Autour des nombres premiers

1 Introduction

2 Manipulation des variables globales

1. Il suffit de mettre `nb_primes` à zéro, pour signifier qu'aucune case du tableau `primes` ne contient d'entiers identifiés comme premiers (il est inutile de toucher au tableau `primes`) :

```c
void init_primes(void) {
    nb_primes = 0;
}
```

2. Attention, dans cette fonction, à l'ordre des deux opérations pour que l'entier ajouté au tableau se retrouve dans la bonne case : s'il y a déjà `nb_primes` entiers enregistrés comme premiers (dans les cases d'index 0 à `nb_primes`−1 du tableau `primes`), le suivant doit aller dans la case d'index `nb_primes`. Puis ont incrémente `nb_primes`. Cela donne :

```c
void add_prime(int p) {
    primes[nb_primes] = p;
    nb_primes++;
}
```

3 Premiers remplissages

3. Rien de bien compliqué ici. On initialise le tableau, puis on considère tous les entiers de 0 à $2^{16}-1$ (inclus). Tous ceux identifiés comme premiers sont ajoutés au tableau.

```c
void gen_basic(void) {
    init_primes();
    for (int i=0; i<65536; ++i) {
        if (test_if_prime_basic(i)) {
            add_prime(i);
        }
    }
}
```

4. `check_primes` permet de vérifier le bon fonctionnement des fonctions précédentes. À noter qu'il proteste à propos du contenu du tableau `is_prime`, ce qui est normal puisque pour l'instant nous n'avons pas rempli ce tableau.

5. On peut améliorer les choses en ne testant que la divisibilité par les premiers déjà découverts. On peut s'arrêter dès que les entiers p candidats à être des diviseurs $\sqrt{k}$ (condition que l'on notera `k/p < p` par exemple, pour éviter les débordements). Mais cela ne peut pas être la seule condition, il faut impérativement éviter d'accéder aux cases du tableau `primes` aux indices supérieurs ou égaux à `nb_primes` !

On traitera par ailleurs à part le cas de $k=0$ et $k=1$, qui ne sont pas premiers même s'ils n'ont pas de diviseurs propres premiers.

Cela donne par exemple :

```c
bool test_if_prime_better(int k) {
    // On traite à part k=0 et k=1
    if (k<2) {
        return false;
    }
    for (int i=0; i<nb_primes; ++i) {
        int p = primes[i];
        if (k/p < p) {
            // p*p > k, inutile de continuer, k est premier
            return true;
        }
        if (k%p == 0) {
            // p divise k, k est composite
            return false;
        }
    }
    // Plus de candidats diviseurs, k est premier
    return true;
}
```

6. Rien de bien sorcier ici, la fonction est similaire à celle de la question 3.

```c
void gen_better(void) {
    init_primes();
    for (int i=0; i<65536; ++i) {
        if (test_if_prime_better(i)) {
            add_prime(i);
        }
    }
}
```

7. C'est une fonction qui nécessite beaucoup de soin pour traiter correctement tous les cas particuliers. On peut tout d'abord traiter à part le cas $k = 1$.

Ensuite, on considère tous les premiers p que l'on a préalablement identifié. Tant que l'entier dont on cherche la décomposition est divisible par p, on affiche p et on divise l'entier par p (on ajoute une virgule si après la division on n'a pas atteint 1).

Deux possibilités vont nous faire sortir de la boucle : si à force de trouver des diviseurs on a atteint 1, la décomposition est achevée. Mais il peut également arriver que l'**on ne dispose plus de nombres premiers à tester**. Seulement, si aucun premier inférieur à 2^{16} ne divise k, puisque k est plus petit que $2^{31} < (2^{16})^2$, alors k est nécessairement premier : c'est donc le dernier terme de la décomposition recherchée.

```c
void print_decomp(int k) {
    printf("%d : ", k);

    // Cas particulier : k=1
    if (k==1) {
        printf("1\n");
        return;
    }

    // Cas général
    for (int i=0; i < nb_primes; ++i) {
        int p = primes[i];
        while (k%p == 0) {
            printf("%d", p);
            k = k/p;
            if (k==1) {
                // La décomposition est terminée
                printf("\n");
                return;
            }
            printf(", ");
        }
    }

    // Dernier terme de la décomposition
    printf("%d\n", k);
}
```

8. Cela donne les résultats suivants :

```
37 : 37
54 : 2, 3, 3, 3
1 : 1
999789210 : 2, 3, 3, 3, 5, 7, 17, 29, 29, 37
999999863 : 25303, 39521
999999873 : 3, 3, 43, 2583979
999999883 : 999999883
```

4 Crible d'Erathostene

9. En suivant simplement à la lettre les explications fournies, on peut implémenter `gen_eratosthene` de la sorte :

```c
void gen_eratosthene(void) {
    init_primes();

    is_prime[0] = false;
    is_prime[1] = false;
    for (int i=2; i<65536; ++i) {
        is_prime[i] = true;
    }

    for (int k = 0; k<65536; ++k) {
        if (is_prime[k]) {
            add_prime(k);
            // On élimite 2k, 3k, 4k...
            for (int m=2*k; m<65536; m+=k) {
                is_prime[m] = false;
            }
        }
    }
}
```

On remarquera l'écriture de la boucle sur `m` : pour égrenner les termes d'une suite arithmétique telle que celle attendue, il est bien plus simple de partir de $2k$ et d'ajouter k à chaque itération !

10. Les multiples de la forme $k^2 + k$, $k^2 + 3k$, $k^2 + 5k$, etc. sont pairs, et donc ont déjà été éliminés comme multiples de 2. Les multiples de la forme $p \times k < k^2$ ont également déjà été éliminés comme multiples de p (s'il est premier) ou du plus petit de ses diviseurs premiers.

11. On traite séparément chacun des trois cas (dans le dernier cas, $k^2 > 65536$, donc il n'y a plus de multiples à éliminer!) :

```
void gen_erathostene_2(void) {
    init_primes();
    is_prime[0] = false;
    is_prime[1] = false;

    // Cas de 2, élimination des entiers pairs > 2
    is_prime[2] = true;
    add_prime(2);
    for (int i=3; i<65536; ++i) {
        is_prime[i] = (i%2 != 0);
    }

    // Entiers impairs de 3 à 256
    for (int k = 3; k<257; k+=2) {
        if (is_prime[k]) {
            add_prime(k);
            for (int m=k*k; m<65536; m+=2*k) {
                is_prime[m] = false;
            }
        }
    }

    // Entiers impairs de 257 à n-1
    for (int k = 257; k<65536; k+=2) {
        if (is_prime[k]) {
            add_prime(k);
        }
    }
}
```

12. On obtient par exemple les temps suivants, qui montrent les gains obtenus avec les différentes améliorations :

```
Basic : 0.636400 ms
Better : 0.593200 ms
Eratosthene : 0.156400 ms
Eratosthene 2 : 0.112500 ms
```

5 Tests de primalité

13. Seul 999999883 est premier, comme nous l'avons vu avec les décompositions un peu plus tôt.

14. On obtient les temps suivants :

```
Pour 999789210    basic : 0.000002 ms   better : 0.000003 ms
Pour 999999863    basic : 0.015860 ms   better : 0.003676 ms
Pour 999999873    basic : 0.000002 ms   better : 0.000003 ms
Pour 999999883    basic : 0.014800 ms   better : 0.001406 ms
```

On obtient la réponse très vite dans les premier et troisième cas car on a un diviseur très petit (2 et 3, respectivement). La différence de temps d'exécution n'est pas significative. Pour les deux autres entiers, il faut tester de nombreux candidats diviseurs. La fonction `test_if_prime_better` effectue moins de calculs de reste de division entière (car elle ne teste que des candidats premiers), donc permet de gagner du temps.

15. On peut par exemple écrire :

```
int pow_mod(int a, int b, int k) {
    if (b == 0) { return 1; }
    long long int c = pow(a, b/2, k);
    if (b%2 == 0) {
        return (c*c) % k;
    } else {
        return (a*c*c) % k;
    }
}
```

16. On peut simplement écrire :

```
bool test_if_prime_fermat(int k, int a) {
    return pow_mod(a, k-1, k) == 1;
}
```

17. On peut générer les premiers avec une quelconque des méthodes précédentes, et vérifier que la boucle suivante n'affiche rien :

```
    for (int i=3; i<65536; ++i) {
        if (is_prime[i] && !test_if_prime_fermat(i, 2)) {
            printf("Problème avec %d\n", i);
        }
    }
```

18. On peut utiliser la boucle suivante pour les trouver. Il y en a 64, les cinq premiers étant 341, 561, 645, 1105 et 1387.

```
    for (int i=3; i<65536; ++i) {
        if (!is_prime[i] && test_if_prime_fermat(i, 2)) {
            printf("Faux premier %d\n", i);
        }
    }
```

19. D'après l'OEIS, on parle de nombres de Sarrus ou de nombres de Poulet.

20. Le test de primalité est bien plus rapide si le nombre n'a pas de petits diviseurs (mais plus lent pour des entiers composite avec un petit diviseur) :

```
Pour 999789210    Fermat : 0.000102 ms
Pour 999999863    Fermat : 0.000107 ms
Pour 999999873    Fermat : 0.000102 ms
Pour 999999883    Fermat : 0.000058 ms
```

21. Les pseudo-premiers obtenus dépendent de a, d'où l'intérêt d'effectuer le test pour plusieurs valeurs de a afin de réduire les risques qu'un nombre composite soit considéré comme potentiel premier. Par exemple, en combinant $a = 3$, $a = 5$ et $a = 7$, seuls 29341 ($13 \times 37 \times 61$) et 46657 ($13 \times 37 \times 97$) sont à tort considérés comme de possibles premiers.

6 Spirale d'Ulam

22. On peut par exemple écrire la fonction suivante (on remarque, pour la construction, que l'on part du centre de l'image, et que l'on fait 1 déplacement vers le bas, 1 vers la droite, 2 vers le haut, 2 vers la gauche, 3 vers le bas, 3 vers la droite, 4 vers le haut, et ainsi de suite) :

```
void ulam(void) {
    init(255, 255, black);
    int ci = 127, cj = 127, d = 1, di = 1, dj = 0;
    for (int i=1; i<=255*255; ++i) {
        if (is_prime[i]) { set_pixel_color(ci, cj, white); }
        if (di>0) {
            if (d%2 == 1) { ci++; } else { ci--; }
            di--; if (di==0) { dj = d; }
        } else {
            if (d%2 == 1) { cj++; } else { cj--; }
            dj--; if (p==0) { d++; di = d; }
        }
    }
```

23. Sur les 257 entiers entre 0 et n de cette forme, 188 sont premiers.

24. Ces entiers forment une droite dans l'image. La spirale d'Ulam, en faisant apparaître des droites alignant de nombreux premiers, a permis d'identifier de telles formules produisant de nombreux nombres premiers.