Procédures

1 Cours

Les modules que vous avez utilisés jusqu'à maintenant sont des fonctions. Il existe une autre catégorie de modules : les procédures.

La différence entre procédure est fonction est celle-ci : une fonction renvoie une valeur alors qu'une procédure ne renvoie *rien*.

Alors à quoi sert une procédure si elle ne renvoie rien ? L'intérêt est la lisibilité du programme. Au lieu d'avoir une longue séquence d'instructions dans le programme principal, on peut les diviser en plusieurs parties et traiter chaque partie dans une procédure.

Imaginons que nous ayons besoin d'un programme qui crée un tableau de n valeurs aléatoires, puis qui essaie différents algorithmes pour trier ces valeurs par ordre croissant (ça vous rappelle quelque-chose ?). Le programme ressemblerait à ceci :

```
programme comparaison_tri
   t1[0..99], t2[0..99] : tableaux d'entiers
debut
   //Remplir le tableau t1 de nombres aléatoires
   ...

   //Remplir le tableau t2 de nombres aléatoires
   ...

   //Test de l'algorithme 1
    ...

    //Test de l'algorithme 2
    ...
fin
```

Pour remplir le tableau `t1`, il faudrait écrire :

```
pour i allant de 0 à 99
   t1[i] = randint(0,100) // nombre entier aléatoire entre 0 et 100
fin pour
```

On peut écrire cela dans une procédure :

```
procédure remplir(T : tableau d'entiers)
   i : entier
début
   pour i allant de 0 à 99
      T[i] = randint(0,100) // nombre entier aléatoire entre 0 et 100
   fin pour
fin
```

Nous voyons que `remplir` ne renvoie rien (pas de type retourné dans la signature, et pas d'instruction `retourner`), mais qu'elle peut accepter des paramètres tout comme une fonction. On peut remarquer que l'on pourra utiliser cette procédure deux fois pour remplir `t1` et `t2`. L'intérêt est de n'écrire les instructions qu'une seule fois, ce qui rendra le code plus lisible et plus facile à maintenir (debogguer, améliorer) puisqu'il n'y aura qu'un seule endroit à modifier. On pourrait d'ailleurs rendre la procédure encore plus réutilisable en passant en paramètre la taille du tableau :

```
procédure remplir(T : tableau d'entiers, taille : entier)
   i : entier
début
   pour i allant de 0 à taille-1
      T[i] = randint(0,taille)
   fin pour
fin
```

Pour appeler la procédure, il suffit d'écrire :

```
programme comparaison_tri
   t1[0..99], t2[0..99] : tableaux d'entiers
debut
   remplir(t1,100)
fin
```

On remarque qu'on ne traite pas `remplir(t1,100)` comme une valeur, c'est normal puisqu'elle n'en renvoie pas.

En C++, une procédure se déclare comme une fonction mais avec le type `void` :

```
void  remplir(int* T, int taille)
```

Vous avez dû remarquer le `int*`, c'est la façon de dire que T n'est pas un entier, mais un tableau d'entiers (on appelle ça un pointeur, mais ce n'est pas au programme...).

2 Travail à faire

Le crible d'Ératosthène (astronome, géographe, philosophe et mathématicien grec du III^{e} siècle av. n.è.) est un procédé qui permet de trouver tous les nombres premiers inférieurs à un certain entier naturel N fixé préalablement.

L'algorithme procède par élimination : il s'agit de supprimer d'une table des entiers de 2 à N tous les entiers non premiers. On va donc supprimer de la table tout entier qui est un multiple d'un autre entier de la table. À la fin il ne restera que les entiers qui ne sont multiples d'aucun entier (hormis 1 et eux-mêmes), et qui sont donc des nombres premiers.

Principe

On se donne un entier N. L'objectif de l'algorithme est d'afficher tous les nombres premiers inférieurs à N. On va se servir d'un tableau `T` de taille $N+1$, dont les cases seront numérotées de `T[0]` à `T[N]`, que l'on va remplir de 0 (nombre barré, autrement dit supprimé de la table) et de 1 (nombre non barré). On peut aussi utiliser des booléens (en général `false` et `true`) au lieu des entiers 0 et 1 ; c'est ce que l'on a fait dans le corrigé qui suit. Initialement toutes les cases sont mises à 1 car tous les nombres sont potentiellement premiers.

L'algorithme commence par barrer tous les multiples stricts de 2 ; puis lorsqu'il a fini de barrer les multiples d'un nombre, il passe au nombre suivant non barré, et barre ses multiples stricts, c'est-à-dire les multiples de ce nombres distincts de lui-même. Il suffit à chaque fois de parcourir les multiples stricts dudit nombre jusqu'à $\sqrt{N}+1$. A la fin de l'algorithme, pour tout entier $k \leqslant N$, si k est premier on aura `T[k]=1` et sinon `T[k]=0`. Passer une case du tableau à 0 revient à barrer l'entier correspondant.

Partie A

1. Écrire une procédure en langage naturel `initialisation_du_tableau` qui remplit le tableau `T` avec des 1.
2. Écrire une procédure en langage naturel `barrer_les multiples_de(k)` qui admet en entrée un entier k et barre tous ses multiples stricts, *i.e.* met 0 dans toutes les cases d'indice multiple de k.
3. Écrire une procédure en langage naturel `affichage_des_premiers()` qui crée une liste des nombres premiers de 2 à N à partir du tableau `T` et affiche la liste.

Remarque : les cases `T[0]` et `T[1]` ne servent à rien puisque les nombres premiers sont supérieurs ou égaux à 2, mais on les place par commodité, afin d'éviter un décalage des indices. En effet, de nombreux langages numérotent les tableaux à partir de 0.

Partie B

1. Implémenter sur ordinateur les trois algorithmes précédents. Procéder au débogage.
2. Implémenter le programme permettant l'affichage des la liste des nombres premiers inférieurs à N. Procéder au débogage.