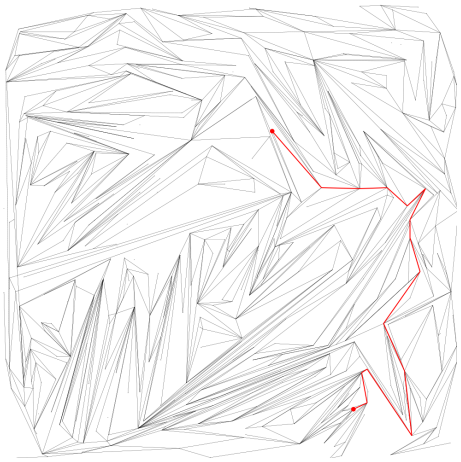
L'objectif de ce sujet est d'écrire une classe `Heap` qui implémente les tas, ainsi que trois applications : le tri par tas, les files de priorité, et le calcul du plus court chemin comme illustré ci-contre.



Pour le calcul du plus court chemin, nous implémenterons l'algorithme de Dijkstra vu en cours, avec et sans file de priorité pour comparer les performances.

Comme toujours, les spécifications précises des méthodes à compléter sont indiquées dans le code. Pensez à regarder les résultats attendus dans les tests si vous n'êtes pas sûr de comprendre une spécification.

1 Méthodes de base

Remarquez que l'on peut construire un tas stockant n'importe quel type `T`, du moment que les éléments du type `T` savent se comparer. Nous allons écrire dans cette section une classe `Heap<T extends Comparable<T> >` implémentant les tas. Tous les algorithmes vu en cours pour un un max-tas (dans lequel les grands éléments sont vers la racine) sont les mêmes pour un min-tas (dans lequel les petits éléments sont vers la racine). En effet, dans toutes les situations où l'on vérifie si un élément `e1` doit être placé au dessus (plus proche de la racine) d'un `e2` dans un max-tas en calculant `e1.compareTo(e2) >= 0`, il suffira pour un min-tas d'effectuer `e2.compareTo(e1) >= 0`. Afin d'éviter des duplications de code, nous avons donc ajouté dans la classe `Heap` une méthode `isSup(T e1, T e2)` qui retournera `e1.compareTo(e2) >= 0` ou `e2.compareTo(e1) >= 0` selon que this est un max-tas ou un min-tas. Vous devez donc coder toutes les méthodes de `Heap` en utilisant uniquement `isSup`.

Exercice 1. left, right, father.
Rappelez vous (cf cours) qu'un tas est un arbre semi-complet ayant la propriété du maximum. Le fait d'être semi-complet permet de stocker l'arbre dans le tableau `t` donné en attribut, tout en ayant facilement accès aux indices des fils gauche/droite ou du père d'un sommet.

Question 1.1.
Ecrire les méthodes `int left(int i)`, `int right(int i)` et `int father(int i)`.

Exercice 2. add.

Question 2.1.
Ecrire la méthode `void heapifyUp(int i)`, et lancez le test approprié. Pensez à utiliser la méthode `swap` qui permet d'échanger le contenu de deux cases d'une arraylist.

Question 2.2.
En déduire la méthode `void add(T e)`, et lancez le test approprié.

Exercice 3. remove.

Question 3.1.
Ecrire la méthode `void heapifyDown(int i)`, et écrivez vous même un test approprié en vous inspirant du test pour `heapifyUp`.

Question 3.2.
En déduire la méthode `void remove(int i)`. Inspirez vous de la stratégie vue en cours pour `removeTop()`.

Question 3.3.
En déduire la méthode `void removeTop()`, et lancez le test approrié
.

2 Application : implémentation du tri par tas

Nous allons ici implémenter le *tri par tas* dont le principe est le suivant. Pour trier un tableau `t` à n éléments par ordre croissant, créer un tas en y insérant tous les éléments de `t`, puis éxecuter ensuite n fois `removeTop()` pour extraire l'élément le plus petit à chaque fois. La complexité est de $\mathcal{O}(t_c + nt_r)$ où t_c est le temps de création (en comptant l'insertion des n éléments) du tas, et t_r le temps de `removeTop()`. Etant donné que `add` et `removeTop` sont tous les deux en $\mathcal{O}(log(n))$, on obtient $t_c = \mathcal{O}(nlog(n))$ et $t_r = \mathcal{O}(nlog(n))$, et donc un tri en $\mathcal{O}(nlog(n))$, ce qui améliore par exemple le tri naïf par minimum (chercher le minimum à chaque étape) qui est en $\mathcal{O}(n^2)$. Nous avons vu dans le cours que l'on peut même obtenir $t_c = \mathcal{O}(n)$.

Question 3.4.
Ecrire le constructeur `Heap(boolean isMaxHeap, ArrayList<T> tt)` si possible en temps $\mathcal{O}(n)$ et pas $\mathcal{O}(nlog(n))$.

Question 3.5.
Ecrire la méthode `ArrayList<T> toSortedArray()` de la classe `Heap`, et la méthode `ArrayList<T> heapSort(ArrayList<T> t)` de la classe `Test`. Vérifiez avec le test `compareTri`.

3 Application : implémentation d'une file de priorité

Une file de priorité (priority queue) est une structure permettant de ranger des couples (e, p) avec e de n'importe quel type `T` (auquel on ne demande pas d'implémenter `comparable`), et p est un `double` représentant la priorité de e.

Notez que nous avons déjà fourni le code de la classe `ElemWithPriority` permettant de stocker de tels couples, et qui elle implémente `comparable` en se basant sur les priorités (regardez comment est codée le `compareTo`).

Question 3.6.
Ecrire les méthodes `void add(T e, double p)`, `T getTop()` et `T removeTop()` de `PriorityQ`. Vérfiez avec le test `testPQeueueAddgGetRemoveTop`.

Notre file de priorité est prête, nous sommes maintenant armés pour implémenter l'algorithme de Dijsktra comme nous l'avons étudié en cours.

4 Application : implémentation de l'algorithme de Dijsktra

Nous allons ici implémenter un algorithme qui étant donné un graphe $G = (V, E)$ avec des poids strictement positifs sur les arêtes, et deux sommets s, t, calcule un plus court (en terme de somme des poids sur ses arêtes) chemin de s à t. Nous fournissons une classe `Graph` modélisant un graphe, stocké sous forme de liste d'adjacences. Vous pouvez ne pas regarder le détail du code de cette classe, mais parcourez les spécifications pour voir quelles méthodes sont fournies.

Question 3.7.
Dans le main de la classe `Test`, créer (en utilisant les méthodes déjà fournies dans `Graph`) le graphe à 3 sommets $\{0, 1, 2\}$ avec les arêtes $\{0, 1\}$ de poids 10, et $\{1, 2\}$ de poids 20, et l'afficher dans le terminal pour comprendre comment est stocké un tel graphe.

Créer de petits graphes à la main sera utile pour tester votre futur algorithme de plus court chemin, mais pour le tester sur de "vrais" grands graphes, nous avons fourni le constructeur `Graph(int n,boolean randomGraphWithCoord)` qui, lorsque le booléen est à vrai, crée au hasard un graphe planaire à (environ) n sommets et $2n$ arêtes. Plus précisément, cette méthode :

- Tire au hasard n points dans le rectangle $[0, n-1] \times [0, n-1]$. Chaque point correspondra à un sommet du graphe, et les coordonées du sommet i sont stockées dans `coord.get(i)`. Remarque, on retirera les points ayant les mêmes coordonnées, et donc en pratique on obtiendra parfois x points seulement, avec x légèrement inférieur à n.
- Ajoute au hasard $2x$ arêtes (ou moins si cela n'est pas possible) qui ne se croisent pas, et ajoute ces arêtes au graphe, qui sera donc un graphe planaire. Si une arête tirée est $\{i, j\}$, alors le poids de cette arête sera la distance (dans le plan) entre `coord.get(i)` et `coord.get(j)`.

Nous avons également fourni deux méthodes `toSVG()` et `toSVGWithPath()` qui servent à visualiser un graphe (et un chemin) au format `SVG`. C'est plus pratique que dans un terminal quand le graphe aura 3000 sommets!

Question 3.8.
Dans le main de la classe `Test`, créer un graphe aléatoire à 200 sommets avec le constructeur précédent, et l'exporter en `SVG`. Ouvrez le fichier `SVG` obtenu pour vérifier!

Question 3.9.
Ecrire dans la classe `Graph` la méthode `ArrayList<Integer> dijkstraWithoutPriorityQ(int s, int t)` qui implémente Dijkstra de façon naïve, c'est à dire avec une simple `ArrayList<Integer>` pour gérer l'ensemble `avoir`. Ecrivez d'abord une version 1 de base dans laquelle on commence par ajouter tous les sommets à `avoir`, comme dans le cours. Notez que, à chaque itération, cela coûte `avoir.size()` de chercher le sommet de plus courte distance à extraire de `avoir`, ce qui peut être long. Ainsi, une fois que votre version 1 fonctionne, écrivez une version 2 dans laquelle on ajoute initialement que s dans `avoir`, puis progressivement les nouveaux sommets (que l'on détecte comme "nouveaux" car leur distance à s est ∞). Cela permet de garder `avoir.size()` plus petit en pratique, même si cela ne change pas le pire cas. Testez avec `testDijWithoutPriorityQ`. Testez également en créeant un petit graphe aléatoire (10 sommets par exemple), et en exportant en SVG le chemin que vous avez calculé avec `toSVGWithPath`.

Question 3.10.
Ecrire la méthode `ArrayList<Integer> dijkstraWithPriorityQ(int s, int t)` qui implémente Dijkstra en utilisant une file de priorité pour gérer l'ensemble `avoir`. Vous pouvez là aussi faire deux versions (en insérant initialement tous les sommets dans la file, ou non). Testez avec `testDijWithPriorityQ`.

Comparons maintenant les versions pour comprendre l'impact de la file de priorité. On fournit dans `PlusCourtChemin` un `main` qui crée un graphe aléatoire G, puis lance, sur 30 couples (s, t) tirés au hasard, une fois chaque version de l'algorithme de Dijkstra.

Question 3.11.
On rappelle que notre constructeur aléatoire de graphe génère un graphe avec $m = 2n$ arêtes. En regardant les complexités des deux versions dans le cours, quelle version devrait être plus rapide quand n devient grand ?