Algorithmique L2 **2019-2020**
Mise en ligne : Vendredi 25 octobre **Date limite : Vendredi 22 novembre**

Devoir Maison d'algorithmique

Ce sujet comporte 4 pages et 4 exercices.

Il est noté sur 23.

Il est à rendre la semaine du 18 au 22 novembre, préférentiellement durant le TD mais tout DM rendu après le vendredi 22 sera pénalisé et possiblement noté 0.

Il possède une partie pratique de programmation (en python ou en C), le fichier est à déposer sur ecampus, nommé selon le schéma `DM_NomDeLEtudiant_NuméroDetudiant.py` ou `DM_NomDeLEtudiant_NuméroDetudiant.c`. Des instructions plus complète seront jointes au fichier sur la page ecampus. En cas de difficulté à y accéder merci de contacter votre chargé de TD.

Attention : en cas de remise tardive du fichier, il sera impossible à l'examinateur de prévenir de son éventuelle illisibilité et il pourra se réserver le droit d'attribuer 0 à la partie programmation.

Introduction

Le théorème fondamental de l'arithmétique (dû à Gauss) dit que tout entier $n \geq 2$ se factorise de façon unique en un produit de nombres premiers. La question de décomposer un entier en son produit de facteurs est une question algorithmique intéressante qui est étudiée depuis longtemps. A titre d'exemple le système de cryptographie le plus utilisé à l'heure actuelle, RSA, repose sur le fait de trouver quels sont les premiers p et q qui divisent le nombre pq, la clé de chiffrement et sa sécurité repose sur le fait que, pour l'instant, il nous est impossible de décomposer ce produit en un temps raisonnable.

Les seules solutions satisfaisantes reposent sur l'utilisation d'algorithmes quantiques mais leur utilisation concrète n'est pas encore possible, faute d'avoir le matériel adapté.

Le but de ce devoir est de voir une méthode basique pour trouver les nombres premiers et donc, entre autres, chercher les premiers qui divisent un entier donné.

Algorithme de décomposition naïf Ici on cherche à décomposer un entier n en son produit de facteurs premiers. Pour cela on va procéder de la façon la plus naïve possible et donc le diviser par tous les nombres premiers inférieurs à n. Si le reste de la division euclidienne est 0, on saura alors que ce nombre premier est un facteur de n. Une des questions préalables est donc de trouver la liste des nombres premiers inférieurs à n, pour cela on va utiliser et implémenter la méthode du *crible d'Ératosthène*.

Crible d'Ératosthène Le but du crible d'Ératosthène est de trouver tous les nombres premiers inférieurs ou égaux à une borne n fixée.

Son principe est de partir d'une liste de tous les entiers de 2 à n (traditionnellement représentée sous forme de grille). On commence par entourer le nombre 2 et barrer tous les nombres qui sont des multiples de 2 jusqu'à n. Puis on entoure 3 qui n'est pas barré et on recommence avec les multiples de 3 et ainsi de suite, à chaque étape on cherche le premier entier suivant celui qu'on vient de voir et qui n'a pas encore été barré, on l'entoure et on barre tout ses multiples. Une fois qu'on a atteint n, tous les nombres entourés sont les nombres premiers.

On trouvera sur ecampus le fichier `Animation Crible dÉratosthène.gif` qui illustre le fonctionnement de cet algorithme.

Pour l'implémentation, au lieu d'utiliser une grille, on utilisera une liste chaînée de noeuds ayant la structure suivante :

```
structure noeudCrible {
    nombre : entier
    coche : booléen
```

```
    suiv : pointeur sur noeudCrible
}

type listeCrible = pointeur sur noeudCrible
```

où `coche` permettra de savoir si un nombre a été coché par le crible d'Ératosthène, c'est à dire qu'il est composé et qu'on l'a rayé. Par défaut cette valeur sera initialisé à `False`.

Par exemple, après avoir appliqué le crible aux nombres de 2 à 7, on devrait avoir la liste suivante :

$$(2, False) -> (3, False) -> (4, True) -> (5, False) -> (6, True) -> (7, False) -> None$$

1. (2 points) **Exercice 1 : création de la liste de nombres**
   Écrire de façon itérative une fonction `creerListe(n : entier) : listeCrible` qui prend en entrée la borne n et renvoie la liste chaînée de tous les entiers de 2 à n avec tous les booléens `coche` initialisés à la valeur `False`. On supposera que l'utilisateur a bien rentré un entier supérieur à 2 et on ne gèrera donc pas de cas particulier. *Indication : on pourra commencer par faire un essai «à la main» avant d'écrire l'algorithme*
2. **Exercice 2 : décomposition d'un entier n en facteurs premiers /4**
   Pour l'instant on va travailler avec une liste précalculée des nombres premiers de 2 à 100. On verra ensuite comment trouver une telle liste. Ici on utilise une liste python basique (ou un tableau C)
   On donne la liste des nombres premiers de 2 à 100 :

   $$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]$$

   L'algorithme de décomposition fonctionnera sur le principe suivant : pour chaque nombre entier $k \leq n$, tester si k divise n. Si c'est le cas, ajouter k à la liste, calculer $n = n/k$ et décomposer le nouveau n selon le même algorithme jusqu'à ce que ça s'arrête.
   (a) (½ point) Décomposer 66 et 23 à la main selon cet algorithme (en ne détaillant que les étapes où on va faire une division).
   (b) (½ point) A quelle condition sur n arrête-t-on l'algorithme ?
   (c) (1 point) Ecrire cet algorithme de façon récursive `decomposition(n : entier) : liste`.
   (d) (½ point) Est-on obligé de tester tous les entiers $k \leq n$? Quel sous-ensemble des entiers peut-on utiliser et pourquoi ? Peut-on trouver une borne plus petite que n pour les k à tester ? (exprimée en fonction de n)
   (e) (1½ points) Réécrire l'algorithme en itératif pour tenir compte de cette nouvelle borne et ce nouvel ensemble de diviseurs possibles

   `decomposition2(n : entier, liste : liste d'entiers) : liste`

   où `liste` est la liste (de type liste comme en python) des entiers k pour lesquels on va tester la divisibilité.
3. **Exercice 3 : Crible d'Ératosthène /8**
   (a) (½ point) Appliquer la méthode du Crible d'Ératosthène aux nombres de 2 à 100 (on représentera cela dans une grille 10x10 avec la première case vide)
   (b) (1 point) Étant donné une liste chaînée `listeNombres` de type `listeCrible` (défini dans l'introduction) donnée par

   `listeNombres=creerListe(n)`,

   (cf exercice 1) donner la fonction itérative

   `cocherMultiples(debut : listeCrible) : listeCrible`

   qui va parcourir la liste `listeNombres` en ne s'intéressant qu'aux nœuds multiples du nœud `debut` et changera la valeur de `coche` de ces nœuds à `True` (on ne se préoccupera pas de la valeur précédente de `coche`). Quant au nœud pointé par `debut`, il changera la valeur en la mettant à la valeur `False`. `debut` sera un pointeur qui se déplacera sur chacun des premiers dont on voudra cocher les multiples.

(c) (½ point) Quelle est la complexité de cette fonction si on compte le nombre de changements de la valeur du pointeur ? Si on compte le nombre de changements de la valeur de `coche` ? (on l'exprimera en fonction de `n` et du nombre premier `p` pointé par la variable `debut`)

(d) (½ point) Ecrire la fonction

`prochainPremier(curseur : pointeur sur noeudCrible) : pointeur sur noeudCrible`

qui renverra un pointeur sur le nœud contenant le prochain (à l'exclusion de celui qu'on est en train de regarder) nombre non coché de la liste et `None` s'il n'en existe pas.

(e) (1 point) Écrire la fonction récursive `sousSuite(liste : listeCrible):listeCrible` qui, étant donné une `listeCrible`, renvoie la même liste ne contenant que les nœuds où `coche` vaut `False`.

(f) (2½ points) Écrire la fonction `crible(n : entier):listeCrible` qui prend en entier la borne du crible `n`, génère la liste de tous les nombres de 2 à `n`, puis applique la méthode du crible à cette liste et renvoie une `listeCrible` ne contenant que les nombres premiers inférieurs ou égaux à `n`. On réutilisera les fonctions `creerListe`, `prochainPremier`, `cocherMultiples` et `sousSuite` définies précédemment.

(g) (1 point) Compléter (sur sa copie) la somme ci-dessous comptant le nombre de fois où on a changé la valeur `coche` d'un nœud (en comptant le fait de mettre à `False` le nœud considéré) :

$$\sum_{\substack{p=2 \\ p \text{ premier}}}^{\dots} \dots \binom{\dots}{\dots}$$

Indice : si on part de 2, combien de nombres va-t-on cocher de 2 à n ?

(h) (½ point) En utilisant le résultat suivant (pour k supposé assez grand), dû à Euler, donner une approximation de la somme de la question précédente.

$$\sum_{\substack{i=2 \\ i \text{ premier}}}^{k} \frac{1}{i} \simeq \ln(\ln(k))$$

(i) (½ point) Quelle est la complexité de l'algorithme du crible d'Ératosthène (on négligera l'effet des fonctions `creerListe`, `prochainPremier`, `sousSuite` qui auraient pu être très différentes si on avait choisi une autre structure de données) ? On comptera en terme de modifications de valeur de `coche`.

Les deux questions suivantes sont facultatives et pourront être omise

(j) Quel est le plus grand entier naturel que l'on peut écrire avec n bits (on ne compte pas de bit de signe) ?

(k) Sur mon ordinateur il m'a fallu environ 0.5 seconde pour calculer tous les premiers se codant en moins de 16 bits. Combien faudrait-il de temps (en années) pour trouver tous les premiers se codant en moins de 1024 bits (la taille moyenne d'une clé RSA il y a quelques années) ?

Astuce : les logarithmes ne différant que d'une constante lorsque l'on change la base, on se permettra ici d'utiliser le logarithme en base 2. On considèrera qu'une année fait 365 jours, qu'un jour contient 86400 secondes.

4. **Exercice 4 : Stockage des résultats dans un arbre binaire de recherche /9**

Ayant calculé la liste des premiers inférieurs à n par le crible d'Ératosthène, on veut maintenant pouvoir utiliser ces données pour les appliquer à notre algorithme de décomposition d'un entier en produit de facteurs premiers. Pour cela on décide de stocker tous les premiers dans un arbre binaire de recherche, afin d'y accéder plus vite.

(a) (½ point) Rappeler la structure d'un nœud d'un arbre binaire

(b) (1 point) Quelle est la particularité des valeurs du sous-arbre gauche d'un arbre binaire de recherche ? Du sous-arbre droit ?

(c) (1 point) Donner l'arbre binaire de recherche obtenu en insérant successivement les nombres 4,2,5,1,3,6. Puis celui obtenu en insérant successivement 1,2,3,4,5,6. Comment qualifie-t-on ce dernier arbre ?

(d) (1 point) Ecrire l'algorithme `ajouterNombre(A:ArbreBinaire, x:entier):ArbreBinaire` qui rajoute un nombre `x` (supposé non déjà contenu dans l'arbre) dans l'arbre de façon à ce que celui-ci reste un arbre binaire de recherche.

(e) (½ point) Donner l'algorithme `estDansLarbre(A:ArbreBinaire,x:entier):booléen` qui renvoie `True` si `x` est dans l'arbre, et `False` sinon.

(f) (2 points) En supposant que l'arbre est plus ou moins régulier (c'est-à-dire qu'il est de hauteur p s'il contient entre 2^p et $2^{p+1}-1$ nombres), donner la complexité au pire de la fonction `estDansLarbre` en fonction de n, son nombre de nœuds, en choisissant comme mesure de la complexité, le nombre de nœuds visités. Même question dans le cas où on a ajouté les nombres dans l'ordre croissant. Pourquoi est-il important de ne pas insérer les valeurs dans cet ordre ?

Indice : pour n donné, comment exprimer p en fonction de n tel que $2^p \leq n \leq 2^{p+1}-1$?

(g) (1½ points) En déduire l'algorithme `creerArbre(listePremiers:liste):ArbreBinaire` qui, à une liste de nombres premiers, associe un arbre binaire de recherche les contenant. On supposera qu'on a maintenant une liste « normale » (au sens de Python et non une liste chaînée) et que les nombres premiers ne sont pas forcément dans l'ordre.

(h) (1 point) Quel ordre doit-on utiliser pour parcourir l'arbre binaire de recherche si l'on veut énumérer tous les nombres dans l'ordre croissant ? Justifier.

(i) (½ point) Écrire l'algorithme récursif

`parcourirArbreOrdreCroissant(A:ArbreBinaire)`

qui affiche les nombres contenus dans l'arbre dans l'ordre croissant.

Les deux questions suivantes sont facultatives et pourront être omises

(j) On suppose qu'on a stocké la liste des entiers premiers dans une `listeCrible` ordonnée de longueur n. Quelle est, au pire, la complexité de la recherche d'une valeur dans cette liste comptée en nombre de nœuds visités ?

(k) Quel est le meilleur format de stockage d'une liste de nombres premiers entre la `listeCrible` et l'arbre binaire du point de vue de la recherche d'une valeur ? Supposons qu'il nous faille une seconde pour le tester dans un arbre de $127 = 2^7 - 1$ nœuds et dans une liste de même taille, combien de temps faut-il pour faire la même recherche dans un arbre et une liste de 255 nœuds chacun ?