Analyse d'algorithmes et programmation - Examen

Enseignants: Christina Boura, Gaëtan Leurent

3 mai 2017

Durée du contrôle : 2h.

Les documents sont autorisés.

1 Questions

Répondre par **vrai** ou **faux** aux questions suivantes en justifiant votre réponse. Des réponses sans justification ne seront pas prises en compte.

1. Soit $f(n) = an^3$, où a une constante et soit $g(n) = n^3$. Alors, $f(n) = \mathcal{O}(g(n))$.

   Vrai. Évidemment $f(n) \leq ag(n)$, pour tout $n \geq 1$, donc la définition est vérifiée pour $n_0 = 1$ et $c = a$.

2. On suppose que $f(n) = \mathcal{O}(n)$ et que $g(n) = \mathcal{O}(n \log n)$. Alors $f(n) = \mathcal{O}(g(n))$.

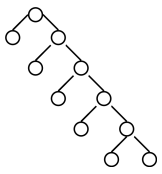
   Faux. Par exemple si $f(n) = n$ et $g(n) = \log n$, alors f et g vérifient les contraintes (puisque la notation $\mathcal{O}$ ne donne qu'une borne supérieure, mais évidemment, $n \neq \mathcal{O}(\log n)$.

3. Le tri par fusion sur un tableau de taille n déjà trié, se fait en temps $\mathcal{O}(n)$.

   Faux. Le tri par fusion découpe et fusionne toujours le tableau $\mathcal{O}(\log n)$ fois, donc le tri se fasse en $\mathcal{O}(n \log n)$ quoi qu'il arrive.

4. Si chaque nœud d'un arbre binaire de recherche contenant n éléments a soit 0 soit exactement 2 fils, alors l'hauteur de l'arbre est $\Theta(\log n)$.

   Faux. L'arbre de la figure ci-dessous, vérifie ces conditions, mais son hauteur est $\Theta(n)$.

5. L'insertion d'un élément dans un arbre binaire de recherche ayant n nœuds se fait toujours en $\mathcal{O}(\log n)$.

   Faux. L'insertion d'un élément dans un arbre binaire de recherche se fait en $\mathcal{O}(h)$, où h est la hauteur de l'arbre. Si l'arbre n'est pas équilibré, h peut être beaucoup plus grand que $\log n$. Dans le cas extrême, $h = n - 1$.

6. Pour une table de hachage de taille n employant la méthode de chaînage pour résoudre les collisions et contenant n éléments, on peut toujours trouver le plus petit élément en temps $\mathcal{O}(1)$.

   Faux. On a besoin de parcourir toute la table, ce qui nécessite toujours un temps linéaire en n dans le meilleur des cas.

7. L'implantation d'un ensemble dynamique avec une table de hachage garantit toujours un temps de recherche en $\mathcal{O}(1)$.

   Faux. Tout depend de la fonction de hachage utilisée, et la taille de la table par rapport au nombre d'éléments insérés. Par exemple, dans le cas extrême où tous les éléments sont hachés vers la même case, le temps de recherche sera en $\mathcal{O}(n)$, qui est le temps de parcourir la liste chaînée engendrée.

8. Lorsqu'on double la taille d'une table de hachage, on peut continuer à utiliser la même fonction de hachage.

   Faux. Si la taille de la table est doublée, on doit modifier la fonction de hachage afin que l'ensemble d'arrivé soit maintenant $\{0, \ldots, 2m-1\}$, à la place de $\{0, \ldots, m-1\}$ qui était l'ensemble d'arrivé précédent. Si vous avez répondu "vrai", en expliquant que c'est possible mais que ce n'est pas optimal, la réponse a été considérée comme correcte également.

2 Tri

Étant donnés k tableaux déjà triés, de taille n chacun, décrire un algorithme en temps $\mathcal{O}(nk \log k)$ qui fusionne ces k tableaux en un seul tableau trié.

On a vu pendant l'étude du tri par fusion une procédure qui permet de fusionner deux listes déjà triées en une seule liste triée en temps linéaire. Cette procédure peut être la base de notre algorithme, décrit ci-dessous. Cet algorithme récursif prend en entrée k listes L_1, ..., L_k de n éléments chacune.

```
Algo(L1, ..., Lk)
   si k = 1
       renvoyer la liste triee
     /* Appeler l'algorithme recursivement sur les k/2 premieres listes ..
         et ensuite sur les k/2 dernieres */
    Algo(L1, ..., L(k/2))
    Algo(L1, ..., L(k/2 + 1))
    Fusionner les deux listes finales avec l'algorithme vu en cours.
```

Il y a $\mathcal{O}(\log k)$ niveaux récursifs et chaque niveau coûte $\mathcal{O}(nk)$. Donc le temps final sera en $\mathcal{O}(nk \log k)$.

3 Structures de données

On suppose qu'on a une liste d'entiers non triés. On veut trouver et imprimer tous les entiers qui sont présents plus d'une fois dans la liste. Par exemple, étant donnée la liste $\{1, 5, 3, 5\}$, on doit imprimer 5.

Donner trois algorithms différents pour effectuer cette tâche. Le premier doit être en temps $\mathcal{O}(n^2)$, le deuxième en temps $\mathcal{O}(n \log n)$ et le troisième en temps $\mathcal{O}(n)$. Vous pouvez écrire du pseudo-code (sans pour autant entrer trop dans les détails) ou simplement décrire dans le language courant vos algorithmes.

Remarque : Les trois algorithmes peuvent être réalisés avec les algorithmes et les structures de données vus en cours.

$\mathcal{O}(n^2)$ L'algorithme suivant employant une double boucle se fait en $\mathcal{O}(n^2)$. Il prend en entrée une liste L d'éléments et ajoute tous les doublons dans une liste T initialement vide.

```
RechercheDoublons(L)
   T = []
   pour i de 1 à len(L)
      pour j de 1 à len(L)
         si L[i] = L[j]
             ajouter i dans T
```

$\mathcal{O}(n \log n)$ Une solution est de trier la liste avec un algorithme de tri en $\mathcal{O}(n \log n)$ (tri-rapide, tri-fusion ..) et ensuite rechercher les doublons avec un algorithme qui se déplace dans la liste en temps $\mathcal{O}(n)$.

$\mathcal{O}(n)$ Insérer les éléments de la liste dans une table de hachage. Un doublon est détecté lors de l'insertion d'un élément dans une case non-vide. Chaque insertion prend un temps $\mathcal{O}(1)$, donc pour n éléments le temps est en $\mathcal{O}(n)$.

4 Une nouvelle structure de données

On considère une nouvelle structure de données qui peut être utilisée comme une alternative aux arbres binaires de recherche. Cette structure de données stockant n éléments peut être représentée comme une matrice carrée de taille $\sqrt{n} \times \sqrt{n}$. On suppose que $\sqrt{n}$ est un nombre entier. Chaque colonne et ligne de la matrice sont triés par ordre croissant (voir la figure ci-dessous pour un petit exemple). Le plus petit élément se trouve toujours en haut et à gauche de la matrice et l'élément le plus grand se trouve toujours en bas et à droite de la matrice.

M	1	2	3	4	5
1	2	4	5	7	10
2	3	6	8	15	17
3	14	19	25	32	39
4	18	26	34	41	47
5	21	28	36	45	52

Ci-dessous est donné le pseudo-code d'un algorithme `Recherche` qui permet de chercher un élément de clé k dans la structure de données M. Pour simplifier le raisonnement, on suppose qu'on ne stocke que les clés, sans données associées. Chaque élément consiste donc en un entier. On note $l = \sqrt{n}$ le nombre de lignes et des colonnes de la matrice. Par $M[r, c]$ on désigne l'élément à l'intersection de la ligne r et de la colonne c de la matrice M. L'algorithme renvoie `Vrai` si l'élément k se trouve dans la matrice et renvoie `Faux` sinon.

```
Recherche(M, l, k)
   r <- l
   pour c de 1 à l
      tant que M[r,c] > k ET r > 1
         r <- r - 1
      si M[r,c] = k
         renvoyer "VRAI"
   renvoyer "FAUX"
```

1. Montrer en détail les étapes que cet algorithme effectue pour la recherche de la clé 15 dans la matrice ci-dessus.

   L'algorithme compare 15 à $21, 18, 14, 19, 6, 8, 15$ et renvoie vrai.

2. Expliquer comment cet algorithme marche de façon générale.

   L'algorithme commence la recherche au coin en bas et à gauche de la matrice et se déplace ensuite verticalement jusqu'à ce qu'un élément plus petit que k est trouvé. Ensuite il se déplace vers la droite jusqu'à ce qu'un élément plus grand que k est trouvé. Cette procédure est répétée jusqu'à ce que l'élément est trouvé ou jusqu'à ce que les limites hautes et droites de la matrice sont atteintes.

3. Quelle est la complexité de cet algorithme pour une structure de données contenant n éléments ? Donner votre réponse en notation $\mathcal{O}$. Justifier.

   La complexité est en $\mathcal{O}(\sqrt{n})$ puisque l'algorithme se déplace seulement vers le haut et la droite de la matrice. Par conséquent, il y a au plus $\sqrt{n}$ déplacements vers le haut et au plus $\sqrt{n}$ déplacements vers la droite. La complexité par étape est constante, car il n'y qu'une comparaison à faire.

4. Comparer cette complexité avec la recherche d'un élément dans un arbre binaire de recherche équilibré de taille n. La complexité de cet algorithme est pire qu'avec un arbre binaire de recherche équilibré, puisque $\sqrt{n}$ a une croissance plus rapide que $\log n$.

5 Graphes

Dans cette partie, nous allons étudier deux problèmes d'optimisation sur un graphe non orienté (non pondéré) $G = (S, A)$.

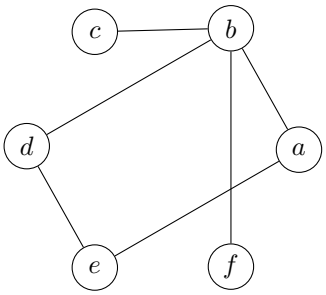


FIGURE 1 – Graphe G_1.

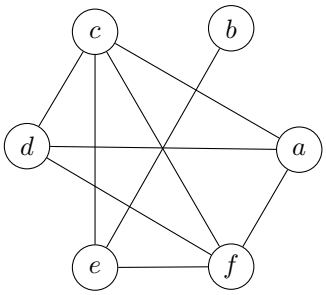


FIGURE 2 – Graphe G_2.

Un ensemble de sommet $T \subset S$ est appelé une *couverture* si T contient au moins une extremité de chaque arrête du graphe. Formellement,

$$T \text{ est une couverture de } G \iff \forall (u,v) \in A, u \in T \text{ ou } v \in T$$

Le problème de *couverture minimum* consiste à trouver une couverture de cardinalité minimale.

Un ensemble de sommet $T \subset S$ est appelé une *clique* si il y a une arrête dans A entre chaque paire de sommets de T. Formellement,

$$T \text{ est une clique de } G \iff \forall u \neq v \in T, (u,v) \in A$$

Le problème de *clique maximum* consiste à trouver une clique de cardinalité maximale.

Exercice 1 (1 point)
Trouver une couverture minimum pour le graphe G_1, et une clique maximum pour le graphe G_2.
***Réponse** : $\{b, e\}$ est une couverture minimum de G_1. $\{a, c, d, f\}$ est une clique maximum de G_2.*

Exercice 2 (1 point)
Definir le problème de décision COUVERTURE *correspondant au problème de couverture minimum, et le problème de décision* CLIQUE *correspondant au problème de clique maximum.*

Expliquer pourquoi ces problèmes sont dans la classe NP.
***Réponse** : Le problème* COUVERTURE *est défini ainsi : étant donné un graphe G et un entier k, décider s'il existe une couverture de taille inférieure ou égale à k.*
Le problème CLIQUE *est défini ainsi : étant donné un graphe G et un entier k, décider s'il existe une clique de taille supérieure ou égale à k.*
Ces problèmes sont dans la classe NP car on peut utiliser une couverture (respectivement une clique) comme certificat, et il est facile de vérifier en temps polynomial si un ensemble de sommets est bien une ouverture (respectivement une clique).

On défini le complémentaire $\overline{G} = (S, \overline{A})$ d'un graphe $G = (S, A)$, comme le graphe dont l'ensemble des arrêtes est le complémentaire de A :

$$\forall u \neq v \in S,\ (u,v) \in \overline{A} \Leftrightarrow (u,v) \notin A$$

Par exemple, G_1 est le complémentaire de G_2 (et réciproquement).

Exercice 3 (2 points)
Montrer que G possède une couverture de taille t si et seulement si $\overline{G}$ possède une clique de taille $|S| - t$.
***Réponse** : Soit X une couverture de G de taille t. Par définition d'une couverture, il ne peut pas y avoir d'arrête entre les sommets de $\overline{X}$. Ainsi, $\overline{X}$ est une clique de taille $|S| - t$ dans le graphe $\overline{G}$.*

Réciproquement, soit Y une clique de G de taille t. Par définition d'une clique, il y a une arrête entre chaque paire de sommet de Y dans A. Alors il n'y a pas d'arrête entre les sommets de Y dans $\overline{A}$, et $\overline{Y}$ est une couverture de $\overline{G}$ de taille $|S| - t$.

Exercice 4 (2 points)
Montrer que COUVERTURE *est un problème NP-Complet, en utilisant le fait que* CLIQUE *est un problème NP-Complet.*
***Réponse** : On construit une réduction de* CLIQUE *à* COUVERTURE*. Étant donné une instance G, k du problème* CLIQUE*, on construit une instance $\overline{G}, |S| - k$ de* COUVERTURE*. D'après l'exercice précédent, cette instance de* COUVERTURE *a une solution si et seulement si l'instance de* CLIQUE *a une solution.*

On en déduit que CouvertureGlouton *est NP-Complet.*

On propose un algorithme glouton pour trouver une couverture minimum :

```
1: fonction CouvertureGlouton(G = (S, A))
2:     T ← ∅
3:     tant que A ≠ ∅ faire
4:         Choisir une arrête (u, v)
5:         Ajouter u et v à T
6:         Retirer de A toutes les arrêtes ayant u ou v comme extremité
7:     retourner T
```

Exercice 5 (2 points)
Quelle est la complexité de CouvertureGlouton *?*
Pensez-vous que cette algorithme trouve une couverture minimum ? Expliquer pourquoi, et donner une preuve ou un contre-exemple.
***Réponse** : L'algorithme glouton fait $\mathcal{O}(|A|)$ opérations dans le pire des cas, car on retire au moins une arrête à chaque itération.*

C'est un algorithme polynomial pour un problème NP-Complet, donc il ne peut pas trouver de solution optimale (à moins de prouver $P = NP$).

Concrètement, sur le graphe G_1, l'algorithme glouton va choisir deux arrêtes, et renvoyer une couverture de taille 4 (par exemple $\{c, b, e, a\}$), alors que la couverture minimum est de taille 2.

Exercice 6 (2 points)
Montrer que la couverture renvoyée par CouvertureGlouton *contient au pire deux fois plus de sommets qu'une couverture minimum.*
***Réponse** : Soit T une couverture minimum. À chaque fois que l'algorithme glouton choisi une arrête (u, v), une des deux extrémités doit être dans T (puisque c'est une couverture). De plus, les arrêtes choisies ne partagent jamais une extrémité, donc on peut associer un sommet distinct de T à chaque arrête choisie. On en déduit que la couverture construite par* CouvertureGlouton *est de taille au plus $2|T|$.*

Exercice 7 (3 points)
Donner un algorithme polynomial pour construire une couverture minimum, dans le cas particulier où le graphe est acyclique.
Indice : on pourra monter que dans un graphe acyclique avec un ensemble d'arrêtes non vide, il existe au moins un sommet qui est incident à une seule arrête.
***Réponse** : Supposons que G soit un graphe acyclique avec un ensemble d'arrêtes non vide, dans lequel tous les sommets ont soit aucune arrête incidente, soit au moins 2. Soit (u, v) une arrête de G. On va construire un chemin dans le graphe en partant de u. À chaque étape, on choisi une arrête sans revenir en arrière ; c'est possible car tout les sommet avec une arrête incidente ont au moins deux arrêtes incidentes. Comme le graphe est fini, ce chemin doit revenir sur un sommet déjà rencontré, et former un cycle. On a donc montré par l'absurde qu'un graphe acyclique avec un ensemble d'arrêtes non vide possède au moins un sommet avec une seule arrête incidente.*

On peut maintenant décrire un algorithme pour construire une couverture minimum :

```
1: fonction CouvertureAcyclique(G = (S, A))
2:     T ← ∅
3:     tant que A ≠ ∅ faire
4:         Choisir un sommet u avec une seule arrête incidente (u, v)
5:         Ajouter v à T
6:         Retirer de A toutes les arrêtes ayant v comme extremité
7:     retourner T
```