Université Paris 7 - Licence d'informatique - L2

EA4 – ERREURS FRÉQUENTES de l'examen du 30 Mai 2012

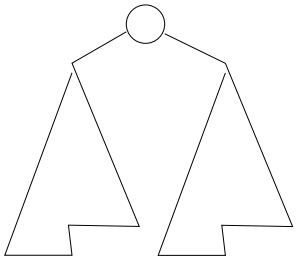
Exercice 1.

- Question 1.1. L'erreur la plus fréquente (la quasi-totalilté des réponses) a été de dire : "on a le cas meilleur quand on ne rentre jamais dans la boucle et donc quand n est -1, 0 ou 1" !.
  Si cet argument était sensé, alors, pour n'importe quel algorithme, on aurait toujours le meilleur cas quand la taille de la donnée est la plus petite possible ! !
  Si ceci était vrai, l'algorithmique n'aurait aucun intérêt et elle n'existerait sans doute même pas comme branche de l'informatique. L'algorithmique ne s'occupe pas du tout de ce que font les algos pour des petites valeurs de la taille de la donnée, et quand elle se pose le problème de déterminer le meilleur et le pire de cas d'un algo, elle le fait pour une taille de la donnée quelconque (vous n'avez pas le droit de la fixer).
- Autre erreur : dire que on a le meilleur cas quand n est pair, alors que ceci ne suffit pas. Il faut que n soit une puissance de 2. Il en a qui ont dit que un nombre pair divisé par 2, donne un nombre pair... Ah bon ? $10006/2 = 5003$ est pair ?
- Question 1.2. Erreur très fréquente : dire que n doit être impair et "grand" ou "très grand". Ceci n'a pas de sens. Est ce que 1000 est grand ? Sans doute oui relativement à 1, mais pas relativement à 1000000. Est ce que $1000000 = 10^6$ est grand ? Sans doute oui relativement à 1000, mais pas relativement à 10^{12}.
- "On a le cas le pire quand n est infini". Aux dernières nouvelles, les ordinateurs ne manipulent que des nombres finis et bornés.
- "On prend le plus grand entier impair : 127". En Java, l'intervalle $[-128, 127]$ est le domaine du type `byte` (représentés sur un octet). Même le type `short` a comme domaine $[-32768, +32767]$ (sur 2 octets), alors que `int` a comme domaine $[-2147483648, +2147483647]$ (sur 4 octets). Vous vous imaginez si on devait être limités à manipuler uniquement des `int` inférieurs à 127 ? Avez-vous déjà programmé ?
- Ne pas savoir évaluer combien d'entiers d'un intervalle $[0, M]$ sont des puissances de 2.

Exercice 2.

- Oublier le cas de base où l'arbre est vide (A = NULL). Ceci risque de donner des bugs catastrophiques quand vous traduisez vos algos en programmes (ou bien de donner des résultats faux), puisque par exemple `A->fg` et `A->fd` n'existent même pas . Cette erreur a été commise par la quasi-totalité des étudiants.
- Ne pas respecter les types des données. Un arbre n'est PAS un noeud, mais un pointeur sur un noeud (l'adresse d'un noeud). En conséquence, par convention, on accède aux attributs par l'opérateur `->` et pas . (point). Par exemple : `A->fg` et non `A.fg`.
- Ne pas respecter le type de l'objet retourné, booléen ou entier, comme demandé par le sujet.
- Ne pas "combiner" correctement (dans notre cas avec l'opérateur booléen ET) les valeurs de retour des appels récursifs la fonction de la première question 2.1. La plupart ont fait deux appels séquentiels `ADeuxFils(A->fg)` suivi de `ADeuxFils(A->fd)`, sans combiner avec un "ET" les deux résultats, et sans comprendre que dans un tel cas, l'algorithme va retourner la réponse relative seulement à l'arbre de gauche.
- Utiliser une multitude de variables auxiliaires inutiles soit entières (`profondeur, hcourante,`

`x, y, res, result, ...`) soit booléennes. Toutes ces variables ne font que rendre l'algorithme incompréhensible par le correcteur dans le meilleur cas, et totalement faux dans le pire.
- Ecrire des algorithmes tellement alambiqués sans comprendre que on ne vous proposerait JAMAIS des exercices avec des solutions aussi difficiles et que le correcteur ne pourra jamais suivre des raisonnements (si on peut les définir ainsi) aussi confus.
- Justifier que la complexité d'un algorithme est on $O(N)$ seulement par l'argument que on fait un appel récursif par noeud, sans spécifier que chaque appel est en $O(1)$ (erreur très fréquente).
- Pour la question 2.3. Certains ont cru pouvoir tester si un arbre est un tas en testant si le sous arbre gauche et le sous arbre droit sont des tas. Un tel arbre n'est pas un tas (voir figure) :



- D'autres ont testé la condition que le contenu d'un noeud (la valeur du champ `val`) est plus petit que celui de ses deux fils. Celle-ci est la seconde propriété caractérisante des tas (qui porte sur le contenu des noeuds), alors que le sujet demandait de tester seulement la première (celle sur la forme). Le sujet était clair : "Une fonction qui, pour un entier h, retourne h si l'arbre binaire donné a la **forme** d'un tas de hauteur h". Et si ceci n'était pas suffisamment clair, la parenthèse suivante expliquait clairement ce qui était demandé : "(tous les niveaux sont "complets", sauf éventuellement le dernier de profondeur h, dans lequel les noeuds "occupés" sont ceux les plus à gauche)". Le champ `val` n'avait aucun rapport avec le problème.

Exercice 3.

- Beaucoup d'étudiants n'ont pas du tout écrit l'algorithme mais ils ont tout de même (on se demande comment ?) répondu aux questions suivantes. Comment peut-on évaluer la complexité d'un algo si on ne l'a pas décrit précisément ? ? ?
- Beaucoup d'étudiants n'ont pas compris que le seul paramètre de l'algorithme demandé était le tableau et non l'ABR, qui, en tant de structure de donnée **auxiliaire**, devrait être plutôt une variable locale de la fonction. En tout cas, la donnée du problème n'est PAS l'ABR.
- Certains ont donné un algo de tri comme tri rapide, ou tri par sélection, qui n'utilisent pas du tout un ABR et qui donc n'ont aucun rapport avec la question.
- Ne pas avoir compris que l'algorithme demandé devait **renvoyer** le tableau trié et non simplement afficher les entiers contenu dans l'ABR en ordre croissant.
- Ceux qui ont choisi d'implémenter la deuxième partie de l'algo par un parcours préfixe de l'arbre ne peuvent pas simplement dire qu'on appelle la fonction `ParcoursPréfixe` sans donner ses détails (son pseudo-code). En effet, la gestion de l'index de la position du tableau où l'on insère l'élément courant du tableau est loin d'être triviale et doit donc être détaillée.
- Pour les questions de complexité, beaucoup d'étudiants n'ont pas compris que la complexité est liée à la hauteur de l'ABR que l'on construit au fur et à mesure, et que le pire des cas est donné quand l'arbre est assimilable à une liste chaînée (chaque noeud a un seul fils). Ceci arrive notamment, quand le tableau est trié en ordre décroissant **mais aussi** quand il est trié en ordre croissant (et donc déjà trié !).
- Certains étudiants ont dit que l'utilisation d'un AVL rend la complexité plus élevée parce que avec les AVL il faudra effectuer des rotations qui, selon eux, ralentissent l'algo. Ceci est totalement faux, le coût de rotations est constant et donc négligeable pas rapport au gain de temps qu'on aura au total, dû à une hauteur de l'arbre plus petite.