Exercices : solutions de la série #5

Solution à l'exercice 1.

a. La solution vorace consiste simplement à céduler les tâches en ordre de durée, de la plus courte à la plus longue. Notons par `t[i]` les durées des diverses tâches dans le pseudocode. Un algorithme vorace possible qui retourne une séquence des tâches dans l'ordre approprié serait alors le suivant :

```
PROCEDURE cedulerTaches( t: sequence{Nat} ): sequence{Nat}
PRECONDITION
  length(t) = n
DEBUT
  fp <- new FilePriorite()
  POUR k <- 1 A length(t) FAIRE
    fp.inserer( t[i], i )
  FIN
  resultat <- []
  TANTQUE NON fp.estVide() FAIRE
    (t, i) <- fp.retirerMaxElement()
    resultat <- [i] ++ resultat
  FIN
  RETOURNER resultat
FIN
```

On remarque qu'on obtient le bon ordre en retirant l'élément maximum et en ajoutant cet élément en début de file. Le plus grand sera donc à la fin de la séquence, alors que le plus petit sera au début.

b. La complexité asymptotique de l'algorithme est $\Theta(n \lg n)$ (tant à cause de la création de la file (n appels de coût $O(\lg n)$ qu'à cause du fait qu'on examine tout son contenu, donc n appels eux aussi $O(\lg n)$). Dans le cas présent, un tri des éléments aurait aussi pu être effectué pour obtenir un algorithme de complexité équivalente ($O(n \lg n)$).

c. Le fait que le résultat vorace est optimal s'explique comme suit. Supposons qu'on ait une cédule qui ordonne les tâches comme suit : $j_{i_1}, \ldots, j_{i_n}$. Notons tout d'abord que la tâche j_{i_1} se termine alors au temps t_{i_1}, la tâche j_{i_2} au temps $t_{i_1} + t_{i_2}$, la tâche j_{i_3} au temps $t_{i_1} + t_{i_2} + t_{i_3}$, etc.

Le temps total d'exécution d'une cédule est donc le suivant :

$$\begin{aligned} T &= \sum_{k=1}^{n} (n-k+1) t_{i_k} \\ &= (n+1) \sum_{k=1}^{n} t_{i_k} - \sum_{k=1}^{n} k t_{i_k} \end{aligned}$$

Or, la somme de gauche ne dépend pas de l'ordre des tâches. Par contre, celle de droite dépend clairement de l'ordre des tâches : si une tâche est cédulée plus tard, donc apparaît tard dans l'ordre, le k associé sera plus grand, donc sa contribution sera plus grande à la somme de droite, donc réduira d'autant le temps total d'exécution. Le temps moyen d'exécution étant simplement le temps total divisé par n, on a donc le résultat désiré.

Solution à l'exercice 2.

Supposons qu'il y ait $n+1$ points d'eau le long de la route traversant le désert, nommés $p_0, p_1, \ldots, p_n$. Notons par $d(p_i)$ la distance entre le site de départ et le point d'eau p_i. On suppose que $d(p_0)$. Supposons aussi, pour simplifier, que $i < j \Rightarrow d(p_i) < d(p_j)$. Définissons alors $dist(p_i, p_j) = d(p_j) - d(p_i)$.

L'algorithme serait alors le suivant :

```
DEBUT
   pointsRemplissage <- [p_0]
   distRestante <- K
   POUR i <- 1 A n - 1 FAIRE
     distRestante <- distRestante - dist(p_{i-1}, p_i)
     SI dist(p_i, p_{i+1}) >  distRestante ALORS
       pointsRemplissage <- pointsRemplissage ++ [p_i]
       distRestante <- K
     FIN
   FIN
FIN
```

La complexité asymptotique de l'algorithme est simplement $\Theta(n)$.

Preuve d'optimalité (proposée par Éric Terroux, moniteur pour le cours INF7440 à l'automne 2002)

La faAçon de dAémontrer que l'approche vorace est optimale pour ce problème consiste Aà effectuer une preuve par contradiction. Nous allons supposer qu'il existe une solution optimale O diffAérente de la solution vorace V. De plus, on appellera *puits de remplissage* un puits sur lequel on s'arrAête pour remplir la gourde. Soit :

$V = \{v_1, \ldots, v_i, \ldots, v_n\}$	Puits de remplissage pour la solution vorace.
$O = \{o_1, \ldots, o_i, \ldots, o_n\}$	Puits de remplissage pour la solution optimale.

Ici, on suppose que si le i-iAème puits de remplissage est avant le j-iAème puits de remplissage (c'est-à-dire $i < j$), alors $v_i < v_j$ ou $o_i < o_j$, tout dAépendant si l'on considAère V ou O. Les deux solutions doivent nAécessairement partir du mAême puits de dAépart, donc $v_1 = o_1$. Pour simplifier, supposons qu'un remplissage s'effectue aussi au point d'arrivée, donc $v_n = o_n$. Pour que O soit une solution meilleure que V, elle doit possAéder strictement moins de puits de remplissage que V. Les deux cas de base suivants sont facilement vAérifiables :

a. Si V ne possAède que deux puits de remplissage, alors O ne peut pas offrir une meilleure solution, car cela voudrait dire que le puits final est identique au puits de dAépart (cas trivial).

b. Si V ne possAède que trois puits de remplissage ($V = \{v_1, v_2, v_3\}$), alors O devrait thAéoriquement possAéder seulement deux puits de remplissage ($O = \{o_1, o_2\}$). Ceci est impossible, puisque si le trajet de o_1 (puits de dAépart) Aà o_2 (puits final) Aétait possible (moins de K kilomètres), alors il aurait AétAé pris par V, Aà cause de son caractAère vorace.

Autrement, soit i la premiAère position pour laquelle V diffAère de O, oAù $1 < i < n$. Ceci implique nAécessairement que $o_i < v_i$, puisque la dAéfinition de l'approche vorace pour ce problAème est de toujours prendre la plus grande distance $\leq K$ entre les puits de remplissage. Par consAéquent, la distance entre v_i et v_{i+1} sera obligatoirement plus petite que celle entre o_i et v_{i+1} :

Chemin pour V : $\ldots \quad v_{i-1} \_\_\_\_\_ v_i \_\_\_\_\_ v_{i+1} \ldots$
Chemin pour O : $\ldots \quad o_{i-1} \_\_\_\_\_ o_i \ldots$

Puisque la distance v_i Aà v_{i+1} est la plus grande distance possible sans s'arrAêter pour remplir la gourde, alors le trajet entre o_i et v_{i+1} va nAécessiter le remplissage de la gourde Aà un puits supplAémentaire

o_j, oAù $j < i+1$. Donc, en partant du puits $v_{i-1} = o_{i-1}$, l'explorateur quittera le puits v_{i+1} en ayant rempli sa gourde au minimum trois fois ($\{v_{i-1}, v_i, v_{i+1}\}$ pour V et $\{o_{i-1}, o_i, o_j\}$ pour O). Il n'est donc pas possible de trouver une solution pour obtenir un nombre plus petit de remplissage entre v_{i-1} et v_{i+1}. Ce principe s'applique pour le prochain puits de remplissage v_k diffAérent de o_k et ce, jusqu'Aà ce que l'on atteigne le puits d'arrivAée (on peut le prouver par induction). Ceci implique donc qu'Aà la fin, le nombre de puits de remplissage de la solution O sera, au minimum, identique Aà celui de la solution V. Par consAéquent, cela contredit la dAéfinition de O, puisque le nombre de puits de remplissage de O ne pourra pas Aêtre plus petit que celui de V et O est considAérAé comme un ensemble optimal (Aà la limite, O sera une solution aussi bonne que V, mais non meilleure).

Solution à l'exercice 3.

Si on ignore la partie tri, on aura que, dans le pire cas, le nombre total d'appels aux opérations de manipulation d'ensembles disjoints, qu'on peut considérer comme nos opérations de base, sera comme suit :

- Exactement n appels dans la première boucle pour créer les n ensembles disjoints.
- Dans le pire cas, on aura m itérations de la boucle `TANTQUE`, puisque le pire cas consiste à examiner chacune des arêtes (il est possible que l'arête de plus grand poids fasse partie de l'arbre de recouvrement, par exemple, si cette arête relie un sommet isolé).

  Or, dans chaque itération, on aura au pire quatre (4) appels à des opérations de manipulation d'ensembles disjoints.

Au total, on aura donc $n + 4m$ appels aux opérations de base sur les ensembles. D'après la formule de Cormen et al., la complexité $W(m,n)$ d'une telle série d'appels sera alors $O((n+4m)\ \lg^*\ n)$. Par la définition de O, ceci signifie que pour des valeurs suffisamment grandes de m et n, on aura que l'inégalité suivante est satisfaite pour une constante c appropriée :

$$W(m,n) \quad \leq \quad c(n+4m)\ \lg^*\ n$$

Or, dans un graphe connexe, on a nécessairement que $n-1 \leq m \leq n(n-1)/2$, c'est-à-dire, $n \leq m+1 \leq 2m$ (pour $m \geq 1$). De plus, il est facile de voir (cf. la définition de $\lg^*$ dans les notes de cours), que $\lg^*\ 2m \leq 2\lg^*\ m$. On aura donc les inégalités suivantes :

$$\begin{aligned} W(m,n) &\leq c(n+4m)\ \lg^*\ n \\ &\leq c(2m+4m)\ \lg^*\ 2m \\ &\leq (c+6)m\ \lg^*\ 2m \\ &\leq 2(c+6)m\ \lg^*\ m \\ &\in O(m\ \lg^*\ m) \end{aligned}$$

Notons finalement qui si on inclut la partie tri, qui est $O(m\ \lg\ m)$, on pourra alors conclure que l'ensemble de l'algorithme de Kruskal est bien $O(m\ \lg\ m)$, puisque c'est le temps du tri qui dominera l'ensemble du temps d'exécution (l'initialisation des ensembles disjoints sera $O(n)$, alors que les m appels aux opérations de manipulation des ensembles disjoints seront $O(m\ \lg^*\ n)$.

Solution à l'exercice 4.

La complexité pour une série de n opérations sera $\Theta(n)$. Ce résultat peut être obtenu par une forme d'*analyse amortie*, en réalisant que l'opération `vider`, bien qu'elle soit $\Theta(n)$ dans le pire cas, ne peut demander ce travail qui si la pile n'a pas été vidée au préalable. Ainsi, si on effectue k appels à `vider` parmi la séquence des n appels, aucun de ces appels ne demandera exactement un travail $\Theta(n)$, puisque la pile ne contiendra pas n éléments au moment où l'appel à `vider` s'effectuera.

Plus précisément, pour simplifier, supposons qu'on ne considére que des appels à `empiler` et à `vider`. Choisissons ensuite les affectations à `elems[i]` comme opération barométrique. Soit alors une série de n appels à `empiler` et `vider`. On peut voir que l'on aura les correspondances suivantes, puisqu'un élément empilé ne peut être dépilé qu'une seule fois :

Appels totaux	Appels à `vider`	Appels à `empiler`	Nombre exact d'affectations
n	1	$n-1$	$2(n-1)$
n	2	$n-2$	$2(n-2)$
n	...	...	...
n	k	$n-k$	$2(n-k)$

Soulignons qu'une telle analyse nous permet donc de conclure que le *temps moyen* d'exécution d'une opération de manipulation d'une telle pile est $\Theta(1)$, et ce sans recours aux probabilités — $\Theta(n)$ pour n opérations, donc $\Theta(1)$ par opération.

Solution à l'exercice 5.

a. Tri par sélection.

b. Tri par insertion

Solution à l'exercice 6.

```
type Pile = ptr PileRec;
type PileRec = rec( FilePriorite fp; int prochain; );

procedure creer() returns Pile p
{
  p = new(PileRec);
  p^.fp = creer();
  p^.prochain = 0;
}

procedure estVide( Pile p ) returns bool r
{ r = estVide(p^.fp); }

procedure sommet( Pile p ) returns int elem
{ elem = maxElement(p^.fp); }

procedure empiler( Pile p, int elem )
{ inserer(p^.fp, p^.prochain++, elem); }

procedure depiler( Pile p )
{
  int cle, elem;
  retirerMaxElement( p^.fp, cle, elem );
}
```

Exercices tirés du manuel.

Note : Les solutions des exercices 1, 2, 3 et 5 ont été développées, en partie, par Éric Terroux.

- 1. Solution discutée en séance d'exercices.
- 2.

Étape	**Arête choisie**	Y
0		$\{\}$
1	$v_1 - v_4$	$\{v_1, v_4\}$
2	$v_4 - v_8$	$\{v_1, v_4, v_8\}$
3	$v_8 - v_9$	$\{v_1, v_4, v_8 : v_9\}$
4	$v_4 - v_5$	$\{v_1, v_4 : v_5, v_8 : v_9\}$
5	$v_9 - v_{10}$	$\{v_1, v_4 : v_5, v_8 : v_{10}\}$
6	$v_{10} - v_6$	$\{v_1, v_4 : v_6, v_8 : v_{10}\}$
7	$v_4 - v_3$	$\{v_1, v_3 : v_6, v_8 : v_{10}\}$
8	$v_3 - v_7$	$\{v_1, v_3 : v_{10}\}$
9	$v_1 - v_2$	$\{v_1 : v_{10}\}$

  L'arbre de recouvrement (l'arbre sous-tendant) minimal est donc un arbre ayant un poids total de 107.

- 3. Une multitude de solutions sont possibles. L'une des façons est de débuter avec *presqu'un arbre*, plus précisément, en incluant tous les sommets sauf un seul. On ajoute ensuite des arêtes de poids *identiques* reliant le sommet isolé à quelques-uns des autres sommets.

  Exemple : Soit $G = (V, E)$, tels que

$$\begin{aligned} V &= \{v_1, v_2, v_3, v_4\} \\ E &= \{(v_2, v_3, 20), (v_3, v_4, 30)\} \end{aligned}$$

  Ajoutons ensuite les arêtes $E' = \{(v_1, v_2, 10), (v_1, v_3, 10)\}$.

  Deux arbres sous-tendants minimums possibles pour le graphe $G = (V, E \cup E')$ seront alors les suivants :

$$\{(v_1, v_2, 10), (v_2, v_3, 20), (v_3, v_4, 30)\}$$
$$\{(v_1, v_3, 10), (v_2, v_3, 20), (v_3, v_4, 30)\}$$

- 5. Plutôt que de toujours exécuter la boucle `repeat` un nombre fixe d'itérations, il faudrait plutôt terminer lorsqu'on reconnait qu'il devient impossible d'ajouter une nouvelle arête à l'ensemble des noeuds Y. Plus précisément, à la sortie de la première boucle `for` à l'intérieur du `repeat`, si `min` est toujours égal à ∞, alors c'est qu'aucune autre arête n'a pu être ajoutée, ce qui signifie que le graphe n'est pas connexe.

  La complexité de l'algorithme résultant sera la même que l'algorithme initial, à savoir $\Theta(n^2)$ (le pire cas est lorsque le graphe est connexe).

- 26. Solution omise.

- 30. Sans perte de généralité (par renommage des items), supposons que l'on ait les items et poids suivants, ordonnés en ordre *décroissant* de ratio bénéfice par unité de poids :

Item	1	2	...	n
Poids	p_1	p_2	...	p_n
Bénéfice	b_1	b_2	...	b_n
Ratio b_i/p_i	r_1	r_2	...	r_n

Pour simplifier, supposons aussi que $r_i \neq r_j$, pour $i \neq j$ (autrement, rien ne distingue les deux items du point de vue de l'algorithme et on peut donc les considérer comme le même item, avec une plus grande quantité disponible).

La solution S obtenue par l'algorithme vorace sera alors de la forme suivante : on choisit les items $1, 2, \ldots, k$ ($k \leq n$) tels que le poids choisi pour l'item i sera simplement p_i, sauf pour le dernier élément dont le poids sera $\alpha \times p_k$, pour $0 < \alpha \leq 1$.

Montrons que cette solution S est nécessairement optimale. Soit S' une autre solution qui contient d'autres items. Pour la solution S', il existe nécessairement un k' tel que $S = S'$ pour les $k' - 1$ premiers items, et la quantité de l'item k' dans la solution S est différente de celle dans la solution S'.

Deux cas sont possibles :

a. $k' = k$: dans ce cas, la quantité de l'item k' pour S' est nécessairement plus petite que pour S, parce que sinon on dépasserait le poids du sac (l'algorithme S s'est arrêté avec cette quantité pour l'item k parce que le sac était plein).
b. $k' < k$: là aussi, la quantité doit être inférieure, puisqu'elle *ne peut pas être supérieure*, l'item k' dans la solution S ayant été sélectionné au complet.

Dans les deux cas, la quantité de l'item k' dans S' est donc inférieure à celle du même item dans la solution S. Cette quantité a nécessairement été remplacée par une quantité d'un autre item l (sinon le sac ne serait pas plein). Or, la ratio bénéfice par unité de poids de cet autre item l est nécessairement inférieur à celui de l'item k', puisqu'on a supposé que les items étaient ordonnés selon leur ratio et qu'ils n'étaient pas de ratios égaux. Donc, le bénéfice de la solution S' est nécessairement inférieur à celui pour S, puisqu'on remplace une quantité d'un certain item par une quantité d'un autre item apportant moins de bénéfice.