Exercices : solutions de la série #4

Solution à l'exercice 1.

Le bénéfice maximum est de 154. Les items à inclure sont les items 1, 2 et 4, pour un poids maximum de 10.

Soulignons qu'une solution *vorace* produirait plutôt le résultat suivant : items 2 (ratio bénéfice/unité de poids = 16), puis item 1 (ratio 15) pour un bénéfice total de 110 et un poids de 7, ce qui n'est clairement pas optimal.

Solution à l'exercice 2.

Définissons par $M[i,j]$ le coût minimum pour aller de la ville v_i à la ville v_j.
L'idée est alors d'examiner toutes les façons possibles d'aller, de façon optimale, de la ville v_i à une ville v_k intermédiaire, puis de faire le reste du trajet de façon directe entre v_k et v_j — en d'autres mots, v_k est le dernier arrêt avant v_j.

- $M[i,i] = 0$, pour $1 \leq i \leq n$
- $M[i,j] = \min_{i \leq k < j} (M[i,k] + c(k,j))$, pour $1 \leq i < j \leq n$

Une solution paresseuse — tant en termes de codage (très court) qu'en termes de langage utilisé (puisqu'il s'agit d'une solution en Haskell, un langage fonctionnel paresseux ;) — est présentée dans l'algorithme 1.

```
coutMin n couts =
  let
    m = array ((1, 1), (n, n)) (
         [((i, i), 0) | i <- [1..n]]
         ++
         [((i, j), cout i j) | i <- [1..n], j <- [1..n], i < j]
        )
        where
          cout i j = minimum [ m!(i, k) + couts!(k, j) | k <- [i..j-1]]
  in
    m!(1, n)
```

Algorithme 1: Solution Haskell (langage fonctionnel) au problème des villes

Une solution impérative et itérative est présentée dans l'algorithme 2.

```
procedure minCoutPD( int C[*,*], int n ) returns int cout
{
  int M[n, n];

  for [i = 1 to n] {
    M[i, i] = 0;
  }
  for [dist = 1 to n-1] {
    for [i = 1 to n, j = 1 to n st i < j & j == i+dist] {
      M[i, j] = high(int);
      for [k = i to j-1] {
        M[i, j] = min( M[i, j], M[i, k] + C[k, j] );
      }
    }
  }
  cout = M[1, n];
}
```

Algorithme 2: Algorithme itératif de programmation dynamique pour le problème des villes

Sélectionnons les affectations à `M[i, j]` comme opération barométrique. Le nombre total d'opérations barométriques sera alors le suivant, en constatant que le nombre d'index `i` et `j` satisfaisant la condition `st` est de `dist` (nombre d'éléments de la diagonale) et que `j-i=dist` :

$$\begin{aligned} B(n) &= \sum_{i=1}^{n} 1 + \sum_{dist=1}^{n-1} \sum_{l=1}^{dist} (1 + (j - 1 - i + 1)) \\ &= \sum_{i=1}^{n} 1 + \sum_{dist=1}^{n-1} [dist * (j - i + 1)] \\ &= \sum_{i=1}^{n} 1 + \sum_{dist=1}^{n-1} [dist * (dist + 1)] \\ &\in \Theta(n^3) \end{aligned}$$

Solution à l'exercice 3.

a. La distance est de 3. La matrice résultante est la suivante :

```
      C  H  A  T  S
   [0, 1, 2, 3, 4, 5]
 C [1, 0, 1, 2, 3, 4]
 H [2, 1, 0, 1, 2, 3]
 I [3, 2, 1, 1, 2, 3]
 E [4, 3, 2, 2, 2, 3]
 N [5, 4, 3, 3, 3, 3]
```

b.

$$\begin{aligned} C(0,0) &= 0 \\ C(i,0) &= C(i-1,0) + \text{coût}_{sup}(A[i]) \\ C(0,j) &= C(0,j-1) + \text{coût}_{ins}(B[j]) \end{aligned}$$

$$C(i,j) \quad = \quad \texttt{min} \begin{cases} C(i-1,j) + \text{coût}_{sup}(A[i]) \\ C(i,j-1) + \text{coût}_{ins}(B[j]) \\ C(i-1,j-1) + \text{coût}_{sup}(A[i]) + \text{coût}_{ins}(B[j]) \text{ si } A[i] \neq B[j] \\ C(i-1,j-1) \text{ si } A[i] = B[j] \end{cases}$$

Note : une solution équivalente serait obtenue en supprimant la troisième clause (cas $A[i] \neq B[j]$). Dans ce cas, les clauses d'insertion et suppression déjà présentes feraient en sorte, implicitement, qu'un caractère serait substitué par un autre par le biais d'une suppression suivie d'une insertion.

c. La distance est de 6. La matrice résultante est la suivante :

```
      C  H  A  T  S
  [0, 1, 2, 3, 4, 5]
C [1, 0, 1, 2, 3, 4]
H [2, 1, 0, 1, 2, 3]
I [3, 2, 1, 2, 3, 4]
E [4, 3, 2, 3, 4, 5]
N [5, 4, 3, 4, 5, 6]
```

d.
```
procedure distance( char ch1[*], int n1, char ch2[*], int n2 )
          returns int dist
{
  int D[0:n1][0:n2];

  D[0][0] = 0;
  for [i = 1 to n1] { D[i][0] = i; }
  for [j = 1 to n2] { D[0][j] = j; }

  for [i = 1 to n1, j = 1 to n2] {
     if (ch1[i] == ch2[j]) {
        D[i][j] = min( D[i-1][j]+1, D[i][j-1]+1, D[i-1][j-1] );
     } else {
        D[i][j] = min( D[i-1][j]+1, D[i][j-1]+1, D[i-1][j-1]+2 );
     }
  }
  dist = D[n1][n2];
}
```

La complexité de cet algorithme est $\Theta(n1 \times n2)$, donc linéaire dans le produit des tailles des deux chaînes. Si on suppose que les deux chaînes sont de même longueur n, on obtient alors une complexité quadratique $\Theta(n^2)$.

Exercices tirés du manuel.

- 12. Le coût optimal est de 1320.
- 29. Pour simplifier, supposons que les nombres entiers sont représentés en binaire (base 2). L'algorithme est $\Theta(nk)$. Notons par t la taille du problème. On aura ici $t = \lg\ n + \lg\ k$. La complexité de l'algorithme devient donc $\Theta(nk) = \Theta(2^{\lg\ (nk)}) = \Theta(2^{\lg\ n + \lg\ k}) = \Theta(2^t)$. L'algorithme est donc exponentiel en termes de la taille du problème lorsqu'exprimée en fonction du nombre de bits nécessaires pour représenter les arguments d'entrée.

- 33.
  - Solution naïve (en Haskell) :

```
sommeMax elems =
  let
    n = length elems
  in
    maximum [sum [elems!!(k-1) | k <- [i..j]] | i <- [1..n], j <- [i..n]]
```

    Cette solution est $\Theta(n^3)$.
  - Solution avec programmation dynamique (toujours en Haskell) :

```
sommeMax elems =
  let
    n = length elems
    -- La position (i, j) du tableau nous donne la somme des elements
    -- i a j (inclusivement) de la liste initiale
    m = array ((1, 1), (n, n)) (
         [((i, i), elems!!(i-1)) | i <- [1..n]]
         ++
         [((i, j), m!(i, j-1) + elems!!(j-1)) | i <- [1..n], j <- [i+1..n]]
        )
  in
    maximum [m!(i,j) | i <- [1..n], j <- [i..n]]
```

    Solution itérative (programmation dynamique ascendante) en MPD :

```
procedure sommeMax( int elems[*], int n ) returns int s
{
  int m[n][n];

  for [i = 1 to n] {
    m[i][i] = elems[i];
    for [j = i+1 to n] {
      m[i][j] = m[i][j-1] + elems[j];
    }
  }
  s = 0;
  for [i = 1 to n, j = i to n] {
    s = max( s, m[i][j] );
  }
}
```

    Cette solution est $\Theta(n^2)$ et s'effectue en deux passes :

    a. Une première passe ($\Theta(n^2)$) qui construit le tableau `m` tel que `m[i, j]` contient la somme des éléments `i` à `j` inclusivement.
    b. Une deuxième passe (elle aussi $\Theta(n^2)$) qui trouve le maximum parmi les éléments du tableau `m`.

    Notons que tous les éléments du tableau n'ont pas besoin d'être calculés, uniquement ceux tels que `i <= j`.