Travaux pratiques 1

Sommes maximales

Option informatique MPSI1 & MPSI2

Dans ce T.P. nous allons étudier diverses méthodes de résolution d'un problème simple : la recherche d'une somme de k termes d'une liste en imposant ou non une contrainte.
Dans tout le sujet on ne considérera que des tableaux dont les éléments sont des entiers **positifs**.

I Premier problème

On cherche, dans un tableau d'entiers positifs k éléments dont la somme est maximale.
Bien entendu la taille n du tableau doit vérifier ;$n \geqslant k$.
Un algorithme immédiat consiste à répéter k fois les deux instructions

- déterminer le maximum du tableau,
- le retirer du tableau.

La première étape est classique.

Question 1 - Indice du maximum
Écrire une fonction `indice_maxi : int array -> int` telle que `indice_maxi t` renvoie un indice k tel que `t.(k)` soit le maximum des `t.(i)` pour $0 \leqslant i < n$ où n est la longueur du tableau.
On pourra supposer, sans avoir besoin de le vérifier, que le tableau n'est pas vide.

Pour éliminer le maximum, on peut recopier les éléments restants dans un nouveau tableau :

```
let eliminer_naif t k =
   let n = Array.length t in
   let t1 = Array.make (n-1) 0 in
   for i = 0 to (n-2) do
      if i < k
      then t1.(i) <- t.(i)
      else t1.(i) <- t.(i+1) done;
   t1;;
```

Les problèmes que crée cette méthode sont

- le nombre d'opérations, $n-1$, effectuées
- la nécessité de définir un nouveau tableau à chaque étape.

Nous allons privilégier une solution qui se fait en place. Comme les éléments sont positifs, on peut éliminer un éléments en le remplaçant par 0 : `t.(i) <- 0`.
L'inconvénient est que cela modifie le tableau ; on travaillera sur une copie du tableau passé en paramètre. On peut obtenir cette copie avec `Array.copy t`.

Question 2 - Somme maximale de k termes
Écrire une fonction `somme_max : int array -> int -> int` telle que `somme_max t k` renvoie la somme des k plus grands termes de `t`.

Question 3 - Cas limites
Que se passe-t-il si `t` a moins de k éléments ?

Question 4 - Enrichissement du résultat
Modifier la fonction `somme_max : int array -> int -> int * int array` pour qu'elle renvoie, en plus de la somme maximale, un tableau de taille k contenant les indices correspondants aux termes de cette somme maximale.

II Second problème, algorithme glouton

On impose maintenant que les termes qui forment la somme maximale ne soient pas consécutifs. Par exemple pour `t = [|11; 5; 23; 22; 8; 18; 19; 7; 16; 12|]` et $k = 3$ la somme maximale que l'on peut obtenir est $23 + 19 + 16$ car, après avoir choisi 23, on ne peut plus choisir 22 et, de même, 18 n'est pas sélectionnable après avoir choisi 18.
On voit donc l'algorithme suivant : répéter k fois les deux instructions

- déterminer le maximum du tableau,
- le retirer du tableau ainsi que ses voisins de droite et de gauche.

Question 5 - Somme maximale de k termes avec contrainte
Écrire une fonction `somme_max2 : int array -> int -> int * int array` telle que `somme_max_ t k` renvoie la somme de k termes de `t` non contigus et un tableau de taille k contenant les indices correspondants aux termes de cette somme en appliquant l'algorithme.

Question 6 - Ça ne marche pas
Appliquer la fonction `somme_max_2` à `t0`.

```
let t0 = [|15; 4; 20; 17; 11; 8; 11; 16; 7; 14;
           2; 7; 5; 17; 19; 18; 4; 5; 13; 8|];;
```

Montrer que le résultat n'est pas maximal.

Que s'est-il passé ?
L'algorithme présenté est un algorithme **glouton** : il essaie de déterminer une solution optimale en effectuant des choix locaux, jamais remis en cause. Au cours de la construction de la solution, l'algorithme résout une partie du problème puis se focalise ensuite sur le sous-problème restant à résoudre et ainsi de suite. Cette stratégie à courte vue n'est que rarement efficace, elle l'a été dans la partie précédente mais ne reste plus ici.

III Second problème, force brute

En l'absence d'une solution subtile, on peut essayer la grosse artillerie : énumérer toutes les partie à k éléments parmi $\{0, 1, 2, \ldots, n-1\}$ et pour chacune, notée A, considérer la somme des `t.(i)` pour $i \in A$ en conservant la somme maximale.
Deux types de stratégies sont possibles pour obtenir tous les ensembles possibles :

- les fabriquer à l'avance, faire un ensemble d'ensemble d'ensembles,
- donner un moyen de passer d'un ensemble à un autre en garantissant de les fabriquer tous.

La seconde méthode est préférable car elle ne grée pas un ensemble trop gros ; par exemple il y a 75 287 520 parties à 5 éléments dans un ensemble à 100 éléments.

Il existe plusieurs algorithmes qui permettent de construire les k-combinaisons (sous ensembles à k éléments) d'un ensemble donné.
Voici un algorithme qui crée les k-combinaisons de $\{0, 1, 2, \ldots, n-1\}$ (cela fonctionne dans tout ensemble ordonné) dans l'ordre croissant si on les écrit les éléments du plus petit au plus grand dans les sous ensembles. On admettra qu'il permet de construire toutes les combinaisons.

Initialisation On part de $(0, 1, \ldots, k-1)$.

Étape suivante Pour construire de successeur de $(c_0, c_2, \ldots, c_{k-1})$ avec $0 \leqslant c_0 < c_1 < \cdots < c_{k-1} < n$:

Point de changement on détermine le dernier p tel que c_p peut encore augmenter : $c_p < n - k + p$ et $c_{p+1} = n - k + (p+1)$, …, $c_i = n - k + i$, …, $c_{k-1} = n - 1$ ou $p = -1$

Fin si $p = -1$ alors on est parvenu à $(n-k, n-k+1, \cdots, n-1)$,
c'est la plus grande combinaison, on a fini,

Calcul du terme suivant sinon on augment c_p de 1
puis on pose $c_{p+1} = c_p + 1$, $c_{p+2} = c_{p+1} + 1$ jusqu'à $c_{k-2} = c_{k-1} + 1$.

Par exemple, pour $n = 20$ et $k = 5$,

- si on part de $(9, 10, 13, 14, 16)$, on a $p = 4$ et le terme suivant sera $((9, 10, 13, 14, 17))$,
- si on part de $(2, 7, 11, 18, 19)$, on a $p = 3$ et le terme suivant sera $(2, 7, 12, 13, 14)$,
- si on part de $(9, 16, 17, 18, 19)$, on a $p = 0$ et le terme suivant sera $(10, 11, 12, 13, 14)$.

Pour $n = 5$ et $k = 3$, on obtient successivement, en partant de $(0, 1, 2)$,
$(0, 1, 3)$, $(0, 1, 4)$, $(0, 2, 3)$, $(0, 2, 4)$, $(0, 3, 4)$, $(1, 2, 3)$, $(1, 2, 4)$, $(1, 3, 4)$ puis enfin $(2, 3, 4)$.
On trouve bien les $\binom{5}{3} = 10$ combinaisons.

Question 7 - Calcul du successeur
Écrire une fonction `suivant : int -> int array -> bool` telle que `suivant n c` renvoie `true` si la combinaison `c` admet un successeur , `c` sera alors transformé en ce successeur.
Par contre si `c` est la dernière combinaison la fonction renvoie `false` et `c` est quelconque.
n est le nombre d'éléments avec lesquels on travaille.

Question 8 - Tester les combinaisons
Écrire une fonction `test : int array -> bool` qui teste si une combinaison d'indice a des sauts d'au moins 2 entre deux termes consécutifs.

Question 9 - Somme maximale de k termes avec contrainte
Écrire une fonction `somme_max2_brute : int array -> int -> int * int array` telle que `somme_max2_brute t k` renvoie la somme maximale de k termes de `t` non contigus et un tableau de taille k contenant les indices correspondants aux termes de cette somme maximale.

IV Second problème, récursivité

Pour résoudre le second problème de manière récursive, on le généralise.
On note $g(t, p, k)$ la somme maximale que l'on peut obtenir en choisissant k termes du tableau `t` parmi les p premiers termes de ce tableau, et en respectant, bien sur, la contrainte d'indices séparés de 2 au moins.

Question 10 - Calculs
Prouver les résultats suivants :

- $g(t, p, k)$ n'a de sens que pour $2k - 1 \leqslant p < n$, n étant la longueur de t,
- $g(t, p, 0) = 0$,
- $g(t, 2k - 1, k) = g(t, 2k - 3, k - 1) +$ `t.(2*k-1)`,
- $g(t, p, k) = \max\big(g(t, p - 1, k), g(t, p - 2, k - 1) +$ `t.(p-1)` pour $p > 2k - 1$.

Question 11 - Somme maximale de k termes avec contrainte
En déduire une fonction `somme_max2_rec : int array -> int -> int * int array` telle que `somme_max2_rec t k` renvoie la somme maximale de k termes de `t` non contigus et un tableau de taille k contenant les indices correspondants aux termes de cette somme maximale.

V Comparaisons temporelles

Question 12 - Calcul du temps
En utilisant la fonction `time : init -> float` du module `Sys` telle que `Sys.time ()` renvoie le temps depuis l'ouverture de la console OCaml, comparer les temps de calculs.

Le désavantage de la méthode de force brute provient du fait que l'on fait trop de comparaisons.

Question 13 - Création des combinaisons adaptées
Modifier la fonction `suivant` pour que `suivant n c` modifie une combinaison à écarts de 2 au moins (adaptée au problème 2) en la combinaison suivante adaptée.
Modifier alors `somme_max2_brute` (on commence par $(0, 2, 4, \ldots, 2k - 2)$.
Comparer les temps.

La méthode récursive peut être beaucoup améliorée en ne calculant plus les termes récursivement mais en calculant systématiquement toutes les valeurs possibles dans un tableau.

Question 14 - Programmation dynamique
Modifier la fonction ainsi `somme_max2_rec` (elle n'est plus récursive).
Le temps de calcul devient comparable à celui de la méthode gloutonne !