TP n° 2 – Algorithmes gloutons et dynamiques

esimon@ens-cachan.fr
thomas.h@ens.fr

1 Oups !

Question 1. Yorel Reivax implémente le code suivant, qui crée une matrice 4×2 remplie de 0 et initialise le premier élément à 1.

```
let v = make_vect 4 (make_vect 2 0);;
v.(0).(0) <- 1;;
```

Recopier son code, et afficher `v`. Que se passe-t-il ? Comment y remédier [1] ?

2 Bin packing

Un algorithme est dit **glouton** s'il fait à chaque étape un choix localement optimal (la solution qui lui semble la meilleure sans chercher trop loin ni revenir sur ses choix) dans l'espoir d'obtenir une solution globalement optimale (la meilleure possible).

L'objectif de cet exercice est de trouver une bonne approximation au problème de *bin packing* (« remplissage de boîte ») [2]. Vous avez des boîtes de taille V et une liste $1, 2, \ldots, n$ d'objets i de taille a_i. Vous voulez ranger tous les objets dans les boîtes en utilisant le moins de boîte possible. Si vous mettez un ensemble d'objet K dans une même boîte, il faut que l'inégalité $\sum_{k \in K} a_k \leq V$ soit vérifiée.

Question 2. Écrire une fonction `frac_pack` qui étant donné une liste d'objet et une taille de boîte retourne le nombre de boîtes utilisées en supposant qu'il est possible de découper les objets. Vous pourrez par exemple, découper un objet de taille 5 en deux objets de taille 2 et un objet de taille 1.

Dans la suite de l'exercice, on s'intéressera à des objets insécables. Vous pourrez représenter un remplissage par une liste de tuples `(int * int list)` (taille occupée, liste des tailles des objets dans la boîte), par exemple `(10, [5; 2; 3])`.

Question 3. Écrire une fonction `pack` qui étant donné un remplissage, la taille des boîtes et un élément à insérer, retourne un nouveau remplissage où l'élément a été ajouté à la première boîte capable de l'accueillir.

Question 4. Écrire une fonction `bin_pack1` qui étant donné une taille de boîte et une liste d'objet retourne un remplissage en utilisant la fonction `pack` de la question précédente.

Question 5. Votre fonction `bin_pack1` retourne toujours un remplissage dont la valeur est au plus deux fois celle du remplissage optimal, voyez vous pourquoi ? Trouvez un exemple où le rapport entre la solution trouvée et la solution optimale s'approche le plus possible de 2.

Question 6. Écrire une fonction `tri_selection` qui trie une liste de nombres dans l'ordre décroissant, en sélectionnant à chaque étape le plus grand élément de la liste pour le placer en tête.

1. Dans la suite du TP, vous pourrez utiliser la fonction `make_matrix` de la bibliothèque standard.
2. Ce problème étant NP-difficile, on ne s'attendra pas à une solution exacte.

Question 7. Définir la fonction `bin_pack2` qui applique la fonction de tri à la liste d'objet avant de la passer à `bin_pack1`. Quelle est la complexité de cet algorithme ? Trouvez un exemple sur lequel cette fonction ne retourne pas la solution optimale.

3 Sous-chaines et sous-séquences

Un algorithme est dit de **programmation dynamique** s'il ramène un problème à plusieurs instances plus petites du même problème qu'il résout récursivement, en mémorisant les résultats pour ne pas avoir à calculer plusieurs fois la même chose.

Soit une liste $l_1, l_2, \ldots, l_n$. Une sous-chaine de l est un suite de nombres consécutifs dans l, pouvant être définie par un intervalle de valeur dans $[\![1, n]\!]$. Une sous-séquence de l est une sous-suite quelconque de l, pouvant être définie par un sous-ensemble de valeur de $[\![1, n]\!]$. Par exemple [42 ; 8 ; 5 ; 12 ; 7] a [8 ; 5 ; 12] comme sous-chaine et [42 ; 5 ; 12] comme sous-séquence.

Question 8. Écrire une fonction qui étant donné deux listes, trouve leur sous-chaine commune la plus longue.

Question 9. Modifier légèrement la fonction précédente pour trouver leur sous-séquence commune la plus longue.

Question 10. Quelle est la complexité mémoire de votre fonction ? Si elle n'est pas linéaire en mémoire, réécrivez là.

Question 11. Une sous-séquence croissante est une sous-suite d'éléments croissants. Écrire une fonction qui étant donné une liste, retourne sa plus longue sous-séquence croissante.

4 Levenshtein est paresseux

Question 12. Écrire une fonction `levenshtein` qui calcule la distance de Levenshtein entre deux mots. La distance de Levenshtein donne une mesure de la similarité entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre. Votre fonction devra avoir une complexité linéaire en mémoire.

Question 13. Écrire une fonction `bounded_levenshtein` qui étant donné une distance de Levenshtein maximale d, trouve la distance de Levenshtein entre deux mots en $O(d)$ en mémoire et $O(nd)$ en temps. Commencez par trouver une borne inférieure sur les valeurs aux coins de la matrice que vous calculez, puis déduisez en un ensemble de valeur qu'il est inutile de calculer étant donné d.

Question 14. Caml utilise une stratégie d'évaluation stricte, c'est-à-dire qu'une expression est évaluée dès qu'elle peut être liée à une variable. L'évaluation paresseuse évalue une expression seulement lorsqu'on en fait explicitement la demande. Écrire une fonction `lazy_levenshtein` qui calcule la distance de Levenshtein de manière paresseuse, cela vous permettra d'atteindre une complexité de $O(nl)$ en temps avec l la distance de Levenshtein entre les deux chaines (ici, on ne vous donne pas de borne, la complexité dépend de la solution).

Pour représenter un résultat non calculé vous pourrez utiliser la valeur `None` du type `int option`, une valeur calculé v sera alors représenté par `Some v`. Remarquez que vous devez propager la propriété de paresse à toute les fonctions, par exemple vous devez écrire une fonction `lazy_min3` qui étant donné une coordonné dans la matrice, n'évalue que les cases parentes nécessaires.

Si vous voulez avoir un algorithme optimal en mémoire, il sera judicieux de considérer la matrice diagonale par diagonale.