TD XIII

ULTIME ENTRAÎNEMENT

1 Derniers conseils

1.1 Présentation

- Utilisez une couleur différente pour le code (par exemple, code en noir, texte général en bleu et commentaires inclus dans le code en bleu également).
- Avant de commencer à coder une fonction, assurez-vous qu'elle tiendra sur la page (sans tasser). En cas de doute, commencez le code en haut de la page suivante.
- Indentez, parenthésez...
- Soignez particulièrement votre graphie lorsque vous écrivez du code ; évitez d'avoir une variable m et une variable n si ces lettres ont tendance à se ressembler chez vous.
- Encadrez vos résultats, numérotez clairement les questions et traitez-les dans l'ordre (en vous laissant la place de revenir sur une question initialement sautée).

1.2 Programmation

- Il faut absolument respecter les types donnés par l'énoncé.
- Pas de confusion entre listes et tableaux !
- Allez au plus simple (en respectant bien sûr les contraintes de complexité). Les fonctions récursives terminales sont souvent plus délicates à écrire (et à lire !) que les versions non terminales.
- En général, évitez de mélanger itératif et récursif.
- Ne pas rajouter des cas de base pour le plaisir ! Il arrive que la liste à un seul élément doive être traitée à part, mais le plus souvent les cas `[]` et `x :: xs` suffisent...
- Si vous faites volontairement un `match` non exhaustif, ajouter un cas `... -> failwith "impossible"` rend souvent le code plus clair.
- Ne faites pas deux appels à une même fonction sur un même argument ; nommez le résultat et réutilisez le (`let y = f x in ... y ... y ...`). Ne pas respecter cette règle empêche souvent d'obtenir la complexité souhaitée.
- Essayez d'éviter les `match` imbriqués (ce n'est pas interdit, mais c'est rarement le plus simple et rarement le plus clair).
- Utilisez `match` quand vous voulez « déconstruire » une liste, un arbre... Privilégiez `if...then...else` si vous souhaitez simplement distinguer des cas :

```
if n < 0 then ...
else if n = 0 then ...
else ...
```

- Vous pouvez utiliser `List.iter` et `List.map` sans problème, mais si l'énoncé ne rappelle pas l'existence de ces fonctions c'est probablement qu'on s'en passe très bien. N'utilisez `List.fold_left` que si vous êtes **sûrs** de maîtriser.
- En dehors de `failwith`, n'utilisez pas d'exceptions à CCP, et soyez très prudents à Centrale et aux Mines. Pour X-ENS il n'y a pas de problème, mais il faut savoir ce que l'on fait...
- La plupart des fonctions demandées s'écrivent en moins de 10 lignes. Au delà de 15 lignes, il faut vous poser des questions sur l'approche choisie (ça peut quand même arriver une ou deux fois dans un sujet).

- Les fonctions `List.hd`, `List.tl`, `fst` et `snd` sont rarement utiles : préférez les `match`.
- Spécifier ce que font vos fonctions auxiliaires (sauf si vous êtes **sûrs** que c'est complètement évident).
- Si on ne vous demande pas la complexité, inutile de la donner. Si on vous la demande, il faut la justifier (même très rapidement), sauf mention explicite de l'énoncé.
- **Ne confondez pas OCaml et Python !**

I.3 Questions théoriques

- Commencez par parcourir rapidement l'énoncé et regardez ce que l'on cherche à faire. Si le but final est de trouver un algorithme permettant de résoudre un problème en temps $\mathcal{O}(n^2)$ et que, dans la première partie, vous pensez avoir trouvé un algorithme naïf en $\mathcal{O}(n)$, c'est sans doute qu'il y a un problème.
- Par défaut, toute réponse doit être justifiée (sauf mention explicite de l'énoncé). Si cela vous semble complètement évident, essayez quand même de trouver un argument (une ligne peut suffire).
- Pour les automates, les preuves se font presque toujours par récurrence sur la longueur du mot ou du chemin. Faites des dessins.
- Si vous avez un langage rationnel, il est reconnaissable par un automate **que l'on peut choisir déterministe et complet**. N'hésitez pas à le faire, cela simplifie souvent les preuves.
- Pour les arbres, les preuves se font par récurrence sur la hauteur ou, mieux, par induction structurelle. Faites des dessins.
- Pour les graphes, les preuves se font de différentes manières (mais certainement pas par induction structurelle, cela ne veut rien dire). Faites des dessins.
- Faites des dessins, mais ne vous contentez pas de faire des dessins.
- Attention au sens des inégalités (cela ne vaut pas qu'en informatique) !
- Pour la complexité spatiale des fonctions récursives non terminales, n'oubliez pas la place sur la pile d'appels (concerne uniquement les sujets X-ENS, normalement).
- La complexité spatiale est toujours inférieure ou égale à la complexité temporelle.
- La complexité temporelle d'un appel `u @ v` est proportionnelle à la longueur du u.
- Méfiez-vous des affirmations péremptoires « le meilleur/pire des cas est celui où. . . » : même si c'est juste, on attend normalement une justification (c'est-à-dire une majoration pour la complexité dans le pire cas, une minoration pour le meilleur cas si on vous l'a explicitement demandé).
- Pour les algorithmes sur les arbres, la complexité est le plus souvent :
  - soit proportionnelle au nombre de nœuds, s'il faut faire (un seul) parcours de l'arbre ;
  - soit majorée par la hauteur de l'arbre (et proportionnelle à celle-ci dans le pire des cas) s'il suffit de descendre le long d'une branche.
- Ayez bien en tête que la hauteur d'un arbre binaire est comprise entre environ $\log_2 n$ (s'il est équilibré) et n (s'il est très déséquilibré).