Exercices de programmation pour les vacances mp2i → mpi

04 septembre

1 Exercice 1 : autour de la recherche par dichotomie

Dans tout l'exercice, on ne considère que des tableaux d'entiers de longueur $n \geq 0$.

Un squelette de programme C vous est donné, avec un jeu de tests qu'il *ne faut pas modifier*. Vous pouvez bien sûr ajouter vos propres tests à part.

1. Écrire une fonction de prototype `bool nb_occurrences(int n, int* tab, int x)` qui renvoie le nombre d'occurrences de l'élément `x` dans le tableau `tab` de longueur `n`. Quelle est la complexité de cette fonction ?

Dans toute la suite, on suppose que les tableaux sont *triés* dans l'ordre croissant. On va chercher à écrire une version plus efficace de la fonction ci-dessus qui exploite cette propriété. On cherche tout d'abord à écrire une fonction `int une_occurrence(int n, int* tab, int x)` qui permet de renvoyer un indice d'une occurrence quelconque de l'élément `x` s'il est présent dans le tableau et `-1` sinon. On procède par dichotomie.

2. Compléter le code de la fonction `int une_occurrence(int n, int* tab, int x)` qui vous est donnée dans le squelette. Cette fonction doit avoir une complexité en $O(\log n)$.
3. Écrire une fonction `int premiere_occurrence(int n, int* tab, int x)` qui renvoie l'indice de la première occurrence d'un élément `x` dans un tableau `tab` de longueur `n` si cet élément est présent et -1 sinon. Cette fonction doit avoir une complexité en $O(\log n)$.
4. Écrire une fonction `int nombre_occurrences(int n, int* tab, int x)` qui renvoie le nombre d'occurrences de l'élément `x` dans le tableau `tab` de longueur `n`. Cette fonction devra avoir une complexité en $O(\log n)$.
5. Justifier que la fonction `une_occurrence` termine et est correcte. On donnera un variant et un invariant de boucle que l'on justifiera.
6. Montrer que la complexité de la fonction `une_occurrence` est bien en $O(\log n)$.

2 Exercice 2 : plus court chemin dans un DAG

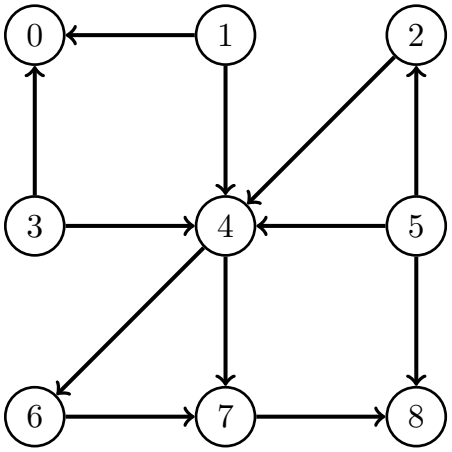
Nous allons ici écrire un programme qui permet de calculer les plus courts chemins dans un graphe orienté acyclique pondéré. Plus précisément, étant donné un graphe orienté acyclique pondéré $G = (S, A, p)$ (on considère les sommets indexés de 0 à $|S| - 1$) et un sommet $s \in S$, on veut renvoyer un tableau `d` de taille $|S|$ tel que pour tout $i \in S$, `d[i]` contienne le poids minimal d'un chemin de s à i.

On utilisera la structure suivante pour représenter les graphes pondérés :

```
struct graph_s {
int n;
int degre [100];
int voisins [100] [10];
int poids [100][10];
};
```

L'entier n correspond au nombre de sommets $|S|$ du graphe. On suppose que $n \leqslant 100$. Pour $0 \leqslant s < n$, la case $degre[s]$ contient le degré sortant $d^+(s)$, c'est-à-dire le nombre de successeurs, appelés ici voisins, de s. On suppose que ce degré est toujours inférieur à 10. Pour $0 \leqslant s < n$, la case `voisins[s]` est un tableau contenant, aux indices $0 \leqslant i < d^+(s)$, les voisins du sommet s et pour $0 \leqslant s < n$, la case `poids[s]` est un tableau contenant, aux indices $0 \leqslant i < d^+(s)$, les poids des arcs `(s,voisins[s][i])`. Il s'agit donc d'une représentation par listes d'adjacences où les listes sont représentées par des tableaux en C.

Un programme en C vous est fourni dans lequel le graphe suivant est représenté par la variable `g_exemple`.



1. Avec cette représentation, quel est le nombre maximum de sommets que le graphe peut avoir ? D'arcs ?
2. Ecrire une fonction `void mise_a_jour(struct graph_s g, int u, int v, int* d)` telle que l'appel à `mise_a_jour(g,u,v,d)` met à jour la case `d[g.voisins[u][v]]` avec le minimum de sa valeur actuelle et de la valeur `d[u]+p(u,g.voisins[u][v])`.
3. Ecrire une fonction `int* creer_tableau_distances(struct_graph_s g, int s)` qui alloue et initialise un tableau des distances `d` de bonne taille tel que `d[s]=0` et pour tous les autres sommets, `x`, `d[x]=10000`.

   Le principe de l'algorithme de recherche de plus court chemin est le suivant : on initialise un tableau `d` avec la fonction précédente et on trie par ordre topologique les sommets du graphe puis on parcourt ces derniers dans cet ordre et pour chaque sommet u, on appelle la fonction `mise_a_jour(g,u,v,d)` pour chaque arc d'origine `u`.

4. Compléter la fonction `tri_topologique` fournie qui permet d'obtenir un ordre topologique.
5. Ecrire une fonction `int* plus_court_chemin(struct graph_s g, int s)` qui renvoie le tableau des poids minimaux des chemins entre s et chacun des sommets du graphe à l'aide de l'algorithme proposé.
6. Prouver la correction de l'algorithme proposé calculant les plus courts chemins dans un graphe orienté pondéré acyclique.
7. Ecrire une fonction `bool detect_cycle(struct graph_s g)` qui vérifie si un graphe orienté pondéré est bien acyclique.

3 Exercice 3 : Timsort

On remarque qu'en pratique les tris par comparaison ont rarement des entrées trop désordonnées. On sait par ailleurs que le tri rapide a de bonnes performances sur des tableaux bien désordonnés ce qui

ne correspond donc pas forcément à la réalité. C'est pourquoi, des modifications récentes des librairies utilisent des algorithmes de tri qui sont performants sur des entrées "pseudo-triées".

Toute liste ℓ non vide peut être décomposée en $\rho \geqslant 1$ séquences croissantes maximales, qui sont des sous listes croissantes $\ell_1, \ldots, \ell_\rho$ dont la concaténation fait ℓ et telles que pour tout $i \in \{1, \ldots, \rho - 1\}$ le dernier élément de ℓ_i est strictement plus grand que le premier élément de ℓ_{i+1}.

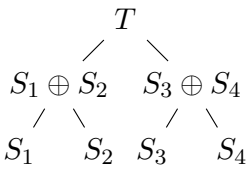
On définit la SCM d'une liste ℓ comme la liste de listes de taille ρ qui contient, dans l'ordre, les séquences définies précédemment.

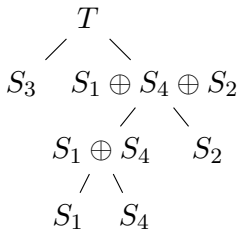
Exemple : Si on considère la liste : `[2;4;20;29;3;3;4;1;1;4;32;8;11]` alors sa SCM est représentée par `[[2;4;20;29];[3;3;4];[1;1;4;32];[8;11]]`.

1. Ecrire une fonction `scm : int list->int list list` telle que `scm l` renvoie la SCM de l'entrée `l`. On attend une complexité linéaire en la taille de la liste passée en entrée.
2. Ecrire une fonction `fusion : int list->int list->int list` telle que si on considère deux listes triées `l1,l2`, `fusion l1 l2` renvoie une liste triée qui contient les éléments de `l1` et ceux de `l2`. Quelle est la complexité (nombre de comparaisons) de votre fonction dans le pire cas ?

   L'objectif ici est, à partir de la SCM d'une liste `l`, de trouver une succession de fusions à effectuer sur les séquences la composant pour obtenir une liste triée composée des éléments de `l`. Une représentation d'une succession de fusions peut être faite par un arbre binaire strict dont les feuilles sont les listes et un noeud interne correspond à la fusion des deux listes triées obtenues par la représentation de chacun de ses deux fils. L'arbre choisi aura une influence sur la complexité finale obtenue.

   Exemple : Considérons de nouveau la liste : `[2;4;20;29;3;3;4;1;1;4;32;8;11]` avec sa SCM `[[2;4;20;29];[3;3;4];[1;1;4;32];[8;11]]`. Si on note $S_1 = [2; 4; 20; 29], S2 = [3; 3; 4], S_3 = [1; 1; 4; 32]$ et $S_4 = [8; 11]$ alors on peut obtenir l'un des deux arbres suivants pour représenter la succession de fusions :





3. Dans chacun des deux cas, donner le nombre de comparaisons total de l'ensemble des fusions effectuées sur l'exemple.

   On utilisera le type suivant pour décrire un arbre binaire strict :

```
type arbre = Feuille of list | Noeud of (arbre*arbre)
```

   Les arbres de la figure ci-dessous correspondent donc aux arbres suivants :

```
let s1 = [2;4;20;29];;
let s2 =[3;3;4];;
let s3 =[1;1;4;32];;
let s4 = [8;11];;

let a1 = Noeud(Noeud(Feuille(s1),Feuille(s2)),Noeud(Feuille(s3),Feuille(s4)));;

let a2 = Noeud(Feuille(s3),Noeud(Noeud(Feuille(s1),Feuille(s4)),Feuille(s2)));;
```

4. Ecrire une fonction `sort_tree : arbre->list` qui à partir de l'arbre des fusions renvoie la liste triée.
5. Une solution est de construire un arbre le plus équilibré possible (arbre binaire strict complet, c'est-à-dire tel que toutes les feuilles sont à même hauteur ou ont une différence de hauteur d'au plus un).

   Ecrire une fonction `constr_arbre : int list list-> arbre` qui construit un tel arbre des fusions. Quelle est la complexité de votre construction ?
6. Ecrire une fonction `tri : int list->int list` qui trie une liste passée en entrée suivant l'algorithme proposé. En déduire une fonction de tri.
7. Déterminer le coût de votre algorithme en fonction de la taille `n` de la liste et de ρ, le nombre de SCM composant la liste.
8. Revenons au cas général où l'arbre des fusions peut-être quelconque : donner une formule qui décrit la complexité de l'algorithme de tri en fonction des profondeurs des différents noeuds de l'arbre et de la taille des séquences de la SCM. A l'aide d'un algorithme au programme, commenter la complexité d'un algorithme glouton qui permettrait de construire un arbre des fusions optimal. Quelle est la complexité de sa construction ?

4 Exercice 4 : Algorithme de Wigderson

Un graphe $G = (S, A)$ admet une coloration $c : S \to \{0, \ldots, k-1\}$ avec k couleurs ssi $\forall (x, y) \in A, c(x) \neq c(y)$.

Pour trouver une coloration d'un graphe on peut utiliser l'algorithme glouton suivant :

Pour chaque sommet $s \in S$, on lui attribut la plus petite couleur non utilisée par un de ses voisins déjà colorié.

On va commencer par coder cet algorithme :

Un graphe sera représenté par sa matrice d'adjacence (`bool array array`), ainsi un graphe à n sommets sera représenté par un tableau de dimension 2 de taille $n \times n$. Une coloration sera représentée par un tableau d'entiers (`int array`) de taille n noté `c` tel que la case `c.(i)` contient la couleur du sommet `i`. On attribuera la value -1 à un sommet qui n'est pas encore colorié.

1. Écrire une fonction
   `couleur_a_utiliser : bool array array -> int array -> int -> int -> int` telle que `couleur_a_utiliser g coloration s c` renvoie la plus petite couleur supérieure ou égale à `c` non déjà utilisée par un voisin de `s` dans `g` partiellement colorié avec `coloration`.
2. En déduire une fonction `coloriage : bool array array -> int array` qui renvoie une coloration du graphe passé en entrée en utilisant l'algo glouton.
3. Donner un majorant du nombre de couleurs utilisées par cet algorithme.

L'algorithme de Wigderson permet de trouver un coloriage utilisant $O(\sqrt{n})$ couleurs pour un graphe dont on sait qu'il est 3-coloriable.

L'algorithme de Wigderson fonctionne selon le schéma suivant :

* On fixe initialement c à 0.
* Pour chaque sommet $s \in G$ ayant au moins $\sqrt{n}$ voisins non encore coloriés :
  1. On 2 colorie avec les couleurs c et $c+1$ le sous graphe induit par les voisins de s non encore coloriés.
  2. On incrémente c du nombre de couleurs utilisées lors de cette étape.

* On utilise l'algorithme glouton pour colorier avec des couleurs supérieures ou égales à c les sommets non coloriés.

Un graphe induit $G' = (S', A')$ du graphe G sera représenté par une matrice d'adjacence `g'` de même taille que celle de G où la case `g'.(i).(j)` contient 1 ssi $i \in S', j \in S'$ et $(i, j) \in A$. La connaissance de cette matrice ne suffisant pas à connaitre le graphe induit, on utilisera (si besoin) en plus un tableau de booléens de taille $|S|$ dont la case d'indice `i` contiendra `true` ssi $i \in S'$.

1. Écrire une fonction `deux_color : int array array->int->int array->bool array->unit` qui prend en entrée la matrice d'adjacence d'un sous graphe induit de `g`, une couleur `c`, un coloriage partiel de `g` et un tableau de booléens qui correspond à l'ensemble des sommets effectivement dans le graphe induit et qui met à jour le coloriage en 2 coloriant le graphe induit avec les couleurs c et $c + 1$.
2. Justifier que si le graphe est 3 coloriable, alors l'étape 1 est toujours possible, c'est-à-dire que les sous graphes considérés sont 2 coloriables.
3. Écrire une fonction `graphe_restant : bool array array-> int array->int->bool array array` telle que `graphe_restant g coloration s` renvoie la matrice d'adjacence du graphe induit par l'ensemble des voisins non coloriés de `s` dans `g` partiellement colorié par `coloration`.
4. Écrire une fonction `graphe_glouton : bool array array->int array->bool array array` qui prend en entrée un graphe et une coloration partielle de celui-ci et qui renvoie la matrice d'adjacence du graphe induit par l'ensemble des sommets non coloriés.
5. Écrire une fonction `continuer : bool array array->int array->int option` telle que `continuer g coloriage` calcule le nombre maximal de voisins non déjà coloriés d'un sommet. On note ce degré maximal `d` et la fonction renvoie `Some s` si $d \geqslant \sqrt{n}$ et que le nombre de voisins non coloriés de `s` est `d` et `None` sinon.
6. Écrire une fonction `wigderson : bool array array-> int array` qui renvoie un coloriage du graphe passé en entrée obtenu à l'aide de l'algorithme de Wigderson.
7. Montrer que cet algorithme va utiliser $O(\sqrt{n})$ couleurs.