TP1 : Décomposition en facteur premier

Télécom Physique Strasbourg — November 9, 2021

Consignes: Traiter le sujet ci-dessous, puis finaliser les exercices des CI précédents .

Théorème 1 (Théorème fondamental de l'arithmétique) *Tout entier positif n peut être écrit sous forme d'un produit de nombres premiers, et cette décomposition est unique.*

Exemple:

$$234 = 2 \times 3 \times 3 \times 13 = 2 \times 3^2 \times 13$$
$$63 = 3^2 \times 7$$
$$61 = 61 \quad \textit{(car } 61 \textit{ est premier, comme } 2, 3, 7 \textit{ et } 13)$$

La décomposition rapide en facteurs premiers de grands nombres semi-premiers est un défi des mathématiques et de l'informatique. Les nombres semi-premiers sont le résultat du produit de 2 nombres premiers entre eux et sont à la base de méthodes de cryptage très utilisées sur les réseaux informatiques. Il aura fallu en 2005, 5 mois à un système composé de 80 CPU pour factoriser et donc casser la clé de cryptage RSA-640 codée sur 640 bits[1]. Les 2 facteurs premiers de la clé étaient :
163473364580925384844313388386509085984178367003309231218111085238933310010450815121211816751157 9 et
190087128166482211312685157393541397547189678996851549366663853908802710380210449895719126146557 1.

Un algorithme simple, mais peu rapide, pour la décomposition en facteurs premiers consiste à tester tous les nombres premiers $p_i \leq n$ pour déterminer lesquels sont facteurs/diviseurs et à quel ordre/puissance. Ainsi, on teste si p_1 est diviseur de n. Si oui, on recommence avec la valeur de n/p_1. On teste alors si p_1 est encore diviseur (de multiplicité 2) de cette valeur, si oui on recommence avec la nouvelle valeur de la division. Si p_1 n'est plus diviseur, on passe au test avec p_2 ....

Exemple:
Pour $n = 20$, l'ensemble p est $\{2, 3, 5, 7, 11, 13, 17, 19\}$
$20/2 = 10$ reste nul, donc 2 est facteur premier
$10/2 = 5$ donc 2 est facteur de multiplicité 2
$5/2 = 2.5$ donc 2 n'est plus facteur
$5/3 = 1.66$ donc 3 n'est pas facteur
$5/5 = 1$ donc 5 est facteur .... et finalement $\boxed{20 = 2 \times 2 \times 5}$

Cet algorithme suppose d'établir la liste des nombres p_i, ce qui implique au préalable de déterminer si un nombre est premier.

$\boxed{\text{1.Test de primalité}} \rightarrow \boxed{\text{2.Liste des nombre premiers} \leq n} \rightarrow \boxed{\text{3.Décomposition en facteurs premiers}}$

Une base de l'algorithmique consiste à décomposer un problème complexe en sous-problèmes plus simples, puis les combiner pour obtenir l'algorithme complet.

[1]La taille recommandée actuelle est d'au minimum 2048 bits pour assurer une bonne sécurité.

1 Test de primalité

Théorème 2 (Nombre premier) *Un nombre entier strictement positif n est premier $\Leftrightarrow$ il a seulement deux diviseurs, 1 et lui-même.*
Corollaire : Le nombre 1 n'est pas premier.

Soit une fonction de prototype **int** is_prime(**int**) qui retourne la valeur 1 (vrai) ou 0 (faux) si le nombre passé en argument est premier[2]. Cette fonction doit aussi gérer le cas des nombres nuls et négatifs.

◦ Ecrire le corps de la fonction is_prime() en complétant le code source `tp1.c` disponible en ligne. Tester votre fonction.
Naturellement, la saisie du nombre à tester et l'affichage du résultat sont effectuées exclusivement dans le main() et non dans votre fonction is_prime().

◦ Améliorer l'algorithme, si cela n'est déjà fait, en tenant compte du fait suivant: si un diviseur est trouvé, le nombre n'est pas premier et donc la fonction peut immédiatement retourner sans poursuivre les tests.

2 Liste de nombres premiers $\leq n$

◦ Ecrire une fonction prime_array() capable de remplir un tableau passé en premier paramètre avec tous les nombres premiers inférieurs à l'entier n. Cette fonction fera avantageusement appel à la fonction précédente.
On alloue dynamiquement avec malloc() le tableau dans le main() en fonction de n avant de le passer à la fonction. L'allocation du tableau n'est effectuée qu'une seule fois. On choisit donc sa taille initiale en fonction de n pour le pire cas. Un tableau trop grand, dont les derniers éléments ne sont pas utilisés, n'est pas un problème.

Note: Retourner par le moyen de votre choix le nombre d'éléments premiers effectifs dans le tableau est alors probablement pertinent.

Info: Dans un cas autre que celui de ce TP, il est possible qu'une taille maximale ne puisse être inférée pour le tableau. La solution consiste alors à allouer un tableau d'une vingtaine d'éléments par exemple, et utiliser realloc () pour augmenter la taille d'un bloc de 20 éléments supplémentaires à chaque fois que le tableau est plein. Afin de ne pas pénaliser le temps d'exécution avec trop de realloc () coûteux, on alloue des blocs de plusieurs éléments plutôt qu'une allocation élément par élément.

On n'oubliera pas de définir le prototype des fonctions en début du code de source pour pouvoir mettre leur définition dans n'importe quel ordre dans le fichier. Tester et valider en affichant le tableau avec la fonction proposée printf_array ().

3 Décomposition en facteur premier

◦ Modifier le main() pour afficher les facteurs premiers de la décomposition d'un nombre via l'algorithme simple proposé. Valider en testant les 4 exemples en début de sujet.

Exemple :
```
Number? 63
        Prime factorization : 3 3 7
```

[2] `http://primes.utm.edu/lists/small/1000.txt` pour une liste des 1000 premiers nombres premiers pour vos tests.

◦ Mettre votre code de décomposition précédent sous forme d'une fonction possédant au moins deux paramètres -le nombre à décomposer n et -un tableau déjà alloué de taille suffisante pour contenir les facteurs premiers résultats.

Warning: Votre fonction de décomposition aura a priori besoin d'allouer dynamiquement un tableau pour les nombres premiers inférieurs à n. Pour éviter une fuite mémoire, conduisant à terme à la saturation de la mémoire, on n'oubliera pas de libérer avec un free() ce tableau de nombres premiers avant le retour de la fonction.

4 Chercher une issue

Dans le film d'horreur-science fiction Cube (1997), 6 personnes se réveillent inexplicablement dans un cube. Ce cube est connecté par 6 trappes au centre de chaque face à 6 autres pièces cubiques. En explorant ce labyrinthe, ils découvrent qu'il y a 17 576 pièces, qui forment ensemble un cube plus grand, de 26 pièces cubiques de côté. Ils découvrent que chaque pièce cubique est numérotée avec trois nombres à trois chiffres; que les cubes, dont au moins un des nombres est premier ou est une puissance d'un nombre premier (par exemple: 064=2^6 ou 343= 7^3), contiennent des pièges mortels [3].

Ainsi la salle 006 456 432 est sûre, alors que les salles 321 431 600 et 169 546 998 sont piégées. En effet, 431 est premier et 169=13^2.

◦ On souhaite savoir si une pièce est sûre. Modifier votre fonction de décomposition précédente pour ajouter en paramètre deux variables booléennes : une qui retourne "vrai" si le nombre n est premier et l'autre qui retourne "vrai" si le nombre n est une puissance d'un nombre premier.

◦Tester et afficher si le nombre correspond à une salle sûre ou piégée.

Note: L'objectif est ici de vous obliger à écrire une fonction qui utilise ses paramètres (avec le bon mode de passage) pour retourner plus d'un résultat.

[3] Source et discussion mathématique autour du film: `https://populscience.blogspot.com/2016/06/mistake-in-cube.html`