X2018 — INF411
Contrôle classant / 19 novembre 2019 / 9h–12h

Tous documents de cours autorisés (polycopié, transparents, notes personnelles). Le dictionnaire papier est autorisé pour les EV2 et EV3. Les ordinateurs, Ipad, dictionnaires électroniques, les tablettes et téléphones portables sont interdits.

L'énoncé est composé de deux problèmes indépendants, que vous pourrez traiter dans n'importe quel ordre (en revanche, merci de clairement numéroter les réponses). Vous pourrez, quand vous traiterez une question, considérer comme déjà traitées les questions précédentes du même problème.

Le correcteur prêtera attention à la qualité de la rédaction et vous remercie d'avance d'écrire lisiblement. Merci également de numéroter vos copies en indiquant leur nombre total.

Quelques éléments de la bibliothèque standard Java sont rappelés à la fin du sujet. Ils ne sont pas forcément tous nécessaires. Vous pouvez utiliser librement tout autre élément de la bibliothèque standard Java.

1 Arbres couvrants

Dans ce problème, on s'intéresse à des graphes *non orientés* dont les arcs sont étiquetés par des *poids*. On ne considère que des graphes *connexes*, c'est-à-dire où il existe toujours un chemin entre deux sommets donnés, et *sans boucles*, c'est-à-dire sans arc $a - a$.

En Java, on choisit de représenter de tels graphes à l'aide de deux classes, `Edge` et `Graph`, données figure 1. Les sommets d'un graphe sont représentés par les entiers 0,1,...,`N`-1 où `N` est le nombre de sommets, donné par le champ `N`. La structure du graphe est donnée par le tableau `adj`, où `adj[`i`]` est la liste des arcs issus du sommet i. Un arc est décrit par un objet de la classe `Edge`, où les champs `src` et `dst` sont les deux extrémités de l'arc et où le champ `weight` est le poids de l'arc. Les poids sont ici des nombres flottants, de type `double`.

Le graphe étant non orienté, un arc $a - b$ entre les sommets a et b apparaît à la fois comme un arc issu de a, c'est-à-dire un objet e de type `Edge` dans la liste `adj[`a`]`, avec e`.src` $= a$ et e`.dst` $= b$, et comme un arc issu de b, c'est-à-dire un *autre* objet e' de type `Edge` dans la liste `adj[`b`]`, avec e'`.src` $= b$ et e'`.dst` $= a$. Ces deux objets représentent le *même arc*. En particulier, on a donc e`.weight` $= e'$`.weight`.

```
class Edge {
  final int src, dst;     // un arc entre src et dst
  final double weight;    // le poids de cet arc
}

class Graph {
  final int N;            // les sommets sont les entiers 0,1,...,N-1
  LinkedList<Edge>[] adj; // un tableau de taille N
                          // adj[i] est la liste des arcs issus de i
}
```

FIGURE 1 – Représentation des graphes.

```
class UF {
  UF(int n)       // construit une structure pour les éléments 0,1,...,n-1
  int find(int i) // renvoie le représentant de la classe de i
  void union(int i, int j) // fusionne les classes des éléments i et j
}
```

FIGURE 2 – Structure union-find.

Arbre couvrant. Étant donné un graphe G, un *arbre couvrant* de G est un sous-ensemble T d'arcs de G tel que

1. T est connexe et sans cycle (c'est pourquoi on parle d'*arbre*) ;
2. chaque sommet de G est l'extrémité d'au moins un arc de T (on dit que T *couvre* tous les sommets de G).

Voici par exemple un graphe de six sommets à gauche et l'un de ses arbres couvrants à droite.

(On a ignoré pour l'instant les poids qui étiquettent les arcs.)

Question 1 Montrer que, si un graphe G possède N sommets, tout arbre couvrant de G est composé d'exactement $N-1$ arcs.

Correction : On montre que tout graphe connexe acyclique de N sommets contient $N-1$ arcs, par récurrence forte sur N. C'est clair pour $N=1$. Soit un graphe connexe acyclique de $N \geq 2$ sommets. Soit v l'un de ses sommets. Il y a au moins un arc issu de v, car G est connexe. Les $k \geq 1$ arcs issus de v relient v à autant de graphes $G_1, \ldots, G_k$ qui sont eux-mêmes connexes et acycliques. Par hypothèse de récurrence, chaque graphe G_i possède N_i sommets et $N_i - 1$ arcs, avec $N = 1 + N_1 + \cdots + N_k$. Par ailleurs, les graphes G_i ne sont pas reliés entre eux, sans quoi il y aurait un cycle dans G. Le nombre d'arcs de G est donc $k + (N_1 - 1) + \cdots + (N_k - 1) = N - 1$, CQFD.

Question 2 Déduire du résultat de la question précédente une méthode

```
static boolean isSpanningTree(Graph g, LinkedList<Edge> t)
```

qui détermine si la liste d'arcs `t` constitue un arbre couvrant du graphe `g`. Indication : pour tester l'absence de cycle, on pourra utiliser une structure union-find, dont l'interface est donnée figure 2. (On ne demande pas de coder une telle structure.) Donner la complexité en temps de votre méthode, en fonction de N. On rappelle que la complexité de la méthode `size` de `LinkedList` est $O(1)$ et que les opérations `find` et `union` peuvent être considérées de complexité $O(1)$ amortie en pratique.

Correction :

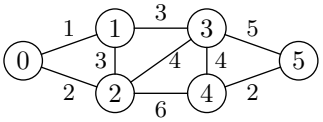
```
static boolean isSpanningTree(Graph g, LinkedList<Edge> t) {
  if (t.size() != g.N - 1) return false;
  UF uf = new UF(g.N);
  for (Edge e: t) {
    if (uf.find(e.src) == uf.find(e.dst)) return false;
```

```
      uf.union(e.src, e.dst);
    }
    return true;
  }
```

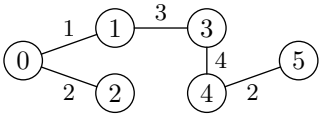
Le premier test est en $O(1)$. La construction de la structure union-find est en $O(N)$. Ensuite, on fait une boucle sur tous les arcs de `t`, au nombre de $N-1$, et pour chacun on exécute un code en $O(1)$ amorti (deux `find` et un `union`). D'où une complexité totale en $O(N)$.
Une autre solution, toujours en $O(N)$, consiste à faire `union` pour tous les arcs de `t` puis à vérifier que tous les sommets sont dans la même classe (celle de 0, par exemple).

Arbre couvrant minimal. Étant donné un graphe G, un *arbre couvrant minimal* de G est un arbre couvrant de G dont la somme des poids des arcs est minimale.

Question 3 Donner un arbre couvrant minimal pour le graphe suivant, où chaque arc est étiqueté par son poids.



Correction :



Le poids total est 12.

Algorithme de Kruskal. Pour construire un arbre couvrant minimal pour un graphe G, on peut utiliser l'*algorithme de Kruskal*, dont le fonctionnement est le suivant :

1. soit U une structure union-find pour les sommets $0, 1, \ldots, N-1$ de G ;
2. soit Q une file de priorité contenant tous les arcs de G, ordonnés par leur poids ;
3. soit T une liste d'arcs, initialement vide ;
4. tant que T contient moins que $N-1$ arcs :
   (a) retirer un arc $x-y$ de poids minimal de la file de priorité Q,
   (b) si x et y ne sont pas dans la même classe pour U, alors
      i. ajouter l'arc $x-y$ à T,
      ii. fusionner dans U les classes de x et y.

À la fin de l'algorithme, la liste T contient un arbre couvrant minimal pour G.

Question 4 Donner les étapes de l'algorithme de Kruskal sur le graphe de la question 3, sous la forme suivante :

étape	arc $x-y$	poids	action	U
				$\{0\},\{1\},\{2\},\{3\},\{4\},\{5\}$
1	...	...	...	...
2	...	...	...	...
$\vdots$				

Chaque ligne indique l'arc $x-y$ retiré de la file, son poids, s'il est ignoré ou ajouté à T (action) et la structure U obtenue (ensemble des classes).

Correction :

arc $x-y$	poids	action	U
			$\{0\},\{1\},\{2\},\{3\},\{4\},\{5\}$
$0-1$	1	ajouté	$\{0,1\},\{2\},\{3\},\{4\},\{5\}$
$0-2$	2	ajouté	$\{0,1,2\},\{3\},\{4\},\{5\}$
$4-5$	2	ajouté	$\{0,1,2\},\{3\},\{4,5\}$
$1-2$	3	ignoré	—
$1-3$	3	ajouté	$\{0,1,2,3\},\{4,5\}$
$2-3$	4	ignoré	—
$3-4$	4	ajouté	$\{0,1,2,3,4,5\}$

On s'arrête là car T contient alors 5 arcs. En particulier, les arcs $3-5$ et $2-4$ ne sont pas retirés de la file.

Note : ce n'est pas la seule exécution possible, car il y a plusieurs arcs de même poids. Ainsi, si on considère l'arc $3-4$ avant l'arc $2-3$, on fait une étape de moins.

Programmation de l'algorithme de Kruskal. Pour programmer l'algorithme de Kruskal en Java, on va utiliser les classes `PriorityQueue` et `LinkedList` de la bibliothèque standard de Java (voir en annexe) pour réaliser Q et T, et la classe `UF` de la figure 2 pour réaliser U. On commence par la programmation de deux méthodes auxiliaires.

Question 5 Ajouter à la classe `Graph` une méthode `LinkedList<Edge> allEdges()` qui renvoie la liste de tous les arcs du graphe, dans un ordre arbitraire. Chaque arc doit apparaître *une seule fois* dans le résultat. Ainsi, si le graphe contient par exemple un arc $4-7$, alors la liste renvoyée doit contenir un objet `e` avec `e.src` $= 4$ et `e.dst` $= 7$, ou un objet `e` avec `e.src` $= 7$ et `e.dst` $= 4$, mais pas les deux.

Correction : Une solution simple consiste à n'ajouter que les arcs $x-y$ avec $x \leq y$.

```
LinkedList<Edge> allEdges() {
  LinkedList<Edge> l = new LinkedList<>();
  for (int v = 0; v < this.N; v++)
    for (Edge e: this.adj[v])
      if (e.src <= e.dst)
        l.add(e);
  return l;
}
```

Note : On peut écrire tout aussi bien `e.src < e.dst` car le graphe ne contient pas de boucle.

Question 6 Pour construire une file de priorité de type `PriorityQueue<Edge>`, il faut que la classe `Edge` implémente l'interface `Comparable<Edge>`, avec une méthode de comparaison adéquate. Modifier la classe `Edge` en conséquence.

Correction : Il faut d'une part ajouter la déclaration

```
class Edge implements Comparable<Edge> {
  ...
```

et d'autre part fournir la méthode de comparaison

```
  @Override
  public int compareTo(Edge that) {
    return this.weight < that.weight ? - 1 :
           this.weight > that.weight ? + 1 : 0;
  }
}
```

Question 7 Donner le code Java de l'algorithme de Kruskal sous la forme d'une méthode

```
static LinkedList<Edge> kruskal(Graph g)
```

Correction : Il n'y a pas de difficulté ici : il suffit de retranscrire l'algorithme.

```
static LinkedList<Edge> kruskal(Graph g) {
  LinkedList<Edge> mst = new LinkedList<>();
  UF uf = new UF(g.N);
  PriorityQueue<Edge> q = new PriorityQueue<>();
  q.addAll(g.allEdges());
  while (mst.size() < g.N - 1) {
    Edge e = q.remove();
    if (uf.find(e.src) == uf.find(e.dst)) continue;
    uf.union(e.src, e.dst);
    mst.add(e);
  }
  return mst;
}
```

Note : on pourrait être tenté d'ajouter `!q.isEmpty() &&` au test de la boucle mais c'est en fait inutile. Cf question 9.

Complexité et correction de l'algorithme de Kruskal.

Question 8 Quelle est la complexité de l'algorithme de Kruskal, en temps et en espace, dans le pire des cas, en fonction du nombre N de sommets ?

Correction : Soit N le nombre de sommets et A le nombre d'arcs de G. Le coût en espace est clairement $O(A)$, car la file contient initialement tous les arcs de G, c'est-à-dire $O(N^2)$.

Pour ce qui est du temps, dans le pire des cas, tous les arcs de G sont insérés et retirés de la file de priorité, soit A tours de boucle. Chaque retrait de la file coûte $\log(A)$ et chaque opération union-find coûte $\alpha(N)$ (amorti), ce qui est dominé par $\log(A)$. Le coût total est donc $O(A\log(A))$, c'est-à-dire $O(N^2\log(N))$ car $\log(A) = O(\log(N^2)) = O(\log(N))$.

Question 9 Justifier que l'algorithme de Kruskal termine toujours et que l'étape 4a de l'algorithme n'échoue jamais sur une file vide.

Correction : À chaque tour de boucle, la file Q contient un arc de moins. On finira donc forcément par sortir de la boucle ou échouer sur l'étape 4a parce que Q est vide. Montrons que ce second cas ne peut arriver.

Par construction, l'ensemble d'arcs T ne contient jamais de cycle. C'est donc à chaque instant un ensemble de sous-graphes connexes acycliques disjoints les uns des autres. Si on parvenait en 4a avec une file vide, alors cela veut dire que tous les arcs ont été considérés. Mais le graphe de départ étant supposé connexe, l'ensemble T est forcément connexe, donc réduit à un seul graphe acyclique. C'est donc un arbre couvrant de G. Mais la question 1 nous dit alors que $|T| = N - 1$, ce qui contredit le fait d'être encore dans la boucle.

Question 10 Justifier que l'algorithme de Kruskal renvoie bien un arbre couvrant minimal.

Correction : Commençons par montrer que le résultat est bien un arbre couvrant. On a vu à la question précédente qu'on termine nécessairement avec $|T| = N-1$. Si T contenait plusieurs composantes, alors chacune de ces composantes contiendrait $C-1$ arcs dès lors qu'elle contient C sommets (même raisonnement qu'à la question 1) et donc T ne pourrait contenir $N-1$ arcs au total. L'ensemble T est donc bien un arbre. C'est en particulier un arbre couvrant de ses propres sommets, ce qui signifie qu'ils sont au nombre de N. L'ensemble T est donc bien un arbre couvrant de G.

Reste à montrer qu'il est minimal. Pour cela, on montre l'invariant suivant pour la boucle « tant que » : l'ensemble T est un sous-ensemble d'un arbre couvrant minimal. C'est vrai initialement, car $T = \emptyset$. Supposons l'hypothèse vérifiée à une étape arbitraire de l'algorithme, c'est-à-dire que T est contenu dans un arbre couvrant minimal M, et considérons l'arc $x - y$ suivant à sortir de la file. On distingue plusieurs cas :

— Si l'arc n'est pas ajouté à T, l'hypothèse reste trivialement vraie.

— Si l'arc est ajouté à T et faisait déjà partie de M, alors l'hypothèse est préservée (avec le même M).

— Si enfin l'arc est ajouté à T mais ne faisait par partie de M, alors $M \cup \{x-y\}$ contient un cycle passant par l'arc $x-y$. Il existe donc un arc a qui relie la composante (actuelle) de x à celle de y. Cet arc n'a pas encore été considéré (sinon, les composantes de x et y seraient déjà réunies) et donc son poids est supérieur ou égal à celui de l'arc $x-y$. Dès lors, l'ensemble $M' = M\backslash\{a\} \cup \{x-y\}$ est un arbre couvrant de poids inférieur ou égal à M et contenant le nouvel arc.

À l'issue de l'algorithme, T est donc contenu dans un arbre couvrant minimal. Comme il a déjà été prouvé que c'est un arbre couvrant, c'est donc un arbre couvrant minimal.

---

2 Recherche d'un mot dans un texte

Dans ce problème, on manipule des chaînes de caractères. Une chaîne S a une longueur notée $|S|$ (en Java, S`.length()`) et des caractères indexés par les entiers $0, 1, \ldots, |S| - 1$. On note S_i le caractère d'indice i de la chaîne S (en Java, S`.charAt(`i`)`), pour $0 \leq i < |S|$. Pour $0 \leq i \leq j \leq |S|$, on note $S[i..j]$ la sous-chaîne de S constituée des caractères d'indices i inclus à j exclu (en Java, S`.substring(`i`,`j`)`).

Dans tout ce problème, on s'intéresse à la recherche de toutes les occurrences d'une chaîne P (pour *pattern*), de longueur M, dans une chaîne T (pour *text*), de longueur N. Il y a une occurrence de P dans T à la position i lorsque $0 \leq i \leq N - M$ et $T[i..i + M] = P$. Nous allons étudier plusieurs algorithmes. À chaque fois, l'objectif est d'écrire une méthode qui affiche les positions de toutes les occurrences de P dans T. Par exemple, pour $P =$ `"bra"` et $T =$ `"abracadabra"`, la sortie du programme devra être la suivante :

```
occurrence à la position 1
occurrence à la position 8
```

Question 11 Écrire une méthode `static search(String p, String t)` la plus simple possible qui affiche toutes les occurrences de `p` dans `t`. Donner sa complexité en fonction de M et N.

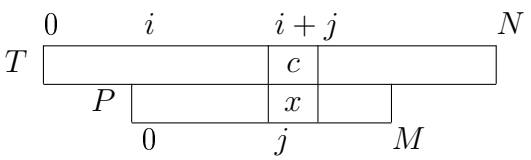
Correction :

```
static void search(String p, String t) {
  for (int i = 0; i <= t.length() - p.length(); i++)
    if (t.substring(i, i + p.length()).equals(p))
      System.out.println("occurrence à la position " + i);
}
```

La complexité est clairement $O(NM)$.

Pré-traitement de P. Lorsque l'on cherche toujours la même chaîne P dans plusieurs textes, ou dans un texte très grand, il peut être avantageux d'effectuer un pré-traitement sur la chaîne P pour accélérer sa recherche. Un algorithme qui exploite cette idée est l'*algorithme de Boyer-Moore*. Les idées sont les suivantes :

- On va tester l'occurrence de P dans T à des positions i de plus en plus grande, en partant de $i = 0$.
- Pour une position i donnée, on va comparer les caractères de P et de T *de la droite vers la gauche*, c'est-à-dire en comparant d'abord P_{M-1} et T_{i+M-1}, puis P_{M-2} et T_{i+M-2}, etc.
- Si tous les caractères coïncident, on a trouvé une occurrence. Sinon, soit j l'indice de la première différence, c'est-à-dire le plus grand entier tel que $0 \leq j < M$ et $P_j \neq T_{i+j}$. Appelons c le caractère T_{i+j}.



L'idée de l'algorithme de Boyer-Moore consiste à augmenter alors la valeur de i de
— la grandeur $j - k$ où k est le plus grand entier tel que $0 \leq k < j$ et $P_k = c$, si un tel k existe (de manière à amener un caractère c sous le caractère T_{i+j}),
— la grandeur $j + 1$ sinon.
Plutôt que de rechercher un tel k à chaque fois, on peut pré-calculer une *table de décalages* contenant, à la case indexée par l'entier j et le caractère c, le plus grand entier k tel que $0 \leq k < j$ et $P_k = c$ s'il existe, et rien sinon.

Question 12 Donner la table de décalages pour la chaîne $P =$ `"banane"`, sous la forme

	a	b	e	n
0				
1				
2				
3				
4				
5				

Certaines cases contiennent une valeur pour k, d'autres restent vides.

Correction :

	a	b	e	n
0				
1		0		
2	1	0		
3	$1^\star$	0		2
4	3	0		$2^\star$
5	3	0		4

Note : les deux entrées marquées d'un étoile peuvent être supprimées en pratique, car elles correspondent à des cas où $P_k = P_j$. Du coup, on ne fera jamais un tel décalage. On économise de l'espace à ne pas les renseigner.

Mise en œuvre en Java. Pour mettre en œuvre l'algorithme de Boyer-Moore en Java, on se donne la classe suivante :

```
class BoyerMoore {
  String P;   // la chaîne que l'on cherche
  int    M;   // sa longueur
  HashMap<Character, Integer>[] table; // la table de décalages
```

La table de décalages est ici représentée par un tableau, de taille M, contenant des tables de hachage. Pour un j donné, avec $0 \leq j < M$, la table de hachage `table[`j`]` associe à un caractère c le plus grand k tel que $0 \leq k < j$ et $P_k = c$, s'il existe.

Question 13 Compléter le constructeur de la classe `BoyerMoore` pour qu'il remplisse la table de décalages.

```
BoyerMoore(String P) {
  this.P = P;
  this.M = P.length();
  this.table = new HashMap<Character, Integer>[M];
  ...
}
```

(En toute rigueur, il faudrait écrire `new HashMap[M]` car il s'agit d'un tableau de génériques, mais ce détail ne nous intéresse pas ici.)

Correction : Il ne faut pas oublier de créer les M tables de hachage avant de les remplir.

```
for (int j = 0; j < M; j++) {
  this.table[j] = new HashMap<Character, Integer>();
  for (int k = 0; k < j; k++)
    if (this.P.charAt(k) != this.P.charAt(j)) // OPTIM
      this.table[j].put(this.P.charAt(k), k);
}
```

Note : La ligne marquée `OPTIM` peut être supprimée. Cf le commentaire dans la correction de la question 12.

Question 14 Dans la classe `BoyerMoore`, écrire une méthode `void search(String text)` qui implémente l'algorithme de Boyer-Moore et affiche toutes les occurrences de `this.P` dans `text` sous la forme demandée.

Correction : On commence par se donner une fonction calculant le décalage :

```
private int shift(char c, int j) {
  if (this.table[j].containsKey(c))
    return j - this.table[j].get(c);
  else
    return j + 1;
}
```

Puis on écrit le code de `search` en suivant l'algorithme donné plus haut :

```
void search(String text) {
  int N = text.length(), i = 0;
  while (i <= N - M) {
    int j = M - 1;
    while (j >= 0 && text.charAt(i + j) == p.charAt(j)) j--;
    if (j >= 0)
      i += shift(text.charAt(i + j), j);
    else {
      System.out.println("occurrence à la position " + i);
      i += 1;
    }
  }
}
```

Question 15 Discuter la complexité en temps de l'algorithme de Boyer-Moore dans le cas où il n'y a aucune occurrence de P dans T. On essayera d'identifier un meilleur cas et un pire cas.

Correction : Dans le pire des cas, le décalage est systématiquement fait pour $j = 0$. C'est le cas par exemple si on recherche le mot $P = ABBB...BB$ dans le texte $T = BBBB...BB$. La complexité est alors $O(NM)$; elle n'est pas meilleure que celle d'une recherche naïve.

Dans le meilleur des cas, le décalage vaut M et il est effectué dès la toute première comparaison. C'est par exemple le cas si on recherche le mot $P = AAA...AA$ dans le texte $T = BBBB...BB$. La complexité est alors $O(N/M)$; on ne peut pas faire mieux.

Question 16 Quelle est la place mémoire occupée par la table de décalages ? Là encore, on essayera d'identifier un meilleur cas et un pire cas.

Correction : Si les M caractères de P sont tous distincts, la table a une taille en $O(M^2)$. C'est clairement un majorant.

Si en revanche les M caractères de P sont tous égaux, alors la table est vide. Elle occupe donc un espace $O(M)$, car il s'agit d'un tableau de taille M contenant des tables de hachage toutes vides. C'est clairement un minorant.

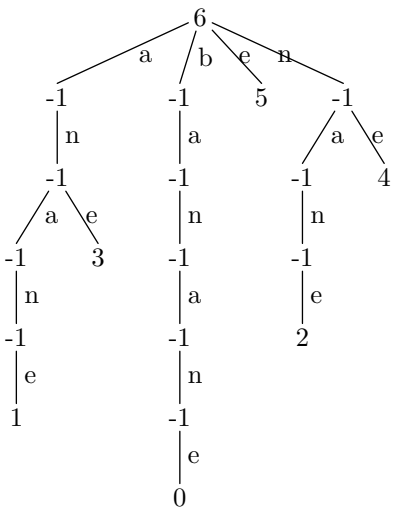
Pré-traitement de T. Lorsque l'on cherche plusieurs chaînes dans un même texte T, il peut être avantageux d'effectuer au contraire un pré-traitement sur le texte T pour accélérer la recherche. Une solution consiste à construire un arbre de préfixes (*trie* en anglais, cf amphi 6) contenant tous les *suffixes* de la chaîne T. Pour représenter cet arbre, on se donne une classe analogue à celle vue en cours :

```
class Trie {
  int suffix; // -1 si pas un suffixe, son indice sinon
  HashMap<Character, Trie> branches;
  ...
}
```

La seule différence est qu'au lieu d'un booléen, pour indiquer la présence du mot, on utilise un entier (champ `suffix`). Si cet entier vaut -1, cela signifie que le mot correspondant à ce nœud de l'arbre n'est pas un suffixe de T. Sinon, c'est un entier i qui signifie que le mot est le suffixe $T[i..N]$, avec $0 \leq i \leq N$.

Question 17 Dessiner l'arbre des suffixes du mot $T =$ `"banane"`. (Les entiers $0, 1, \ldots, 6$ correspondant aux sept suffixes de T doivent tous apparaître dans l'arbre, chacun une seule fois.) Expliquer comment cet arbre permet de trouver facilement toutes les occurrences d'un mot donné dans T. L'illustrer avec le mot `"an"`.

Correction :



Pour trouver les occurrences d'un mot donné P, il suffit de chercher le nœud correspondant à P dans cet arbre. S'il n'y a pas de tel nœud, c'est qu'il n'y a pas d'occurrence de P dans T. Sinon, les occurrences de P dans T sont exactement tous les indices qui se trouvent dans le sous-arbre issu de ce nœud. Dans l'exemple de $P =$ `"an"`, le sous-arbre contient les indices 1 et 3, qui sont bien les positions des deux occurrences de `"an"` dans `"banane"`.

Question 18 Écrire une méthode `void search(Trie st, String p)` qui affiche toutes les occurrences de `p` dans le texte dont l'arbre des suffixes `st` est passé en argument. (L'arbre `st` est supposé non `null`, car un arbre des suffixes n'est jamais vide.)

Correction : On commence par chercher le nœud de `st` correspondant au mot `p`.

```
void search(Trie st, String p)} {
  for (int i = 0; i < p.length(); i++) {
    st = st.branches.get(s.charAt(i));
    if (st == null) return;
  }
```

puis on parcourt tout le sous-arbre correspondant avec une méthode `traverse`

```
  traverse(st);
}
```

dont le code, récursif, est le suivant :

```
void traverse() {
  if (this.suffix >= 0)
    System.out.println("occurrence à la position " + this.suffix);
  for (SuffixTreeNode t: this.branches.values())
    t.traverse();
}
```

Question 19 Quelle est la complexité d'une recherche lorsque le mot P n'apparaît pas dans T ?

Correction : La recherche du nœud correspondant au mot P a une complexité au pire $O(M)$: une descente dans l'arbre par caractère et la recherche dans la table de hachage `branches` se fait à chaque fois en $O(1)$.

Si le mot P n'apparaît pas dans T, cette descente va se terminer prématurément sur l'arbre `null` et la recherche sera terminée. La complexité est donc $O(M)$, indépendamment de la taille du texte.

Question 20 Une construction de l'arbre des suffixes par insertion successive de tous les suffixes $T[i..N]$ prend clairement un temps $O(N^2)$. Qu'en est-il en revanche de l'espace mémoire occupé par l'arbre des suffixes au final ? On essayera d'identifier un meilleur cas et un pire cas.

Correction : Dans le meilleur des cas, tous les caractères de T sont identiques ($T = AAA\ldots AA$) et l'arbre a une taille $O(N)$. Il ne peut être plus petit, car il doit contenir tous les entiers $0, 1, \ldots, N$.

Dans le pire des cas, tous les caractères de T sont distincts et l'arbre a une taille $O(N^2)$. Il ne peut être plus grand, car le temps de sa construction n'excède pas $O(N^2)$.

A Bibliothèque standard Java

`class LinkedList<E>`	
`void add(E e)`	ajoute l'élément `e` à la fin de la liste
`int size()`	renvoie le nombre d'éléments de la liste

On peut parcourir tous les éléments d'une liste `l` avec la construction
`for (E x: l) ...`

`class PriorityQueue<E>`	une file de priorité dont les éléments sont de type `E`
`void add(E x)`	ajoute l'élément `x`
`void addAll(LinkedList<E> l)`	ajoute tous les éléments de `l`
`E remove()`	supprime et renvoie le plus petit élément de la file
`boolean isEmpty()`	renvoie `true` si et seulement si la file est vide

Note : la classe `E` doit implémenter l'interface `Comparable<E>`

`class String`	le type des chaînes de caractères
`int length()`	la longueur de la chaîne
`char charAt(int i)`	le caractère d'indice `i`, pour $0 \leq$ `i` $<$ `length()`
`String substring(int i, int j)`	la sous-chaîne constituée des caractères `i` inclus à `j` exclu

`class HashMap<K, V>`	un dictionnaire dont les clés sont de type `K` et les valeurs de type `V`
`void put(K k, V v)`	associe la valeur `v` à la clé `k` (en écrasant toute valeur précédemment associée à `k`, le cas échéant)
`V get(K k)`	renvoie la valeur associée à `k`, si elle existe, et `null` sinon
`Collection<V> values()`	renvoie l'ensemble de toutes les valeurs de la table

On peut parcourir toutes les valeurs d'un dictionnaire `d` avec la construction
`for (V v: d.values()) ...`

* *
*