectif est de trouver la «meilleure» solution (selon un critère donné) parmi un ensemble de solutions également valides mais potentiellement moins bonnes. Par exemple, déterminer le minimum ou le maximum d'une fonction est un problème d'optimisation. On peut également citer la répartition optimale de tâches suivant des critères précis, le problème du rendu de monnaie, le problème du sac à dos, la recherche d'un plus court chemin dans un graphe, le problème du voyageur de commerce.

Le contexte d'un problème d'optimisation est donc :

- ▷ un très grand nombre de solutions (dans le cas contraire, il n'y aurait pas de difficulté à trouver la meilleure),
- ▷ une fonction permettant d'évaluer la qualité de chaque solution,
- ▷ l'existence d'une solution optimale (ou suffisamment satisfaisante).

De nombreuses techniques informatiques sont susceptibles d'apporter une solution exacte ou approchée à ces problèmes. Certaines de ces techniques, comme l'énumération exhaustive de toutes les solutions (on parle de méthode par force brute), ont un coût machine qui les rend souvent peu pertinentes au regard de contraintes extérieures imposées (temps de réponse de la solution imposé, moyens machines limités).

Un algorithme glouton (greedy algorithm en anglais) est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local, dans l'espoir

d'obtenir un résultat optimum global, et de ne jamais revenir en arrière sur un choix déjà effectué.

Nous allons étudier dans quelques situations où l'on peut mettre en œuvre un algorithme glouton. Nous verrons que :

- ▷ le principal avantage des algorithmes gloutons est leur facilité de mise en œuvre.
- ▷ leur principal défaut est qu'ils ne renvoient pas toujours la solution optimale.

### Exercice 4 (Problème du rendu de monnaie).

Vous allez à la boulangerie acheter du pain. Vous payez, et attendez votre monnaie. Il est souhaitable que cela se fasse avec un minimum de pièces, aussi bien pour le vendeur que pour le client. Des caisses automatiques ont été mises au point à cet effet : le client glisse les billets ou les pièces dans la machine, et elle se charge de rendre la monnaie.

Nous allons à présent nous intéresser à la manière dont un automate rend la monnaie. En interne, l'automate ne manipule que des centimes d'euro, qui ont l'avantage d'être des nombres entiers.

L'automate a accès aux valeurs en cents des pièces, stockées dans la liste `VALEURS` définie de la manière suivante : `VALEURS = [200, 100, 50, 20, 10, 5, 2, 1]`

Vous définirez cette liste comme une variable globale en début de script à l'aide de la commande `global`, afin d'y accéder dans les fonctions que vous écrirez par la suite.

On modélise les pièces utilisées par le monnayeur pour rendre la monnaie par une liste de même taille que `VALEURS`, et qui stocke à chaque indice  $i$  le nombre de valeurs `VALEURS[i]` utilisées.

Par exemple, la liste `pieces = [0, 0, 5, 0, 2, 1, 0, 2]` représente une liste de cinq pièces de 50 cents, deux de 10 cents, une de 5 cents et deux de 1 cent, pour un montant total de 277 cents, soit 2,77 euros.

1. Écrire une fonction `pieces2cents(pieces)` qui convertit une liste de nombres de pièces (de même taille que la liste `VALEURS`) en sa valeur totale en cents. La fonction `sum` est interdite.
2. Écrire une fonction `rendu_glouton(montant)` qui implémente l'algorithme glouton pour le rendu de monnaie, cherchant à chaque étape

à rendre la pièce de plus haute valeur possible.

3. On appelle canonique un système de pièces pour lequel l'algorithme glouton donne toujours une solution optimale (au sens où le nombre pièces utilisées est minimal), quelle que soit la valeur du montant à rendre.

Prouver qu'un système avec trois types de pièces : 1,3 et 4 euros (le nom de la monnaie importe peu) n'est pas canonique.

On peut heureusement prouver que le système de monnaie en euro est canonique.

4. Le système et billets que l'on utilise (1,2,5,10,20,50,...) ressemble en partie à une utilisation de puissances de 2, mais pas totalement. Pourquoi n'utilise-t-on pas un système de pièces dont les valeurs sont des puissances de 2 (1,2,4,8,16) ?

5. Un automate ne dispose en réalité que d'un nombre fini de pièces de chaque type pour rendre la monnaie.

Le nombre de pièces restantes de chaque type est stocké dans une variable globale (de type `list`) `CAISSE` telle que `CAISSE[i]` représente le nombre de pièces de valeur faciale `VALEURS[i]` restant en stock.

Créer une fonction Python `rendu_glouton2(montant)` pour prendre en compte le fait que le réservoir de pièces disponibles n'est pas infini.

Par exemple, s'il faut 5 pièces de 50 cents mais qu'il n'y en a que 3 dans la caisse, on ne peut en rendre que 3 et il faudra rajouter par exemple 5 pièces de 20 cents pour combler le manque.

6. Tester la fonction sur quelques exemples judicieusement choisis : l'automate peut-il toujours rendre la monnaie ?