Corrigé du TD 1

Exercice 1. *Suivi d'un traileur.*

Question 1. *Écrire une procédure qui prend en argument un tableau de distances, deux points de passage* `i` *et* `j`, `i` $<$ `j`, *et retourne la distance pour aller du point* `i` *au point* `j`.

```
distancePartielle (Distance : tableau d'entiers, i : entier, j : entier) : entier
   total : entier ; total = 0
   pour k de i+1 à j faire
      total = total + Distance[k]
   retourner total
```

`Distance[k]` contient la distance du point `k-1` au point `k`, donc en sommant de `k=i+1` à `k=j` on mesure bien la distance du point `i` au point `j`. On utilise une variable `total` pour enregistrer les sommes partielles au fur et à mesure qu'on parcourt le tableau.

Question 2. *Écrire une procédure qui prend en argument un tableau de temps de passage, deux points de passage* `i` *et* `j`, `i` $<$ `j`, *et retourne le temps mis pour aller du point de passage* `i` *au point de passage* `j`.

```
tempsPartiel (Temps : tableau d'entiers, i : entier, j : entier) : entier
   retourner Temps[j] - Temps[i]
```

Question 3. *Utiliser les deux procédures précédentes pour écrire une procédure qui prend en argument les tableaux des distances et temps de passage et deux points de passage* `i` *et* `j`, `i` $<$ `j`, *et retourne la vitesse du coureur entre les points* `i` *et* `j`.

On calcule la vitesse en divisant la distance par le temps :

```
vitessePartielle (Distance : tableau d'entiers, Temps : tableau d'entiers,
                  i : entier, j : entier) : réel
   retourner distancePartielle(Distance, i, j) / tempsPartiel(Temps, i, j)
```

Question 4. *Écrire une procédure qui prend en argument le tableau des altitudes, sa longueur* `n`, *et retourne le dénivelé positif total.*

```
déniveléPositif (Altitude : tableau d'entiers, n : entier) : entier
   total : entier ; total = 0
   pour i de 1 à n-1 faire
      si Altitude[i] > Altitude[i-1] alors
         total = total + Altitude[i] - Altitude[i-1]
   retourner total
```

Question 5. *Écrire une procédure qui prend en argument le tableau des altitudes, sa longueur* `n`, *et retourne le dénivelé de la plus grande montée continue.*

On parcourt le tableau en entier. Au cours du parcours on tiens à jour deux variables temporaire : `montée` qui contient le dénivelé de la montée en cours (si on est en train de monter), et `max` qui contient le dénivelé de la plus grande montée jusqu'à ce point. Le test `si ... alors` regarde si on a monté depuis le dernier point de passage, si oui on ajoute le dénivelé à `montée`, sinon on remet `montée` à `0`. À chaque étape on s'assure que `max` reste égal à la plus grande montée.

```
grandeMontée (Altitude : tableau d'entiers, n : entier) : entier
   max, montée : entier ; max = 0 ; montée = 0
   pour i de 1 à n-1 faire
      si Altitude[i] >= Altitude[i-1] alors
         montée = montée + Altitude[i] - Altitude[i-1]
      sinon
         montée = 0
      si montée > max alors
         max = montée
   retourner max
```

Exercice 2. *On considère la procédure suivante.*

```
mystere(n : entier) : entier
    si n == 0 alors
        retourner 2
    sinon
        retourner mystere(n-1)*mystere(n-1)
```

Question 1. *Quelle fonction mathématique est calculée par* `mystere` *?*

Cette procédure calcule la suite $(u_n)_n$ qui vérifie $u_0 = 2$, $u_n = u_{n-1}^2$. La suite $l_n = \ln(u_n)$ vérifie $l_0 = \ln(2)$, $l_n = 2l_{n-1}$: c'est une suite géométrique de raison 2, donc $l_n = 2^n l_0$. Ainsi $u_n = \exp(2^n \ln(2)) = 2^{(2^n)}$.

Question 2. *Calculer le coût $C(n)$ de la procédure* `mystere`*, défini comme étant le nombre de multiplications effectuées.*

On a $C(0) = 0$, $C(n) = 2C(n-1) + 1$.
On a bien $C(0) = 2^0 - 1$ et, si $C(n) = 2^n - 1$, alors $C(n+1) = 2(2^n - 1) + 1 = 2^{n+1} - 1$.

Question 3. *Proposez une procédure ayant un meilleur coût, donnez ce coût.*

Il faut faire un seul appel récurent au lieu de 2, en stockant le résultat dans une variable `tmp` :

```
mystere2 (n : entier) : entier
    tmp : entier
    si n=0 alors
        retourner 2
    sinon
        tmp = mystere2(n-1)
        retourner tmp * tmp
```

Le nouveau coût $D(n)$ vérifie $D(0) = 0$, $D(n) = D(n-1) + 1$. C'est une suite arithmétique : on a $D(n) = n$.

Exercice 3.

Question 1. *Écrivez une procédure qui renvoie la position du premier élément x dans un tableau d'entiers si x est présent et -1 sinon.*

Voici une possibilité parmi d'autres. On utilise une boucle `tant que`, ce qui permet de s'arêter dès qu'on a trouvé `x`. La condition `r==-1` signifie « on n'a pas encore trouvé `x` ».

```
rechercheElement (tab : tableau d'entiers, taille : entier, x : entier) : entier
    i : entier ; i = 0  # compteur de boucle
    r : entier ; r = -1 # résultat à retourner
    tant que i < taille et r == -1 faire
        si tab[i] == x alors
            r = i
        i = i + 1
    retourner r
```

Question 2. *Déterminer, en fonction de la taille du tableau, le coût maximum de la procédure, définie comme étant le nombre d'éléments examinés.*

On a $C(taille) = taille$.

Question 3. *On suppose maintenant que le tableau est trié. Proposez un algorithme dichotomique pour rechercher un élément x.*

Voici une procédure qu'on appelle sous la forme `rechercheDicho(T,n,0,n,x)` pour chercher `x` dans un tableau de taille `n`. Les arguments `a` et `b` servent lors des appels récursifs.

```
rechercheDicho (tab : tableau d'entiers, taille : entier,
                a : entier, b : entier, x : entier) : entier
    # on vérifie (lors du premier appel) si x est au début ou à la fin du tableau :
    si a == 0 alors
        si tab[a] == x alors
            retourner a
```

```
    si b == taille - 1 alors
        si tab[b] == x alors
            retourner b
    # sinon on cherche x au milieu m du tableau :
    m : entier ; m = (a+b) div 2
    si m == a alors
        # on est dans ce cas si b=a ou b=a+1
        # il n'y a rien entre les deux, donc x ne figure pas dans le tableau
        retourner -1
    y = tab[m]
    si y == x alors
        retourner m
    # comme le tableau est rangé par ordre croissant, on sait en comparant tab[m] et x
    # s'il faut continuer à chercher à gauche ou à droite de m :
    si y < x alors
        retourner recherche(tab, taille, m, b, x)
    sinon
        retourner recherche(tab, taille, a, m, x)
```

Si $D(taille)$ est le coût maximal de cette fonction, alors $D(2^k + 1) = k + 2$.

Exercice 4. *Drapeau hollandais (exercice non traité en TD).*

On veut trier un tableau contenant des boules de trois couleurs : bleues `'B'`*, jaunes* `'J'` *et rouges* `'R'`*, afin d'avoir d'abord toutes les boules bleues, puis toutes les boules jaunes et finalement toutes les boules rouges. Le coût de l'algorithme est le nombre de fois que l'algorithme accède à un élément du tableau. Notre objectif est de ne regarder qu'une seule fois la couleur de chaque boule.*

La méthode consiste à conserver à chaque étape la configuration suivante : on met au début toutes les boules bleues déjà rencontrées, ensuite toutes les boules jaunes déjà rencontrées, puis toutes les boules non traitées et, en dernier, toutes les boules rouges déjà rencontrées.

```
tri (boules : tableau de chaînes de caractères, n : entier) : tableau de chaînes de caractères
    B, J, R : entier ; B = 0 ; J = 0 ; R = 0
    tant que B + J + R < n faire
        si boules[B + J] == 'bleu' alors
            boules[B], boules[B + J] = boules[B + J], boules[B]
            B = B + 1
        sinon si boules[B + J] == 'rouge' alors
            boules[B + J], boules[n - R - 1] = boules[n - R - 1], boules[B + J]
            R = R + 1
        sinon
            J = J + 1
    retourner boules
```

Le coût est n — si on admet qu'on réalise les échanges (dans les cas bleu et rouge) sans lire les valeurs échangées. En fait quand on réalise les échanges, il y a une seule valeur qu'il est nécessaire de lire car on ne la connait pas encore, c'est `boules[n - R - 1]`, dans le cas rouge. Si on compte cette lecture, le pire coût est $2n - 1$ (cas où toutes les boules sont rouges). On peut faire mieux en mettant toutes les boules de couleurs connues à gauche :

```
tri2 (boules : tableau de chaînes de caractères, n : entier) : tableau de chaînes de caractères
    B, J, R : entier ; B = 0 ; J = 0 ; R = 0
    tant que B + J + R < n faire
        si boules[B + J + R] == 'bleu' alors
            boules[B + J + R] = 'rouge'
            boules[B + J] = 'jaune'
            boules[B] = 'bleu'
            B = B + 1
        sinon si boules[B + J + R] == 'jaune' alors
            boules[B + J + R] = 'rouge'
            boules[B + J] = 'jaune'
            J = J + 1
        sinon
            R = R + 1
    retourner boules
```