Exercices INF7440 : série #5

1. Soit n tâches (*jobs*) $j_1, \ldots, j_n$ avec des temps d'exécution $t_1, \ldots, t_n$. On veut ordonner (céduler) l'exécution de ces tâches sur une machine uniprocesseur de façon à *minimiser le temps moyen* requis pour qu'une tâche soit finalement complétée.

Par exemple :

Tâche	**Temps requis**
j_1	15
j_2	8
j_3	3
j_4	10

Deux ordonnancements possibles sont les suivants :

- j_1, j_2, j_3, j_4: temps total = 36, temps moyen de fin d'exécution = 25 ([15 + (15+8) + (15+8+3) + (15+8+3+10)]/4).
- j_3, j_2, j_4, j_1: temps total = 36, temps moyen de fin d'exécution = 17.75 ([3 + (3+8) + (3+8+10) + (3+8+10+15)]/4).

a. Donnez un algorithme vorace permettant d'obtenir un ordonnancement optimal de n tâches. L'en-tête de votre algorithme devrait être le suivant :

```
PROCEDURE cedulerTaches( t: sequence{Nat} ): sequence{Nat}
```

b. Quelle est la complexité de votre algorithme?

c. Qu'est-ce qui assure qu'une solution vorace sera optimale?

2. Un explorateur désire traverser le désert en ne transportant qu'une seule bouteille d'eau. Il a à sa disposition une carte qui indique tous les endroits où de l'eau est disponible. Supposons qu'il peut réussir à marcher au plus k kilomètres avec une bouteille pleine d'eau. Concevez un algorithme vorace lui permettant de déterminer à quels points d'eau il devrait remplir sa bouteille, et ce en minimisant le nombre de remplissages. Justifiez brièvement pourquoi la solution produite par votre algorithme est optimale.

3. L'analyse amortie des opérations de manipulation d'ensembles disjoints décrite dans le livre de Cormen, Leiserson et Rivest permet de conclure que la complexité d'une série de m' appels à ces opérations sera $O(m' \lg^* n)$. [1] Or, dans cette formule, m' et n doivent être interprétées comme suit :

[1] Ici, m' est utilisée plutôt que m pour éviter, lors de l'analyse de l'algorithme de Kruskal, toute confusion avec m, le nombre d'arêtes dans le graphe.

- n = nombre d'éléments de l'ensemble de base.

  Dans la présentation de Cormen *et al.*, les n ensembles disjoints de départ, plutôt que d'être créés à l'aide d'une unique opération d'initialisation, le sont à l'aide de n appels à une opération de type `creerSingleton`.

- m' = nombre total d'appels à des opérations de manipulation des ensembles disjoints.

  Toujours dans la formulation de Cormen *et al.*, cette série d'appels doit *aussi inclure* les appels initiaux à `creerSingleton`. On a donc nécessairement $m' \geq n$.

Dans le contexte de l'algorithme de Kruskal, pour un graphe $G = (V, E)$, suppoposons alors $|V| = n$ et $|E| = m$. Supposons aussi, pour utiliser une présentation semblable à celle de Cormen *et al.*, que l'algorithme de Kruskal soit exprimé comme suit :

```
aretes <- Trier les arêtes E de G (en ordre croissant de poids)
F <- {}
POUR i <- 1 A n FAIRE
  creerSingleton( i )
FIN
TANTQUE |F| < n FAIRE
  e <- aretes[1]
  aretes <- tail aretes
  (i, j) <- index des sommets reliés par l'arête e
  p <- find(i)
  q <- find(j)
  SI  NON equal(p, q) ALORS
    merge(p, q)
    F <- F U {e}
  FIN
FIN
```

Montrez que, *si on ignore la partie tri de l'algorithme*, le reste de l'algorithme de Kruskal est bien $O(m \lg^* m)$.

4. Soit la classe Java suivante, qui définit une classe d'objets `Pile` :

```
public class Pile {
  private final int MAXELEMS = 100;
  private Object[] elems;
  private int nbElements;

  public Pile() {
    elems = new Object[MAXELEMS];
    nbElements = 0;
  }

  public void empiler( Object o ) {
    elems[nbElements++] = o;
  }

  public void depiler() {
    elems[--nbElements] = null;
  }

  public Object sommet() {
    return( elems[nbElements-1] );
  }

  public boolean estVide() {
    return( nbElements == 0 );
  }

  public void vider() {
    int i;
    for( i = 0; i < nbElements; i++ ) {
      elems[i] = null;
    }
    nbElements = 0;
  }
}
```

Supposons que, après avoir créé une pile `p` (avec `p = new Pile();`), on exécute une série de n opérations sur cette pile. Quelle sera, dans le pire cas, la complexité asymptotique du temps d'exécution pour cette série de n opérations? Que peut-on alors conclure sur *le temps moyen* d'exécution d'une opération sur une telle pile?

5. Soit l'algorithme de tri présenté dans les notes de cours et utilisant les files de priorité. À quel algorithme de tri bien connu correspond le tri avec file de priorité lorsque les files de priorités sont réalisées de la façon indiquée :

a. Files de priorité réalisées à l'aide de séquences *non ordonnées* d'items (par ex., tableau non ordonnée d'éléments).

b. Files de priorité réalisées à l'aide de séquences *ordonnées* d'items (par ex., liste chaînées d'éléments).

```
type Pile = ptr ...;

procedure creer() returns Pile p
# POSTCONDITION
#   new(p)
#   p.elems = []

procedure estVide( Pile p ) returns bool r
# POSTCONDITION
#   r <=> p.elems = []

procedure sommet( Pile p ) returns int elem
# PRECONDITION
#   ~estVide(p)
# POSTCONDITION
#   elem = p.elems[1]

procedure empiler( Pile p, int elem )
# POSTCONDITION
#   p.elems = add( elem, p'.elems )

procedure depiler( Pile p )
# PRECONDITION
#   ~estVide(p)
# POSTCONDITION
#   p.elems = tail(p'.elems)
```

Figure 1: Spécification d'un type abstrait pour des piles d'entiers

6. Soit le type abstrait suivant définissant des `Piles` (d'entiers) présenté à la Figure 1 (à l'aide d'un type et de procédures MPD). Donnez une mise en oeuvre de ces piles *qui utilise les files de priorité*.

Exercices tirés du manuel.

Exercices du Chapitre 4 (pp. 181–186)

- 1.
- 2.
- 3.
- 5.
- 35 (26 dans la première édition).
- 39 (30 dans la première édition).