Retour sur le DS3

Ceux qui ont eu plus de 10 à ce devoir, c'est très encourageant !

DM: à faire avant le 23/02, je mettrai le corrigé en ligne sur Moodle. Je suis à disposition pour les questions. Refaites bien le DS avant, en vous aidant du corrigé si besoin, en comprenant vos erreurs et en étant capable de tout rédiger très proprement sans le corrigé (faites le VRAIMENT, ne vous contentez pas de lire le corrigé, ça ne marche)

Stratégie: lire l'énoncé en entier, repérer les questions faciles.
L'exercice 3 était plus facile et vous aviez vu le cours de mathématiques correspondant

Pour certains d'entre vous, la propreté et l'écriture de la copie posent problème. Faites attention.

Côté programmation

Utilisation des références en OCaml : elles ne doivent utilisées que pour les variables qui vont être modifiées au cours de l'exécution
Ayez le réflexe de ne pas modifier les variables d'entrée mais de les recopier dans des variables locales pour les modifier. Cela vous aidera aussi pour faire coller vos démos théoriques à vos algorithme

Confusion entre :
&a opérateur d'adresse, qui s'applique à des variables « classiques »

&: variable → adresse

*a opérateur de déréférencement qui s'applique à des pointeurs (variables contenant des adresses)

*: pointeur → variable pointée

int *tot : déclare un pointeur sur un entier, nommé toto, et alloue l'espace nécessaire (8 octets), dans la pile, dans le bloc d'activation où cette déclaration a été faite. On colle l'étoile au nom du pointeur pour bien voir que
toto est un (int *) (pointeur sur un entier)
(*toto) est un int (déréférencement)

On ne fait jamais de tests d'égalité sur des variables flottantes, à cause de l'imprécision machine
Au mieux, on a 16 chiffres significatifs après la virgule en double précision donc on fait des tests du genre
|a-a0|/|a0| < epsilon

Confusions C et OCaml

!a en OCaml si let ref a = 3 ;

Preuves théoriques

Dans les démonstrations théoriques, on ne doit pas faire apparaître les côtés techniques de l'implémentation (passages par adresse, pointeurs..). Ce sont des aspects liés au langage, à l'implémentation, il faut s'en abstraire. Essayez d'imaginer que vous rédigez la démo à partir du pseudo-code.
Ces démonstrations doivent être en lien fort avec votre algorithme. Elles ne sont pas hors-sol.
Pour chaque technique de démonstration (méthode du variant, récurrence, preuve par invariant de boucle...etc) il faut écrire explicitement la technique utilisée sur la copie pour rassurer le correcteur.

Preuves de terminaison et méthode du variant

Erreur de vocabulaire, en conclusion d'une preuve de terminaison « Donc la fonction est terminale » → Non, **terminal, c'est un adjectif qui s'applique aux fonctions récursives, pour indiquer qu'elles ont été travaillées pour que le bloc d'activation courant puisse être réutilisé et éviter le débordement de pile.**

Dites simplement : « Donc la fonction se termine »

Il faut trouver une quantité, **en lien avec les variables de l'algorithme**, qui diminue au cours de l'algorithme.
Il faut écrire très clairement le variant choisi et ensuite argumenter, prouver que c'est un variant, et montrer en quoi ce variant est en lien avec la terminaison de l'algorithme
Utiliser des noms de suite qui sont en lien avec vos noms de variables pour bien connecter votre démonstration théorique à vos algorithme.
Attention, dans le début de la démo, collez un maximum à l'algorithme: dans le cas de div_euclid, on enlève b à chaque itération donc une définition récurrente est plus adaptée a_0 = a, a_{n+1} = a_{n} -b

Preuves de correction

Dans une démonstration de correction, **il faut que la propriété prouvée soit en lien avec les spécifications de l'algorithme… on ne prouve pas une propriété pour prouver une propriété.**

Démonstration par invariant != démonstration par récurrence
Pour les preuves de correction avec invariant de boucle,
Il faut é**crire très clairement la propriété dont on va montrer qu'elle est un invariant de boucle** :

Exemple : P : pgcd(a0, b0) = pgcd(a, b) où a0 b0 sont les valeurs données en entrée de l'algorithme et a et b le valeurs contenues dans les variables a et b

P n'est pas indexée par un mystérieux « n », ce n'est pas P(n) comme dans les preuves par récurrence

C'est une propriété qui relie les valeurs contenues dans les variables de l'algorithme
Structure de la démo :

- Initialisation : on prouve que P est vrai juste avant la boucle
- Conservation : On suppose que P est vrai au début d'un tour quelconque

  On introduit des notations pour bien différencier les valeurs des variables **en debut de tour** de celles **en fin de tour**

  On écrit des relations entre début de tour et fin de tour en s'appuyant sur les instructions effectuées par l'algorithme

  On essaie de prouver avec ça que P est vrai pour le contenu des variables en fin

de de tour

- Conclusion : montrer en quoi la conservation de la propriété à la toute fin du dernier tour garantit la terminaison de l'algorithme

Expressions de complexité

La complexité d'un algorithme dépend de un, deux ou plusieurs des paramètres d'entrée de l'algorithme et éventuellement d'autres critères mais pour des algorithmes plus complexes.

Il faut d'abord déterminer de quoi dépend la complexité, avant d'écrire un ordre de grandeur asymptotique. Mais qui est « n » ??? Ce n'est pas toujours n, une notation doit avoir un sens.

Ces paramètres dont dépend la complexité doivent bien sûr apparaître dans l'expression asymptotique

C'est souvent à partir du critère d'arrêt de la boucle principale ou du cas de base d'un algo récursif que l'on peut déterminer clarifier :

- de quoi dépend la complexité

- et quand l'algo va s'arrêter en fonction des données d'entrée, ce qui va nous permettre de compter le nombre d'appels récursifs ou de tours de boucles, et ainsi donner une estimation de la complexité

Retour sur les exercices

Exercice 1 : Questions de cours

- Réexpliquer la complexité spatiale des tours de Hanoi : elle est linéaire en le nombre de disques
  → cf Séquence 6, pages 25/26
- Algorithme d'exponentiation rapide itératif : rappeler le principe, montrer les erreurs
  Attention, utiliser l'opérateur d'exponentiation ** dans un algorithme d'exponentiation pose problème… c'est le serpent qui se mord la queue…
  cf Séquence 7 page 15
- Tableau → qui est n??? c'est cela que je demandais, pas de m'expliquer les notations de Landau
  Concaténation de deux listes ? $O(n_1)$ où n_1 est la taille de la première liste

Exercice 2 : Euclide

- Vous pouvez traiter des cas simplifiés si trop compliqué. Ici, ok si vous dites : je suppose dans un premier temps a > 0 pour simplifier
- PGCD positif : sinon ambiguïté !
- Rappel : 0 ne divise personne, il est interdit de diviser par 0
- pgcd(63,11) =1 car 11 est premier… c'est vrai mais pas du tout dans la philo de l'énoncé qui essaie de vous montrer une méthode algorithmique de calcul de PGCD
- Démonstration pgcd(a,b) = pgcd(b,r)
  En 2 étapes
  pgcd(a,b) | pgcd(b,r)
  pgcd(b,r) | pgcd(a,b)
  Donc pgcd(a,b) = ± pgcd(b,r) et on peut trancher car par définition le PGCD est positif

donc ils doivent avoir le même signe donc c'est un +

Exercice 3 : Recherche de racines

- C'était un algorithme dichotomique
- Faire des dessins pour expliquer : TB, mais ne faites pas le dessin d'une fonction affine, car c'est une cas où un tel algorithme n'a aucun intérêt car on a une formule exacte pour la racine
- f(x) =pq+r <=> x_0 = -r/p
- La précision est sur la racine x_0 et non sur f(x_0). Le bon test est |b-a| < epsilon et non f(c) < epsilon
- Pour prendre en compte à la fois le signe de f(c) et le fait que f est croissante ou décroissante, il faut comparer f(c) à l'une des images des bornes, par exemple f(b)
- Variant : c'est la taille l de l'intervalle de recherche qui diminue à chaque itération : l_k = b_k – a_k
- et on montre en utilisant le fait que c_k = (a_k+b_k)/2 que l_{k+1} =l_{k}{2}
- Complexité : elle ne dépend que de a, b et epsilon. C'est la taille de l'intervalle initial de recherche qui joue, et non la position de la racine dans cet intervalle. Le comportement de la fonction n'a pas non plus d'influence. Comme on divise par deux la taille de l'intervalle à chaque étape de l'algorithme, il existe un appel récursif ou une itération n pour laquelle |b-a|/2^n < epsilon
- La complexité est liée à ce nombre d'appels/d'itérations n, donc on isole n dans la relation précédente n > E(log2(|b-a|/epsilon))+1