DEVOIR MAISON IV

Longuest Increasing Subsequence

Dans ce DM, on s'intéresse au problème **L**onguest **I**ncreasing **S**ubsequence :

- Entrée : Une suite finie $s_0, ..., s_n$
- Sortie Une plus longue sous-suite croissante (au sens large) de (s_i) .

Dans ce DM :

- En partie 1, on révise comment lire dans un fichier et convertir les données lues.
- En partie 2, on implémente une solution naïve qui trouve la longueur d'une LIS ; en mauvais complexité.
- En partie 3, on optimise la solution précédente en utilisant de la programmation dynamique.

On procède en OCaml [1]. L'usage de l'entierté du module `List`, du module `Array` et du module `Option` sont autorisés [2]. Pour d'autres modules (`Stdlib`, `Scanf`, `Printf`, `Seq`, `String`) : envoyez-moi un mail et je vous dirai probablement "OK".

Consignes de rendu

Le DM est à rendre en solo ou en duo, pour dimanche 6 avril 2025 20h00 (heure de Paris). Le lien de rendu est le suivant :

`https://nuage04.apps.education.fr/index.php/s/JL5qjg6ypEjKRYj`

Si vous rendez en duo, merci de rendre un simple zip (nommé avec vos deux noms de famille).

Fichiers fournis

Un fichier `Makefile` est fourni. Il contient les instructions de compilation (sous une forme quelque peu difficile à lire) afin que vous n'ayez pas à compiler vous-même. Voici les différents modes qu'il propose :

- Dans le terminal, `make` ou `main main` compile et exécute ensemble `load.ml`, `naif.ml`, `dyn.ml` et `main.ml` ensemble (le fichier `main.ml` est le fichier dans lequel vous allez faire vos tests, mais que je ne récupererai pas).
- Dans le terminal, `make test` compile et exécute ensemble `load.ml`, `naif.ml`, `dyn.ml` et mon testeur (vous noterez que `main.ml` n'est pas lu cette fois-ci).
- Dans le terminal, `make clean` supprime l'exécutable ainsi que les fichiers intermédiaires de compilation.
- Dans le terminal, `make doc` lit les interfaces pour générer de la jolie documentation (sous format `html`, dans le dossier `doc/`). Vous pouvez ensuite ouvrir cette documentation avec votre navigateur internet, par exemple : `firefox doc/index.hmtl`
- Dans le terminal, `make zip` vous demande votre nom puis créer le zip à rendre.

Les fichiers que vous pouvez modifier sont `load.ml`, `naif.ml`, `dyn.ml` et `main.ml` . Si vous voulez également modifier les interfaces, envoyez-moi un mail [3].

Enfin, la documentation de toutes les fonctions est disponible dans les interfaces. Vous pouvez soit la lire depuis celmles-ci, soit depuis le format html (cf `make doc`).

I - `load.ml`

1) Fonctions utiles

Voici, en vrac et non-exhaustivement, des fonctions qui peuvent vous servir pour cette partie : `Array.of_list`, `String.split_on_char`, `List.map` et `int_of_string` .

Je vous encourage à aller lire leur documentation et à tester ces fonctions dans `utop`.

1. Version 4.14.1 ou 4.14.2
2. Pour la bonne version d'OCaml ; n'utilisez pas des fonctionnalités de OCaml 5.x ...
3. Là encore, j'accepterai probablement vos modifs ; mais je devrai modifier mon processus de test pour prendre en compte vos nouvelles interfaces.

2) Programmons !

Dans cette section, on cherche à lire des données depuis un fichier. Vous pouvez vous référer à ce sujet au TP 19. Le fichier que l'on va s'efforcer de lire est `sequences.txt` : n'hésitez pas à l'ouvrir pour voir la tête de son contenu.

Toutes les questions ci-dessous concernent `load.ml` . Vous pouvez aussi `main.ml` pour faire vos tests, ou tester dans `utop`. Dans les deux cas : **testez** !! Ne vous contentez pas de mon testeur : le but est que vous compreniez ce que font les fonctions que vous manipulez.

0. Complétez la fonction `get_lines` . Par exemple, si on l'utilise pour lire `sequences.txt`, elle doit renvoyer la liste suivante :

```
["1 2 3 4 5";
 "1 2 3 4 5 4 3 2 1";
 "1 5 2 3 3 4 2 5 1";
 "4 1 5 2 4 1 3 5 3";
 "4 7 3 1 8 2"]
```

1. **a.** Compléter la fonction `intlist_of_line`

b. Déduisez-en `intarray_of_line`

2. En déduire `get_intlist` et `get_intarray`

II - `naif.ml`

1) Fonctions utiles

Voici, en vrac et non-exhaustivement, des fonctions qui peuvent vous servir pour cette partie : `List.map`, `List.filter`, `List.length` (ainsi que nos vieilles amies `::` et `@`).

Je vous encourage à aller lire leur documentation et à tester ces fonctions dans `utop`.

2) Programmons !

Ici, pas de surprise dans l'ordre conseillé pour programmer. Encore une fois, **testez** sur vos propres séquences (et sur celles de `sequences.txt`).

3. Compléter `is_increasing` .

4. Compléter `all_subsequence` .

Indication : calculer toutes les sous-séquences de la queue, puis à chacune d'entre elle ajouter ou non la tête. Par exemple, pour $l = [1; 2; 3]$:

- On calcule les sous-séquences de la queue. Ce sont `[]`, `[2]`, `[3]`, `[2;3]`
- On calcule ce que l'on obtient en ajoutant la tête à ces sous-séquences. Ce sont `[1]`, `[1; 2]`, `[1; 3]` et `[1; 2; 3]`
- Les sous-séquences de la liste sont l'union de ces deux ensembles de sous-séquences.

5. En déduire `all_sub_increasing` .

6. En déduire `lis` .

7. (Bonus) Compléter `sol_lis` .

III - `dyn.ml`

1) Fonctions utiles

Voici, en vrac et non-exhaustivement, des fonctions qui peuvent vous servir pour cette partie : `Option.get`, `List.map`, `Array.make`, `Array.map`, `Array.of_list`, `List.rev`.

Vous ressentirez peut-être le besoin d'avoir des boucles `while` (j'ai tout fait sans, sans me forcer, mais je vois pourquoi on pourrait en avoir envie). Une boucle `while` fonctionne comme une boucle `for` (c'est une expression de type unit) et voici sa syntaxe :

```
(* la boucle est une expression de type unit *)
  while booleen do
    (* ce qu'on fait à chaque iter, comme dans un for
    ...
    ...
    ...
    ...
    *)
  done
```

2) Résolution théorique du problème

On va compléter une mémoire `lg` telle que **`lg.(i)` soit la longueur de la plus longue sous-suite croissante se terminant par l'élément d'indice `i`** .

Exemple.

arr :	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15
lg :	1	2	2	3	2	3	3	4	2	4	3	5	3	5	4	6

Dans l'exemple ci-dessous :

- `lg.(0) = 1` car la plus longue sous-suite croissante se terminant à l'indice `0` de `arr` est « 0 ».
- `lg.(4) = 2` car la plus longue sous-suite croissante qui se termine à l'indice 4 de `arr` est « 0, 2 ».
- `lg.(7) = 4` car les plus longues sous-suite croissantes se terminant à l'indice 7 de `arr` sont par exemple « 0, 8, 12, 14 », « 0, 8, 10, 14 », « 0, 4, 12, 14 », « 0, 4, 10, 14 » (il y en a d'autres, toutes de longueur 4).

8. Vérifiez que vous êtes d'accord avec chacune des valeurs de `lg` dans l'exemple ci-dessus, c'est à dire que vous arrivez bien à trouver une sous-suite croissante de cetet longueur se terminant à l'indice indiqué, et que vous n'en trouvez pas une plus longue[4].

On cherche donc à calculer toutes les `lg.(i)` . Ils vérifient une formule de récurrence : en effet, une plus longue sous-suite croissante se terminant à l'indice `i` de `arr` peut se décomposer comme :

- Une sous suite croissante se terminant avant l'indice `i`. Notons `k` son indice de fin.
- À quoi suit `arr.(i)` : en particulier, pour que le tout soit croissant, il est nécessaire et suffisant que `arr.(k) <= arr.(i)`

Pour un indice `i` de `arr`, notons $prec(\texttt{arr}, \texttt{i}) = \{0 \leq j < i \; tq \; \texttt{arr.(j) <= arr.(i)}\}$.
Pour que la sous-suite croissante se terminant à l'indice `i` soit la plus longue possible, on en déduit la formule de récurrence possible :

$$\texttt{lg.(i)} = \begin{cases} 1 + \max\{\texttt{lg.(k)} \text{ tels que } 0 \leq k < i \text{ et } \texttt{arr.(k)} \leq \texttt{arr.(i)}\} & \\ 1 & \text{si l'ensemble précédent est vide} \end{cases}$$

3) Programmons !

Je conseille fortement de procéder par programmation dynamique de bas en haut : on va donc créer un tableau `lg` et le remplir dans le bon ordre, sans utiliser de récursivité.

9. Proposer un ordre de calcul des valeurs de `lg` (indication : c'est vraiment pas compliqué).

10. Complétez la fonction `search` qui sert à calculer le max de la formule de récurrence. On supposera que toutes les valeurs de `lg` appelées sont correctement remplies.

11. Tant que vous avez la tête dedans, complétez la fonction `search_indice` : elle fait comme `search`, sauf qu'au lieu de renvoyer le `lg.(k)` optimal elle renvoie le `k` associé. Plus précisément, elle renvoie `Some k` si un tel `k` existe et `None` sinon (ce qui correspond au second cas de la formule de récurrence).

12. À l'aide de `search`, codez `lis` . *Indication : calculez tous les* `lg.(i)` *par programmation dynamique ascendante, puis parcourez le tableau pour trouver le maximum.*
Par exemple, sur l'exemple ci-dessus, `lis` doit renvoyer 6. N'hésitez pas à également utiliser les séquences de `sequences.txt` pour vous tester, ainsi que vos propres exemples.

4. Si vous en exhibez une plus longue, envoyez moi un mail et je corrigerai la faute.

Reste le gros morceau :

13. À l'aide de `search_indice`, complétez `sol_lis`. On demande de renvoyer une sous-séquence optimale comme un tableau. Je vous conseille de procéder comme suit :
    - D'abord, construisez la solution comme la liste des indices associés à une sous-séquence optimale.
    - Puis transformez cette liste en le tableau des valeurs de `arr` associées, et renvoyez-le.

Remarque. Dans mon testeur, les tests finaux pour `lis` et `sol_lis` sont de *grandes* entrées : il est normal que votre code prenne un peu de temps. En guise de référence, mon code (pas très optimisé, mais qui tourne sur un bon processeur) compile et passe tous les tests en 6,5s au total.