Corrigé du test algorithmique n°1

Exercice 1 : Deux calculs simples

Deux boucles itératives simples, avec un accumulateur pour contenir les résultats intermédiaires.

1) (a) Par définition, $n^k = 1 \times \underbrace{n \times \cdots \times n}_{k \text{ facteurs } n}$. On en déduit l'algorithme :

```
fonction puissance(n,k : entiers) : entier;

  // puissance(n,k) calcule l'entier n^k
  // acc : entier contenant les produits
     intermédiaires
  // i : entier compteur de boucle

  acc <- 1
  // après l'itération n°i, acc contient n^i
  pour i de 1 à k faire acc <- acc * n
  retourner acc
```

Pour vérifier que cet algorithme effectue le bon calcul, il suffit de constater qu'à l'itération i, `acc` contient n^i.

(b) Le corps de boucle est exécuté pour toutes les valeurs de i entre 1 et k, donc k fois. On effectue donc k multiplications.

On pourrait faire une multiplication de moins en initialisant acc à n, mais alors on perdrait le cas où $k = 0$.

2) (a) C'est le même algorithme, seuls les calculs changent :

```
fonction factorielle(n : entier) : entier;

  // factorielle(n) calcule n!=1*2*...*n
  // acc : entier contenant les produits
     intermédiaires
  // i : entier compteur de boucle

  acc <- 1
  // après l'itération n°i, acc contient i!
  pour i de 1 à n faire acc <- acc * i
  retourner acc
```

Il est facile de voir qu'à l'itération i, `acc` contient $i!$.

(b) Encore une fois, le corps de boucle est exécuté n fois, donc on effectue n multiplications.

EXERCICE 2 : PLUS HAUTE MARCHE

1) L'algorithme consiste à calculer les différences entre deux éléments consécutifs, et à trouver la plus grande. C'est un exercice de recherche de maximum. `tableau` est le type du tableau d'altitudes.

```
fonction haut_max(m : tableau) : entier;

  // haut_max(m) renvoie la plus grande différence entre deux
  // cases consécutives du tableau d'entiers m.

  // hm : entier contenant la plus grande hauteur rencontrée
  // i : entier compteur de boucle

  hm <- m[1]   // on démarre de 0
  pour i de 2 à n faire
  // après la i-ème itération, hm contient la plus grande
  // différence trouvée dans le tableau m entre les indices
  // 1 et i.
     hm <- max(hm,m[i]-m[i-1])
  retourner hm
```

2) Les comparaisons sont effectuées lors de l'exécution des instructions `max`. On en effectue une par passage dans la boucle, soit $N-1$. Il en est de même du nombre de soustractions.

3) Dans cet algorithme, il faut à la fois se souvenir de la hauteur maximale déjà rencontrée, comme précédemment, et de l'indice correspondant.

```
fonction ind_haut_max(m : tableau) : entier;

  // ind_haut_max(m) renvoie l'indice de départ de la plus
  // grande différence entre deux cases consécutives du tableau
     m.

  // hm et i comme précédemment.
  // im : entier contenant l'indice de la hauteur maximale.
  hm <- m[1]
  im <- 0     // indice du point de départ du plus grand pas
  pour i de 2 à n faire
     h <- m[i] - m[i-1]    // h est la hauteur du pas courant
     si h > hm alors
        hm <- h
        im <- i - 1        // hm correspond à m[im+1]-m[im]
  retourner im
```

EXERCICE 3 : RECHERCHE DANS UN TABLEAU, DANS UNE MATRICE

Dans tout cet exercice, le type des tableaux à une dimension sera noté `tableau`, celui des tableaux à deux dimensions `matrice`.

1) **Exploration itérative du tableau et de la matrice**

(a) Encore une fois, on explore entièrement un tableau, une boucle `pour` et un compteur devraient faire l'affaire.

```
fonction nb_zero_tableau(t : tableau) : entier;

  // nb_zero_tableau(t) renvoie le nombre de cases du tableau
       t
  // contenant 0.

  // nbzero : entier contenant le nombre de zéros au cours de
  // l'exploration.
  // i : entier compteur de boucle.

  nbzero <- 0
  pour i de 1 à N faire
  si t[i] = 0 alors
     nbzero <- nbzero + 1
  retourner nbzero
```

(b) Le corps de boucle, contenant une comparaison, est effectué `N` fois, on fait donc `N` comparaisons.

(c) Pour une matrice, on emploie une paire de boucles imbriquées, ce n'est pas beaucoup plus compliqué que pour un tableau à une dimension.

```
fonction nb_zero_matrice(m : matrice) : entier;

  // nb_zero_matrice(m) renvoie le nombre de cases de la
      matrice m
  // contenant 0.

  // nbzero : entier contenant le nombre de zéros au cours de
  // l'exploration.
  // i : entier compteur de boucle, numéro de la ligne en cours
      .
  // j : entier compteur de boucle, numéro de la colonne en
      cours.

  nbzero <- 0
  // On explore la matrice ligne par ligne
  pour i de 1 à N faire
     pour j de 1 à M faire
        si t[i] = 0 alors
           nbzero <- nbzero + 1
  retourner nbzero
```

La boucle interne fait `M` comparaisons, elle est elle-même exécutée `N` fois, on effectue donc au total `NM` comparaisons.

À l'étape `(i,j)` de la double boucle, `nbzero` contient le nombre de zéro dans les `i-1` premières lignes, ainsi que dans les `j` premières cases de la ligne `i`.

2) **Exploration conditionnelle du tableau et de la matrice**

(a) La boucle est désormais conditionnelle. On s'arrête dès qu'on a atteint la fin du tableau, **ou** si on a trouvé le nombre demandé de zéros.

```
fonction au_moins_k_zero_tableau(t : tableau; k : entier) :
   booléen;

  // au_moins_k_zero_tableau(t,k) renvoie Vrai si le tableau t
  // contient au moins k cases égales à 0, Faux sinon.

  // nbzero et i comme dans la première partie.
  nbzero <- 0
  i <- 1
  // La condition de sortie :
  // * on est arrivé à la fin du tableau (i>N)
  // * ou on a rencontré suffisamment de zéros (nbzero>=k).
  tant_que (i <= N) et (nbzero < k) faire
     si t[i] = 0 alors nbzero <- nbzero + 1
     i <- i + 1
  // On renvoie le résultat du test (nbzero >= k).
  renvoyer (nbzero >= k)
```

(b) Comme le nombre de zéro peut être déjà atteint dès le départ (`k=0`), ou jamais atteint, on ne peut prédire le nombre d'itérations de la boucle à l'avance (c'est-à-dire sans connaître le résultat !). On sait par contre que chaque itération demande trois comparaisons (les tests `i<=N` et `nbzero<k`, et le test `t[i]=0`), on exécutera de 2 comparaisons (cas où `k=0`) à 3`N`+1 comparaisons (cas où on atteint la fin du tableau.

On pourrait effectuer un test à chaque étape pour vérifier si il reste assez de cases pour remplir la mission. Par exemple, si `k=N`, dès qu'on rencontre une case contenant autre chose qu'un 0, on sait qu'on ne peut plus compter suffisamment de cases égales à 0. Mais cela alourdit le corps de boucle, et ne peut donc être envisagé que si ce cas est suffisamment fréquent. Seule une étude statistique permet de vérifier si le gain attendu compense l'allongement des calculs à chaque étape.

(c) L'écriture d'une boucle conditionnelle pour explorer une matrice est un peu plus compliquée. Voici un exemple parmi d'autres :

```
fonction au_moins_k_zero_matrice(m : matrice; k : entier) :
   booléen;

  // au_moins_k_zero_matrice(m,k) renvoie Vrai si la matrice m
  // contient au moins k cases égales à 0, Faux sinon.

  // nbzero, i et j comme précédemment.

  // Initialisation.
  nbzero <- 0
  i <- 1
  j <- 1
  // La matrice est explorée ligne par ligne.
  // Condition de sortie :
  // * on est arrivé à la fin de la matrice (i>N, on a dépassé la
  // dernière ligne)
  // * ou on a rencontré suffisamment de zéros (nbzero>=k).
  tant_que (i <= N) et (nbzero < k) faire
     si m[i,j] = 0 alors nbzero <- nbzero + 1
     j <- j + 1          // Cette partie calcule les coordonnées de
         la case
     si j > M alors      // suivante. On augmente l'indice de
        colonne à
        j <- 1           // moins qu'on soit à la dernière colonne.
        i <- i+1         // (i,N) -> (i+1,1), (i,j) -> i,j+1) si j<
           N.
  // On renvoie le résultat du test (nbzero >= k).
  renvoyer (nbzero >= k)
```

Comme indiqué, la matrice est explorée ligne par ligne (`i` est le compteur de ligne, `j` le compteur de colonne). La case suivant la case d'indice `[i,j]` est la case `[i,j+1]` *à moins que* `j` soit égal à `M` (fin de ligne), auquel cas on passe à la ligne suivante et on réinitialise `j` à 1. Le calcul se termine lorsque `i` prend la valeur `N+1`, juste après avoir exploré la case d'indice `N,M`.