



Rapport de projet

Analyse d'algorithmes et
validation de programmes



Professeur: M. Devan SOHIER

Table des matières

Introduction	3
Fonctions récursives	4
Fonctions itératives	9
Fonctions itératives avec table de recherche	11
Preuves	13
Bilan	16

Introduction

L'importance des nombres premiers ne peut être surestimée en matière de sécurité informatique, notamment dans le domaine de la cryptographie. En effet, ces nombres sont essentiels pour générer des clés cryptographiques qui garantissent la confidentialité et l'intégrité des communications et des données sensibles. La sécurité de nombreux systèmes cryptographiques actuels repose sur l'hypothèse que la factorisation de nombres entiers est un problème difficile à résoudre.

Dans ce rapport, nous aborderons un aspect clé de la théorie des nombres premiers : la recherche du nombre d'entiers positifs premiers inférieurs à un entier positif donné.

Comment implémenter ce problème de la meilleure manière ?

Nous examinerons les différentes approches pour résoudre ce problème, en commençant par étudier les versions récursives et itératives de l'algorithme. Nous analyserons également l'impact de l'ajout d'une table de recherche sur la version itérative. Enfin, nous présenterons une preuve détaillée de l'algorithme.

Fonctions récursives

J'ai tout d'abord implémenté de problème sous forme récursive. L'idée globale pour déterminer le nombre d'entiers premiers positifs inférieur à x était de tester la primalité de chaque nombre inférieur à x .

Voici la fonction qui teste si un nombre est premier:

```
// renvoie 1 si le nombre est premier, 0 sinon
// compteur2 commence à 2 au premier appel de est_premier
int est_premier(int n, int compteur2)
{
    // cas d'arrêt 1:
    // si on n'a pas trouvé de diviseur parmi tous les entiers positifs entre 1 exclu et n exclu
    // le nombre est premier
    if (compteur2 == n)
    {
        return 1;
    }
    // sinon si le reste de la division euclidienne de n par compteur2 n'est pas 0
    // alors compteur2 n'est pas un diviseur de n
    // On continue de vérifier les nombres suivants
    else if (n % compteur2 != 0)
    {
        compteur2++;
        return est_premier(n, compteur2);
    }

    // Cas d'arrêt 2: on a trouvé un diviseur de n autre que 1 et n
    // Le nombre n'est pas premier
    else
    {
        return 0;
    }
}
```

La complexité de ***est_premier*** dépend de la valeur de ***n***.

Dans le pire des cas, où ***n*** est un nombre premier, la fonction va effectuer $n-2$ appels récursifs (1 pour chaque ***compteur2*** entre 2 et $n-1$ inclus).

Chacun de ces appels effectue un modulo et une opération d'addition. Ces opérations ne dépendent pas de l'entrée donc ils ont une complexité en temps constant $O(1)$.

La complexité *au pire* de *est_premier* est donc $O((n-2)*1) = O(n-2) = \mathbf{O(n)}$

Dans le **meilleur** des cas, **n** n'est pas premier et possède un diviseur commun avec 2. Dans ce cas, n % compteur2 sera égal à 0 et on return 0. C'est une opération indépendante de n et donc en temps constant.

Il y a un deuxième "**meilleur**" cas: si n vaut 2. Dans ce cas, n=compteur2 dès la première itération et on return 1. C'est aussi une opération indépendante de l'entrée.

Donc, la complexité *au mieux* de *est_premier* est $O(1)$.

La complexité **en moyenne** de cette fonction dépend de la répartition des nombres premiers. En effet, moins il y a de nombres premiers (nombres qui n'ont de diviseurs qu'eux-mêmes et 1), plus il y a de chance que n rencontre un diviseur. Et donc plus il y a de chances que notre fasse moins d'appels récursifs et fasse donc un return 0 plus tôt.

Je pensais avant d'effectuer ce projet que la répartition des nombres premiers était aléatoire. Au premier abord, je pensais donc, qu'en moyenne, il y aurait une chance sur 2 pour qu'un n aléatoire supérieur à 2 soit premier.

La complexité en moyenne de *est_premier* aurait alors été $O(n/2)$.

Cependant, après quelques recherches sur la répartition des nombres premiers, je suis tombé sur ce théorème:

https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_de_la_rar%C3%A9faction_des_nombres_premiers

Le **théorème de la raréfaction des nombres premiers** est un résultat démontré par Adrien-Marie Legendre en 1808¹. C'est, aujourd'hui, un corollaire du théorème des nombres premiers², conjecturé par Gauss et Legendre dans les années 1790 et démontré un siècle plus tard.

Le résultat énonce que la **densité asymptotique** de l'ensemble des **nombres premiers** est nulle, c'est-à-dire que le **nombre de nombres premiers inférieurs à n**, $\pi(n)$, est **négligeable** devant n lorsque n tend vers l'infini, autrement dit que

$$\lim_{n \rightarrow +\infty} \frac{\pi(n)}{n} = 0.$$

La preuve initiale utilise les **techniques de crible** fondées sur le **principe d'inclusion-exclusion**². L'interprétation est qu'à mesure que n croît, la **proportion** de nombres premiers parmi les **entiers naturels** inférieurs à n **tend vers** zéro, d'où le terme de « **raréfaction des nombres premiers** ».

Autrement dit, plus x est grand, plus il y a de chances de rencontrer un nombre non premier.

On peut établir un lien avec notre fonction, puisque plus il y a de chances de rencontrer un nombre non premier (nombre qui possède un ou plusieurs autres diviseurs que lui-même et 1), plus il y a de chances que notre n rencontre un diviseur et que la boucle s'arrête tôt.

Donc la complexité en moyenne, pour des nombres très grands, tend vers moins de $O(n/2)$.

```

// Teste si chaque positif inférieur à x est premier.
// res représente le nombre des positifs premiers inférieurs à x
// compteur: nombre en cours de test (pour déterminer s'il est premier) dans cet appel
// compteur vaut 2 lors du 1er appel de cette fonction
int nb_entiers_positifs_premiers_inferieurs_x(int x, int compteur, int res)
{
    if ((x == 0) || (x == 1))
    {
        return 0;
    }

    else if (compteur == x) // cas d'arrêt: on a fini de vérifier tous les nombres inférieurs à x
    {
        return res;
    }

    else
    {
        if (est_premier(compteur, 2))
        {
            res++;
        }

        compteur++;

        return nb_entiers_positifs_premiers_inferieurs_x(x, compteur, res);
    }
}

```

La fonction `nb_entiers_positifs_premiers_inferieurs_x` effectue un appel récursif pour chaque `compteur` de 2 à `x` inclus. Dans chacun de ces appels, elle effectue un appel à la fonction `est_premier`.

Elle effectue donc $n-1$ appels récursifs, dont chacun effectue un appel à `est_premier`.

Le paramètre passé à `est_premier` incrémenté à chaque appel.

En effet, `compteur` commence à 2. Puis incrémenté à 3 etc jusqu'à $n-1$.

La complexité de l'appel à `est_premier` vaut donc d'abord $O(2)$ puis $O(3)$ puis ... puis $n-1$.

La complexité *au pire* de `nb_entiers_positifs_premiers_inferieurs_x` est donc égale à la somme des complexités de `est_premier`.

La somme des entiers de 2 à n-1 peut être calculée en utilisant la formule de la somme des premiers entiers.

Cette formule nous dit que la somme des k premiers entiers est égale à $(k*(k+1))/2$.

Dans ce cas, la somme des entiers de 2 à n-1 est égale à $(n-2)*((n-2)+1)/2$, qui peut être simplifié en $(n-2)*(n-1)/2$.

Ainsi, la somme des entiers de 2 à n-1 est égale à $(n^2 - 3n + 2)/2$.

$$O((n^2 - 3n + 2)/2) = O(n^2).$$

Dans le meilleur des cas, $x == 0$ ou $x == 1$ et on effectue seulement l'opération return 0.
Donc sa complexité ***au mieux*** est $O(1)$.

Dans les autres cas, nombre d'appels ne change pas en fonction de l'entrée puisque pour n'importe quel x en entrée, on vérifie la primalité de chaque entiers positifs entre 2 et x-1.

Donc la complexité en ***moyenne*** (moyenne des complexités de tous les cas) sera la même que la complexité au pire, soit $O(n^2)$.

Fonctions itératives

```
// renvoie 1 si le nombre est premier, 0 sinon
// compteur2 commence à 2 au premier appel de est_premier
int est_premier(int n, int compteur2)
{
    for (; compteur2 < n; compteur2++)
    {
        // si le reste de la division euclidienne de n par compteur2 est 0
        if (n % compteur2 == 0)
        {
            return 0; // n n'est pas premier car on a trouvé un diviseur autre que 1 et n
        }
    }

    return 1;
}
```

Dans le pire des cas, le nombre est premier. La boucle qui effectuera donc $n-2$ itérations (une pour chaque n de 2 à $n-1$). Dans chaque itération, on effectue une comparaison et l'opération modulo. Ce sont des opérations indépendantes de n . Donc chaque itération a une complexité constante.

Donc la complexité **au pire** de cette fonction est $O((n-2)*1)=O(n)$.

Dans le meilleur des cas, n vaut 2. Dans ce cas, la boucle n'est pas effectuée et la fonction retourne directement 1. C'est une opération en temps constant donc la complexité au mieux est $O(1)$. Pour la complexité en moyenne, on peut utiliser le même raisonnement

que pour la version itérative. La boucle finit plus tôt quand elle a plus de chances de croiser des nombres non premiers.

```
int nb_entiers_positifs_premiers_inferieurs_x(int x, int compteur, int res)
{
    if ((x == 0) || (x == 1))
    {
        return 0;
    }

    for (; compteur < x; compteur++)
    {
        if (est_premier(compteur, 2))
        {
            res++;
        }
    }

    return res;
}
```

Cette fonction effectue $x-2$ itérations (une pour chaque *compteur* de 2 à $x-1$).

A chaque itération, elle effectue un appel à *est_premier* qui prend en paramètre *compteur*.

Compteur incrémenté à chaque tour de boucle.

En utilisant le même raisonnement que pour sa version récursive, on trouve que sa complexité *en moyenne* et *au pire* sont $O(x^2)$ et que sa complexité *au mieux* est $O(1)$.

Cependant, bien que la complexité ne change pas, j'ai pu observé des améliorations notables en temps d'exécution.

Fonctions itératives avec table de recherche

```
int nb_entiers_positifs_premiers_inferieurs_x(int x, int compteur, int res, int *table_recherche)
{
    if ((x == 0) || (x == 1))
    {
        return 0;
    }
    else if (mode == 0) // aucun élément dans la table de recherche
    {
        FILE *fichier;
        fichier = fopen("table_recherche", "a+");

        for (; compteur < x; compteur++)
        {
            if (est_premier(compteur, 2))
            {
                fprintf(fichier, "1 ");
                res++;
            }
            else
            {
                fprintf(fichier, "0 ");
            }
        }
        fclose(fichier);
    }
    else if (mode == 2) // tous les éléments sont dans la table de recherche
    {
        for (; compteur < x; compteur++)
        {
            if (table_recherche[compteur] == 1)
            {
                res++;
            }
        }
    }
    else
    {
        FILE *fichier;
        fichier = fopen("table_recherche", "a+");

        for (; compteur < nb_elements_dans_table_recherche; compteur++)
        {
            if (table_recherche[compteur] == 1)
            {
                res++;
            }
        }

        for (; nb_elements_dans_table_recherche < x; compteur++)
        {
            if (est_premier(compteur, 2))
            {
                fprintf(fichier, "1 ");
                res++;
            }
            else
            {
                fprintf(fichier, "0 ");
            }
        }
        fclose(fichier);
    }
    return res;
}
```

NB: est_premier ne change pas rapport à la version itérative sans table de recherche.

Dans le **pire** des cas (quand la table de recherche est vide) cette fonction effectue les mêmes opérations que sa version itérative sans table. Sauf qu'elle ajoute à chaque itération une opération fprintf, fopen et FILE*fichier. Ce sont des opérations constantes qui ne dépendent pas de l'entrée.

Donc la complexité **au pire** reste la même que son autre version itérative. Autrement dit $O(x^2)$. Mais le temps d'exécution empire par rapport à la version itérative.

Dans le **meilleur** de cas, tous les éléments sont dans la table de recherche.

Dans ce cas, elle effectue $x-2$ itérations (1 pour chaque compteur de 2 à $x-1$ inclus).

Chaque itération effectue des opérations en temps constant: une addition, un accès mémoire et une comparaison. La complexité **au mieux** est donc $O(x-2)=O(x)$.

Il n'est pas judicieux ici de calculer la complexité **en moyenne** car on peut manuellement influencer sur le remplissage de la table de recherche, en effaçant ses éléments à la main ou la remplaçant avant l'exécution.

La diminution de complexité quand la table est assez remplie se fait très fortement ressentir au niveau du temps d'exécution puisqu'il est quasiment instantané tant que les tests de primalité de tous les nombres inférieurs sont déjà dans la table:

```
rayanehammoumi@macbook-air-de-rayane v2AAVP % ./version_recursive 115000
Nombre d'entiers positifs premiers inférieurs à 115000: 10871
Temps d'exécution: 4.608188
rayanehammoumi@macbook-air-de-rayane v2AAVP % ./version_iterative 115000
Nombre d'entiers positifs premiers inférieurs à 115000: 10871
Temps d'exécution: 0.779586
rayanehammoumi@macbook-air-de-rayane v2AAVP % ./version_iterative_avec_table_recherche 115000
Nombre d'entiers positifs premiers inférieurs à 115000: 10871
Temps d'exécution: 0.000263
```

Le temps d'exécution est exprimé ici en secondes. Etant donné qu'il y a une infinité de nombre premiers et qu'on peut, en théorie, choisir une infinité de x différents, voire un x infiniment grand, je n'ai pas pu calculer le temps d'exécution en moyenne et au pire.

NB: la version_recursive fait une erreur de segmentation pour tout nombre supérieur à environ 115 000. C'est aussi un avantage de la version itérative.

Preuves

Spécification (*nb_entiers_positifs_premiers_inferieurs_x*)

- Préconditions: x est un entier positif, compteur=2 à la première itération
- Postcondition: le programme renvoie le nombre d'entiers positifs premiers inférieurs à n.
- Invariant: Dans *nb_entiers_positifs_premiers_inferieurs_x*, à chaque itération (ou appel récursif selon la version choisie), *res* est égal au nombre d'entiers positifs premiers inférieurs à la valeur de *compteur*.

Compteur incrémenté à chaque itération/appel.

Montrons par récurrence que l'invariant de boucle est vrai pour toute itération.

Initialisation : Avant la première itération, compteur est initialisé à 2.

Les entiers positifs inférieurs à 2 sont 0 et 1. Ils ne sont pas premiers. Or, avant la première itération, res=0 donc l'invariant de boucle est vérifié.

Héritage: Supposons que l'invariant de boucle est vrai pour une certaine itération de la boucle.

Lors de l'itération suivante, la fonction *est_premier* est appelée avec compteur en argument. Selon le code de la fonction *est_premier*, elle renvoie 1 si compteur est premier, sinon elle renvoie 0. Si *est_premier*(compteur, 2) renvoie 1, cela signifie que compteur est premier, donc res est incrémenté de 1. Sinon, compteur n'est pas premier et res reste inchangé. Dans les deux cas, l'invariant de boucle est préservé.

Donc l'invariant de boucle est vrai pour toute itération.

La boucle se termine lorsque compteur atteint n. L'invariant de boucle implique que res contient le nombre de nombres premiers inférieurs à n.

Ainsi, on a montré que *nb_entiers_positifs_premiers_inferieurs_x* est correcte.

Spécification (*est_premier*)

Préconditions: n est un entier positif, compteur2 est un entier positif supérieur ou égal à 2

Postcondition: retourne 1 si n est premier, 0 sinon

Invariants: $n \% (\text{compteur2} - 1) \neq 0$

Preuve par récurrence que les invariants sont vrais pour toute itération:

Initialisation

Avant la boucle, n et compteur2 sont des entiers positifs (Le programme ne se lance pas si on ne spécifie pas un argument positif et compteur2 commence à 2. Donc les préconditions sont vérifiées. (NB: *est_premier* n'est jamais appelé avec n = 0 ou 1)).

De plus, aucun nombre compris entre 2 et n - 1 ne divise n, car la boucle n'a pas encore été exécutée. L'invariant est donc vérifié.

Hérité: Supposons que l'invariant soit vrai pour une itération/appel. Montrons qu'il est vrai à la prochaine itération/appel.

Lors de cette prochaine itération/appel, on incrémente compteur2 de 1. Donc compteur2 devient un nombre compris entre 3 et n-1 inclus.

Si $n \% (\text{compteur2}) == 0$, alors la boucle s'arrête. Compteur2 n'incrémentera pas et on a supposé que $n \% (\text{compteur2} - 1) \neq 0$ donc l'invariant reste vrai.

Si $n \% \text{compteur2} \neq 0$, alors à la prochaine itération compteur2 va incrémenter de 1. L'invariant $n \% (\text{compteur2} - 1) \neq 0$ reste donc évidemment conservé puisque c'est la même expression que celle qu'on vient de supposer.

Conclusion: Pour tout compteur2 de 3 à n-1, $n \% (\text{compteur2} - 1) \neq 0$.

Donc si la boucle s'arrête lorsque compteur2 est égal à n, on sait que pour tout compteur2 de 3 à n-1, $n \% (\text{compteur2} - 1) \neq 0$ sinon la boucle aurait déjà été interrompue. Autrement dit, n n'est divisible par aucun nombre entre 2 et n-1.

Donc, si la boucle s'arrête lorsque compteur2 est égal à n, alors n est premier et la fonction retourne 1.

Ainsi, la fonction `est_premier` est correcte, c'est-à-dire qu'elle retourne 1 si n est premier et 0 sinon, pour tout entier positif n.

Bilan

Pour résumer, nous avons pu déterminer plusieurs programmes corrects pour l'implémentation de notre problème.

En fin de compte, nous avons vu que l'efficacité des algorithmes dépend de la distribution des nombres premiers, mais que l'utilisation d'une version itérative présente déjà des améliorations par rapport à une version récursive. Aussi, l'ajout d'une table de recherche présente une amélioration considérable par rapport à une version itérative sans table. Cependant, elle nécessite d'abord d'être remplie, ce qui prend du temps.

À partir de l'étude de la recherche des nombres premiers et de leurs implications en sécurité informatique, il est possible d'ouvrir le débat sur les limites de la cryptographie actuelle.

Au vu, des recherches sur les ordinateurs quantiques et du fait que la puissance de calcul informatique croît de manière générale d'année en année, certaines méthodes de cryptage reposant sur la factorisation de nombres premiers pourraient un jour être plus vulnérables à des attaques.