# TP 1 : Utilisation des piles

## 1 Préambule

Télécharger sur mon site web une implémentation des piles non bornées. Les fonctions sont les mêmes que celles du cours. Avec `P` une pile et `x` un élément :

- `creer_pile()` crée et renvoie une pile vide.
- `empiler(P,x)` ajoute `x` au sommet de `P` ;
- `pile_vide(P)` renvoie un booléen indiquant si `P` est vide ;
- `sommet(P)` renvoie le sommet de `P` si `P` est non vide ;
- `depiler(P)` supprime et renvoie le sommet de `P`, si `P` est non vide.

Dans la suite, vous ne manipulerez des piles que via ces fonctions [1]. Ceci dit, une fonction `afficher(P)` vous affichera le contenu de la pile au moment où vous le demandez, pour vous aider à débuguer éventuellement.

## 2 Un labyrinthe parfait

Sur la grille discrète $[\![0, n-1]\!] \times [\![0, n-1]\!]$, on considère les $n^2$ nœuds $c = (i, j)$. Deux nœuds de la grille sont voisins si ils sont à distance exactement 1. Un chemin simple entre deux nœuds $c_1$ et $c_2$ est une suite de nœuds tous distincts et successivement voisins $c_1, \ldots, c_k, \ldots, c_2$, d'extrémités $c_1$ et $c_2$. Un labyrinthe parfait est le résultat du tracé de segments entre nœuds voisins, tel que, pour toute paire de nœuds de la grille, il existe un et un seul chemin simple entre ces nœuds.

Voir les figures ci-dessous pour des exemples de labyrinthes parfaits de dimension $5 \times 5$ et $18 \times 18$. Pour le second, on a marqué en rouge l'unique chemin simple allant du coin inférieur gauche (0, 0) au coin supérieur droit (17, 17).

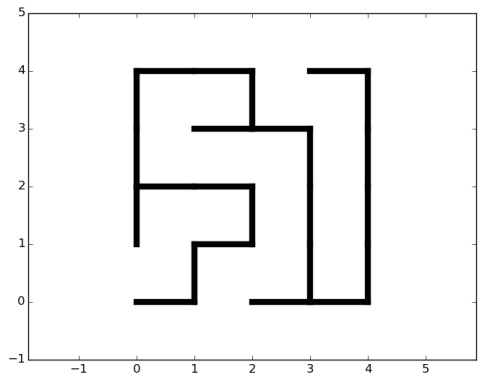


FIGURE 1: Un labyrinthe parfait de taille $5 \times 5$.

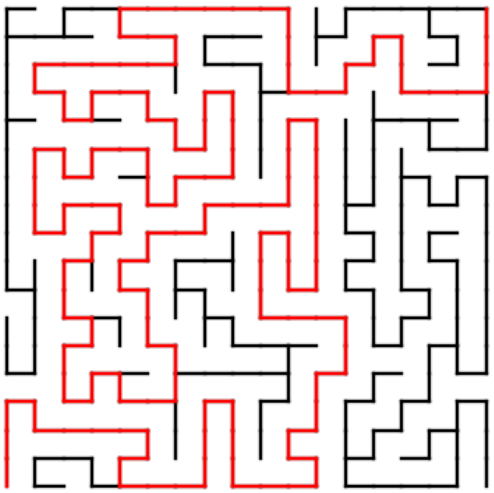

FIGURE 2: Un labyrinthe parfait de taille $18 \times 18$, l'unique chemin de la case $(0, 0)$ à la case $(17, 17)$ marqué en rouge

À la différence des livres de jeux pour enfants, cheminer dans ces labyrinthes consiste à suivre les « lignes-chemins » plutôt qu'à se déplacer entre des « lignes-parois ».

La construction d'un tel labyrinthe consiste à visiter les nœuds de la grille à l'aide d'une structure de pile. Pour savoir quels nœuds de la grille ont été visités, on utilisera une grille de booléens, de dimension $N \times N$, pour identifier les nœuds déjà visités. La structure est une liste de listes. Par exemple, avec $N = 3$, la liste

1. Et vous n'avez pas le choix. L'implémentation choisie fait que seul l'élément au sommet est accessible.

```
noeud_visites=[[True, True, False], [True, False, False], [False, False, True]]
```

est une telle liste de listes. Notez que la taille $N$ du labyrinthe est alors accessible comme la longueur de cette liste. Si $(i, j) \in [\![0, N-1]\!]$, la case $(i, j)$ a été visitée si et seulement si `noeuds_visites[i][j]` est `True`.

Pour construire le labyrinthe, on utilise l'algorithme 1.

**Algorithme 1 :** Construction d'un labyrinthe parfait de taille $N \times N$

`noeuds_visites` ← grille de booléens de taille $N \times N$, tous faux;
`a_traiter` ← une pile vide ; empiler $(0, 0)$ sur `a_traiter` ; marquer $(0, 0)$ comme visité dans `noeuds_visites` ;
**tant que** *la pile* ***a_traiter*** *est non vide* **faire**
  Dépiler le « nœud-sommet » $c$ de la pile ;
  Identifier les nœuds voisins de $c$ non encore visités;
  **si** *il y a au moins un nœud voisin de c non encore visité* **alors**
    en choisir un aléatoirement : $s$;
    tracer le chemin $(c, s)$;
    empiler $c$ sur `a_traiter`;
    empiler $s$ sur `a_traiter`;
    marquer $s$ comme visité dans `noeuds_visites`

## 2.1 Des fonctions auxiliaires

Les fonctions suivantes aident à mettre en oeuvre l'algorithme. Attention, une difficulté est de ne pas sortir de la grille. Pour le tracé, on importe (déja fait dans l'annexe) classiquement le module `matplotlib.pyplot` sous la forme :

```
import matplotlib.pyplot as plt
```

**Question 1.** Écrire une fonction `trace(c,d)` prenant en entrée deux couples de coordonnées, et traçant (avec `plt.plot`) le segment `[c,d]`. (Rappel : `plt.plot(X,Y)` relie les points $(x_i, y_i)$ par des segments. Ici, il suffit de créer deux listes de taille 2, contenant les deux abscisses et les deux ordonnées !). On ne l'affichera pas (pas de `plt.show()` ici).

Les trois fonctions qui suivent prennent en paramètre :

— un couple d'entiers ;

— une liste de listes de booléens `nv`.

`nv` est une grille carrée de booléens, analogue de `noeuds_visites` présenté précédemment. La longueur $N$ de la liste donne la taille du labyrinthe.

**Question 2.** Écrire une fonction `visiter(c, nv)`. On suppose que $c$ est un couple de $[\![0, N-1]\!]^2$. La fonction positionne simplement la case de `nv` indexée par `c` à `True`.

**Question 3.** Écrire une fonction `a_ete_visitee(c,nv)`. En notant $N$ la taille de `nv`, la fonction doit renvoyer un booléen indiquant si la case $c$ a déja été visitée, **en considérant une case hors de la grille comme visitée.** Ainsi, votre fonction devra lire le booléen de `nv` indexé par $c$, seulement si $c$ est dans $[\![0, N-1]\!]^2$.

**Question 4.** En déduire une fonction `voisins_non_visites(c, nv)` prenant en entrée un couple d'entiers (qu'on suppose être une case de la grille, première et seconde composante sont entre 0 et $N-1$, avec `N` la taille de `nv`), et renvoyant la liste des voisins de $c$ non encore visités. La liste renvoyée a donc entre 0 et 4 éléments, attention à ne pas faire de dépassement d'indice dans le cas où la case `c` est sur le bord ou dans un coin. (*Indication :* `for x,y in [(a,b-1), (a,b+1), (a+1,b), (a-1,b)]` fait parcourir au couple $(x, y)$ les 4 voisins de $(a, b)$).

**Question 5.** La fonction `randint` du module `random` (déja importée dans l'annexe) prend en entrée deux entiers $a \leq b$ et retourne aléatoirement un entier de l'intervalle $[\![a, b]\!]$. Écrire une fonction `tirage_sort(L)` prenant en entrée une liste supposée non vide, et renvoyant un élément aléatoire de cette liste.

### 2.2 Le programme principal

**Question 6.** Écrire une fonction `labyrinthe(N)` permettant de tracer un labyrinthe parfait de taille $N \times N$. On rappelle que `[[False]*N for i in range(N)]` permet de créer une liste $N$ de listes contenant $N$ `False`. On utilisera `plt.show()` pour afficher le labyrinthe, en fin de fonction. On pourra, pour obtenir quelque chose de plus joli, préciser :

```
plt.figure()
plt.axis('equal')
plt.axis('off')
plt.ylim(ymin=-1)
plt.ylim(ymax=N)
```

en début de fonction, et modifier `trace` pour qu'elle affiche les segments en noir (option `'k'`).

**Question 7.** Estimer la complexité du script en fonction de $N$, en supposant toutes les opérations de piles de complexité constante.

## 3 Enveloppe convexe dans le plan

Cette partie a pour objectif de calculer des enveloppes convexes de nuages de points dans le plan affine, un grand classique en géométrie algorithmique. On rappelle qu'un ensemble $C \subseteq \mathbb{R}^2$ est convexe si et seulement si pour toute paire de points $p, q \in C$, le segment de droite $[p, q]$ est inclus dans $C$. L'enveloppe convexe d'un ensemble $\mathcal{N} \subseteq \mathbb{R}^2$, notée $\text{Conv}(\mathcal{N})$, est le plus petit convexe contenant $\mathcal{N}$. Dans le cas où $\mathcal{N}$ est un ensemble fini (appelé nuage de points), le bord de $\text{Conv}(\mathcal{N})$ est un polygône convexe dont les sommets appartiennent à $\mathcal{N}$, comme illustré dans la figure 3.

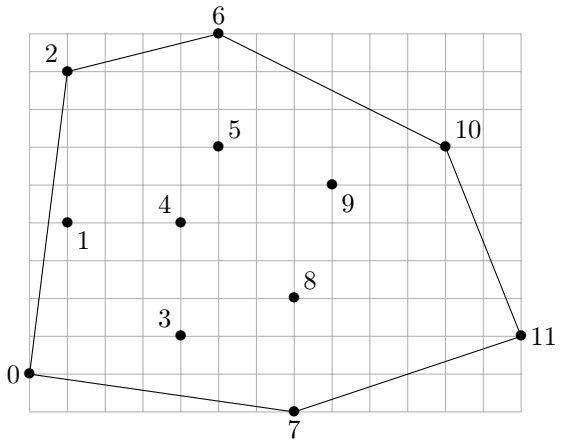


FIGURE 3: L'enveloppe convexe d'un nuage de points

Les points du nuage sont donnés sous la forme de couples $(x, y)$. Lorsque deux tels couples $(x, y)$ et $(x', y')$ sont comparés en Python (avec `<`, `<=`, `>` ou `>=`), ils le sont d'abord sur la première composante, puis sur la deuxième en cas d'égalité. En clair :

$$(x, y) \leq (x', y') \quad \iff \quad x < x' \quad \text{ou} \quad x = x' \quad \text{et} \quad y \leq y'$$

### 3.1 Tri du nuage, et génération aléatoire

On rappelle que si `L` est une liste, `L.sort()` trie en place la liste (`sort` ne renvoie rien, mais la liste est triée après appel, comme pour les tris du cours).

**Question 8.** La fonction `random()`, sans argument, issue du module du même nom (déja importée dans l'annexe), permet de générer aléatoirement un réel de l'intervalle $[0, 1[$. À partir de cette fonction, écrire une fonction `genere_nuage(N)` générant un nuage de $N$ points (couples de réels) de $[0, 1[^2$, dont les points sont triés dans l'ordre croissant (on générera une liste de $N$ couples, qu'on triera à l'aide de la méthode `sort`). *Remarque :* le nuage de points du texte est présent dans l'annexe sous le nom `nuage_texte`.

**Question 9.** Écrire une fonction `trace_nuage(L)` permettant de tracer un nuage de points `L` (liste de couples de réels) à l'écran. Pour cela, il suffit de répartir abscisses et ordonnées dans deux listes `X` et `Y`, et d'utiliser `plt.plot(X,Y,'o')`.

## 3.2 Test d'orientation

On suppose que le nuage de points (de taille au moins 3) est « en position générale », c'est-à-dire que :

— les points du nuage sont distincts ;

— trois points quelconque du nuage ne sont jamais alignés.

Ces hypothèses ne sont pas restrictives pour des nuages générés aléatoirement avec des flottants.

On suppose donc le nuage de points ordonné par abscisse croissante, et par ordonnée croissante en cas d'égalité sur l'abscisse, et donné sous la forme d'une liste de points. L'algorithme utilisé procède par balayage, en construisant une enveloppe convexe supérieure et une enveloppe convexe inférieure. Le seul test géométrique utilisé est celui de l'orientation donné par un triplet de points (voir schéma suivant) : un triplet $(P, Q, R)$ est dit orienté positivement si l'angle $(\vec{PQ}, \vec{PR})$ est dans $]0, \pi[$ (modulo $2\pi$) et négativement dans le cas contraire. Rappelons que trois points du nuage n'étant jamais alignés, l'angle n'est jamais nul modulo $\pi$.


FIGURE 4: Orientation positive et orientation négative

**Question 10.** Écrire une fonction `orientation(P,Q,R)`, prenant en entrée trois points supposés non alignés, et renvoyant 1 si le triplet $(P, Q, R)$ est orienté positivement, et $-1$ dans le cas contraire. Indication : on s'intéresse essentiellement au signe du déterminant $\det(\vec{PQ}, \vec{PR})$. Calculer d'abord les composantes de ces vecteurs, puis le déterminant.

## 3.3 Calcul de l'enveloppe convexe

L'idée de l'algorithme est de balayer le nuage de points horizontalement de gauche à droite par une droite verticale, tout en mettant à jour l'enveloppe convexe des points du nuage situés à gauche de cette droite, comme illustré dans la figure 5.

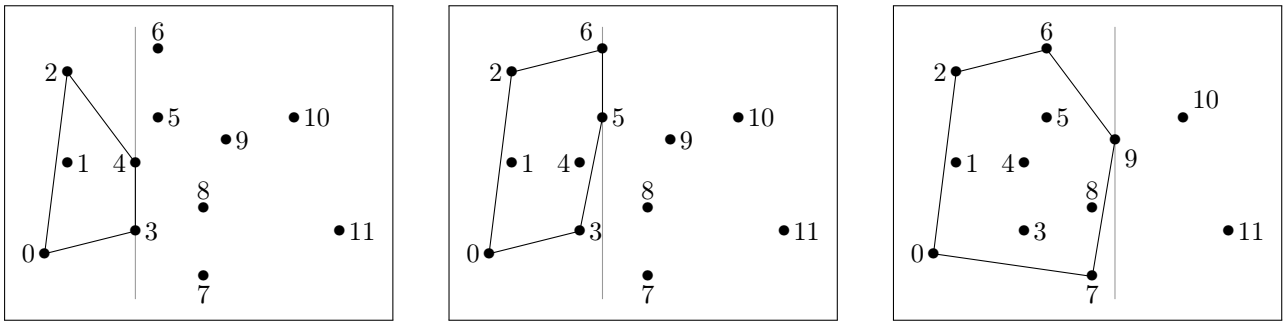


FIGURE 5: Différentes positions de la droite de balayage

Plus précisément, l'algorithme visite chaque point du nuage une fois, par ordre croissant d'abscisse (donc par ordre croissant d'indice dans la liste `L` car celle-ci est triée). À chaque nouveau point $p_i$ visité, il met à jour le bord de l'enveloppe convexe du sous-nuage $\{p_0, \dots, p_i\}$ situé à gauche de $p_i$. On remarque que les points $p_0$ et $p_i$ sont sur ce bord, et on appelle enveloppe supérieure la partie du bord de $\text{Conv}\{p_0, \cdots, p_i\}$ située au-dessus de la droite passant par $p_0$ et $p_i$ ($p_0$ et $p_i$ compris), et enveloppe inférieure la partie du bord de $\text{Conv}\{p_0, \dots, p_i\}$ située au-dessous ($p_0$ et $p_i$ compris). Le bord de $\text{Conv}\{p_0, \dots, p_i\}$ est donc constitué de l'union de ces deux enveloppes, après suppression des doublons de $p_0$ et $p_i$.

Par exemple, dans le cas du nuage de la figure 5 à gauche, le sous-nuage $\{p_0, p_1, p_2, p_3, p_4\}$ a pour enveloppe supérieure la séquence $(p_0, p_2, p_4)$ et pour enveloppe inférieure la séquence $(p_0, p_3, p_4)$, le bord de son enveloppe convexe

étant donné par la séquence $(p_0, p_3, p_4, p_2)$. Informatiquement, les sommets des enveloppes inférieure et supérieure seront stockés dans deux piles séparées, `ei` (pour enveloppe inférieure) et `es` (pour enveloppe supérieure).

La mise à jour de l'enveloppe supérieure est illustrée dans la figure 6 : tant que le point visité ($p_9$ dans ce cas) et les deux points situés au sommet de la pile `es` (dans l'ordre : $p_8$ et $p_6$) forme une séquence $(p_9, p_8, p_6)$ d'orientation négative, on dépile le point situé au sommet de `es` ($p_8$ dans ce cas). On poursuit ce processus d'élimination jusqu'à ce que l'orientation devienne positive ou qu'il ne reste plus qu'un seul point dans la pile. Le point visité ($p_9$ dans ce cas) est alors inséré au sommet de `es`.

La mise à jour de l'enveloppe inférieure s'opère de manière symétrique.

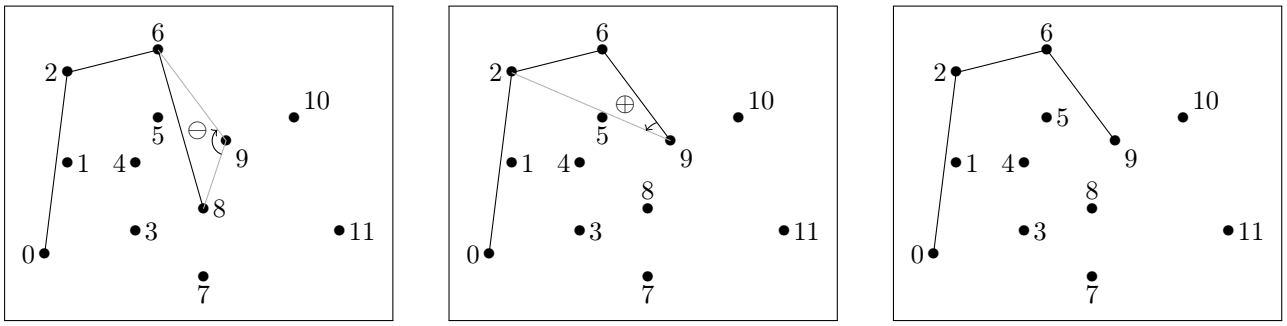


FIGURE 6: Mise à jour de l'enveloppe supérieure lors du traitement du point $p_9$ : l'orientation $(p_9, p_8, p_6)$ est négative donc $p_8$ est supprimé. L'orientation $(p_9, p_6, p_2)$ est positive donc le procédé s'arrête, la pile `es` contient en fin de fonction 0, 2, 6, 9, le 9 étant au sommet.

**Question 11.** Écrire une fonction `majES(es, c)` qui prend en paramètre la pile `es` et un nouveau point $c$ à visiter, et qui met à jour l'enveloppe supérieure du sous-nuage. Le temps d'exécution de votre fonction doit être un $O$ du nombre de points dans l'enveloppe supérieure. On n'utilisera que les fonctions sur les piles, en prenant garde à ne pas accéder au sommet ou dépiler une pile vide. On rappelle à toute fin utile que les opérateurs sur booléens sont `not`, `and` et `or`. On pourra supposer que `es` contient au moins un point (en pratique $p_0$ est toujours au fond).

**Question 12.** Écrire de même une fonction `majEI(ei, c)` qui effectue la mise à jour de l'enveloppe inférieure, avec le même temps d'exécution.

**Question 13.** Ecrire maintenant une fonction `enveloppe_convexe(L)` qui prend en paramètre la liste `L` de taille $n$ ($n$ est accessible avec `len`) représentant le nuage, et qui effectue le balayage des points du nuage comme décrit précédemment. On supposera les éléments de `L` déjà triés par ordre croissant d'abscisse. La fonction renverra les deux piles contenant l'enveloppe convexe sous la forme d'un couple `ei, es`, avec $p_0$ au fond et $p_{n-1}$ au sommet. Par exemple, sur le nuage de la figure 3, le résultat de la fonction `enveloppe_convexe` doit être une pile (`ei`) contenant la suite de points $p_0, p_7, p_{11}$ d'une part, et une pile (`es`) contenant $p_0, p_2, p_6, p_{10}, p_{11}$ d'autre part.

*Indication :* créer deux piles `ei` et `es` dans lequelles on empilera le premier point $p_0$, puis appliquer dans une boucle les deux fonctions précédentes.

**Question 14.** Analyser le temps d'exécution de l'algorithme de balayage décrit précédemment, en supposant une fois encore que les points du nuage fourni en entrée sont déjà triés par abscisse croissante. On essaiera de ne pas surestimer le temps d'exécution.

**Question 15.** Écrire une fonction `trace_enveloppe(ei,es)` prenant en entrée les deux enveloppes convexes inférieure et supérieure et traçant l'enveloppe totale. On pourra faire usage de la fonction `trace` de la section II.

**Question 16.** Faire des tests !