Autour des nombres premiers

1 Introduction

Depuis la ligne de commande, naviguez vers un répertoire vous appartenant, et exécutez la commande suivante, qui téléchargera et décompressera un répetoire `premiers` contenant quelques sources C et un makefile :

```
curl cdn.sci-phy.org/mp2i/tp3-premiers.tgz | tar xvz
```

Vous y retrouverez notamment les fichiers `basic_ppm` du TP précédent, des fichiers `time_it.h` et `time_it.c`, ainsi qu'un fichier `premiers.c` à compléter et un makefile.

Durant cette séance, **nous supposerons que les entiers de type `int` sont compris entre -2^{-31} et $2^{31}-1$** (c'est le cas pour les machines de la salle de TP et plus que probablement de vos propres ordinateurs). Ces deux valeurs correspondent précisément aux constantes `INT_MIN` et `INT_MAX`).

Nous allons nous efforcer d'identifier tous les nombres premiers entre 0 et $n = 2^{16} = 65536$ (exclu). Puisque $(n-1)^2 >$`INT_MAX`, tout entier de type `int` est donc soit premier, soit un multiple d'au moins un des facteurs premiers dans $[\![0\,..\,n]\!]$. Afin de mémoriser les résultats de cette recherche et les exploiter, on définit donc dans le fichier `premiers.c` trois variables globales [1] :

- un entier `nb_primes` qui contiendra le nombre d'entiers premiers identifiés dans $[\![0\,..\,n-1]\!]$;
- un tableau de $n = 65536$ booleens `is_prime` qui indiquera, via le booléen dans la case d'index `i`, si `i` est premier ou non (les sept premières cases du tableau contiendront donc `false`, `false`, `true`, `true`, `false`, `true` et `false`, puisque 2, 3 et 5 sont les premiers nombres premiers) ;
- un tableau de 10000 entiers `primes` qui contiendra, dans les cases 0 à `nb_primes`-1 les `nb_primes` nombre premiers identifiés (et des valeurs arbitraires dans le reste du tableau).

La taille du tableau `primes` mérite une explication. On ne sait pas à l'avance exactement combien il existe d'entiers premiers entre 0 et n. On s'attend à en trouver à peu près 7000, d'après les propriétés sur la fréquence des nombres premiers. On a donc alloué un tableau pouvant (amplement) recevoir ces quelques milliers de valeurs, au fur et à mesure qu'on les calculera. Nous admettrons que l'on est certain que le nombre de premiers dans l'intervalle considéré ne dépassera pas 10000, et qu'il n'est point besoin de se préoccuper d'un éventuel débordement du tableau `primes` lorsque l'on va le remplir.

1. Généralement, on s'efforce d'éviter d'abuser de variables globales. Toutefois, comme il nous faudra utiliser ces grandeurs dans à peu près toutes les fonctions, c'est une situation ou ce choix peut éventuellement être défendu. Et faute de savoir encore comment transmettre de telles informations d'une fonction à une autre, nous on profiterons.

2 Manipulation des variables globales

À tout instant, on souhaite mémoriser la liste des entiers premiers identifiés. Pour ce faire, `nb_primes` contiendra le nombre de nombres premiers identifiés, et les `nb_primes` premières cases du tableau `primes` correspondront aux `nb_primes` plus petits entiers premiers, rangés par ordre croissant. Le contenu des autres cases du tableau `primes` n'a pas d'importance.

1. Proposer une fonction `init_primes`, ne prenant aucun paramètre et ne retournant rien, modifiant, *selon les besoins*, les variables `nb_primes` et `primes` pour indiquer que l'on n'a identifié aucun nombre premier.

L'ajout d'un premier à la liste se fait de la façon suivante. Supposons que l'on ait identifié pour l'instant trois premiers, 2, 3 et 5. La situation est la suivante :

`primes`	2	3	5	22	11	37	42	29	54	1	10	17
`nb_primes`	3											

Les valeurs estompées sont des valeurs arbitraires (les cases ne peuvent être vides, mais leur contenu est sans signification particulière). L'ajout d'un premier supplémentaire, par exemple 7, conduit alors à la situation suivante :

`primes`	2	3	5	7	11	37	42	29	54	1	10	17
`nb_primes`	4											

2. Proposer de même une fonction `add_prime` prenant en argument un entier `p` et modifiant `nb_primes` et `primes` de façon à inclure ce nouvel entier dans la liste des premiers identifiés.

3 Premiers remplissages

On retrouvera, dans le fichier `premiers.c`, la fonction `test_if_prime_basic` retournant un booléen indiquant si l'entier positif passé en argument est un nombre premier.

3. Compléter la fonction `gen_basic` testant tous les entiers de 0 à $n-1$ avec la fonction `test_if_prime_basic` et remplissant les tableaux au fur et à mesure. **On ne fait aucune hypothèse sur le contenu de `nb_primes`, `primes` et `is_prime` avant l'appel à `gen_basic`.** Le contenu de ces trois variable devra être celui souhaité après l'appel à `gen_basic`.

On dispose d'un mécanisme de test du code à travers la fonction `check_primes`, fournie dans le fichier `premiers.c`, ne prenant aucun argument et ne retournant rien, vérifiant le contenu des trois variables et affichant un bilan de ce qui a été calculé.

4. Appeler la fonction `gen_basic` suivie de la fonction `check_primes` afin de vérifier le bon fonctionnement de la première.

La fonction `test_if_prime_basic` vérifie, pour déterminer si un entier k est premier, si 2 ainsi que *tous* les entiers impairs supérieurs à 3 et inférieurs ou égaux à $\sqrt{k}$ ne sont pas des diviseurs de k. C'est inutilement couteux, il suffirait de tester tous les *premiers* inférieurs ou égaux à $\sqrt{k}$.

Lorsque l'on construit la liste des nombre premiers, on est certain de trouver ces premiers dans le tableau `primes` qui, lorsque l'on s'intéresse à k, contient déjà tous les premiers strictement inférieurs à k.

5. Proposer une fonction `test_if_prime_better` prenant en argument un entier (positif) `k` et retournant un booléen indiquant si `k` est premier. On suppose que `primes` contient, dans les `nb_primes` premières cases, au moins tous les premiers inférieurs à $\sqrt{k}$ et qu'ils sont rangés par ordre croissant.

6. En déduire une fonction `gen_better` fonctionnant comme `gen_basic` mais utilisant `test_if_prime_better`, et vérifier son bon fonctionnement.

Une fois la génération des nombres premiers entre 0 et $n = 65536$ (exclu) effectuée, on remarquera que tout nombre composite a au moins un facteur premier dans la liste ainsi construite. Par conséquent, `test_if_prime_better` fonctionne à présent pour *tous* les entiers positifs de type `int` sur les machines considérées.

7. Proposer une fonction `print_decomp` qui prend en argument un entier `k` strictement positif et affiche la décomposition de `k` en facteurs premiers. Par exemple, « `print_decomp(42)` » affichera quelque chose comme « `2, 3, 7` », « `print_decomp(37)` » affichera « `37` » et « `print_decomp(54)` » affichera « `2, 3, 3, 3` ».

8. Tester la fonction précédente avec 37, 54, 1, 999789210, 999999863, 999999873 et 999999883 (on notera les résultats de ces quatre dernières décompositions, qui nous serviront dans la suite).

4 Crible d'Erathostene

Il est possible de construire notre liste de nombres premiers plus efficacement encore. Plutôt que de tester la divisibilité de chacun des entiers k entre 0 et $n-1$ par les premiers p précédemment identifiés, on peut, dès que l'on trouve un premier p, affirmer que ses multiples ne sont *pas* des nombres premiers.

C'est le principe du « *crible d'Ératosthène* ». L'algorithme fonctionne de la façon suivante :

- on initialise les variables globales `nb_primes` et `primes` en appelant `init_primes` ;
- on place `true` dans toutes les cases du tableau `is_prime` (a priori, on suppose que tous les entiers sont possiblement premiers) ;
- puis, pour tout entier k de 2 à $n-1$ (inclus) :
  - si `is_prime[k]` contient `false`, cela signifie que `k` n'est pas premier, et on ne fait rien ;
  - si `is_prime[k]` contient `true`, alors `k` est premier, aussi ajoute-t-on `k` à la liste des premiers identifiés avec `add_prime`, puis on modifie le tableau `is_prime` de façon à indiquer que tous ses multiples propres ne sont pas premiers.

Illustrons cet algorithme. Initialement, les 40 premières cases du tableau `is_primecorrespondent` à la situation ci-dessous où les chiffres indiqués font référence aux index dans le tableau, les cases claires indiquent que la case correspondante contient la valeur `true`, les cases sombres qu'elle contient la valeur `false`) :

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39

0 n'est donc pas premier, 1 non plus, mais 2 est premier. On l'ajoute à la liste, puis on « élimine » donc ses multiples en plaçant un `false` dans toutes les cases dont les index sont des multiples de 2. Cela conduit à :

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39

3 est premier, puisque la valeur d'index 3 est `true`, on l'ajoute à la liste et on élimine ses multiples :

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39

4 n'est pas premier (car `is_prime[4]=false`), mais 5 l'est, même chose :

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39

9. Proposer une fonction `gen_erathostene` appliquant à la lettre l'algorithme du dessus. Comme pour les fonctions génératrices précédentes, on ne fait aucune hypothèse sur le contenu de `nb_primes`, `primes` et `is_prime` avant l'appel à `gen_erathostene`. Le contenu

de ces trois variables devra être celui souhaité après l'appel à `gen_erathostene`.

Note : Si vous travaillez sous Unix, en raison du risque d'écrire une fonction faisant référence à des cases hors du tableau, il sera intéressant d'utiliser la directive `-fsanitize=address` (à la fois à lors de la compilation et de la liaison) qui provoque une erreur (plutôt qu'un comportement indéterminé) si on essaie d'écrire hors du tableau. Notez que cette directive n'est qu'une aide lors du développement, on ne voudra pas forcément la conserver lorsque le programme est correct car elle ralentit (légèrement mais inutilement) le code compilé. Cette directive de compilation n'est malheureusement pas encore disponible sous Windows.

On peut encore améliorer quelque peu les choses...

10. Justifier que, pour $k > 2$, lorsque l'on élimine les multiples de k, il suffit en fait d'éliminer les multiples k^2, k^2+2k, k^2+4k... strictement inférieurs à n.

11. En déduire une fonction `gen_erathostene_2` plus efficace construisant `is_prime`, `nb_primes` et `primes` et traitant, successivement :

- le cas particulier de 2 à part ;
- les entiers impairs entre 3 et $256 = \sqrt{n}$;
- les entiers impairs entre 257 et $n-1$.

Les fichiers `time_it` fournissent une fonction `time_it` prenant comme seul argument une fonction ne prenant aucun argument, et retournant un flottant (`double`) représentant le temps d'exécution (en millisecondes) de la fonction en argument. La fonction sera appelée plusieurs fois si nécessaire pour que la mesure puisse être raisonnablement précise [2].

12. Calculer et afficher les temps nécessaire à la détermination des entiers premiers entre 0 et $n-1$ avec les fonctions `gen_basic`, `gen_better`, `gen_erathostene` et `gen_erathostene_2`. On peut afficher un flottant `x` en écrivant par exemple « `printf("x = %lf\n", x);` ».

5 Tests de primalité

On dispose pour l'instant de deux fonctions permettant de tester si un entier est premier : `test_if_prime_basic` et `test_if_prime_better` [3]

13. Utiliser ces deux fonctions pour déterminer si les entiers 999789210, 999999863, 999999873 et 999999883 sont premiers.

On souhaite déterminer, avec `time_it`, le temps nécessaire à chacune de ces deux fonctions pour retourner le résultat. L'ennui, c'est que `time_it` prend en argument une fonction ne prenant aucun argument. On ne peut donc pas l'utiliser directement. On peut donc écrire deux fonctions telles que :

```c
void test_basic(void) {
    test_if_prime_basic(999789210);
}

void test_better(void) {
    test_if_prime_better(999789210);
}
```

et utiliser, par exemple, `timeit(test_basic)` pour estimer le temps d'exécution [4] de l'appel `test_if_prime_basic(999789210)`.

14. Comparer le temps d'exécution des deux fonctions `test_if_prime_basic` et `test_if_prime_better` pour chacun des quatre entiers précédents. En se basant sur leur décomposition en facteurs premiers, interpréter les différences de temps d'exécution de chacun des appels.

Pour de grands entiers (ou si l'on a beaucoup de tests à faire), ces fonctions peuvent ne pas être suffisamment rapides. Il existe cependant, pour ces situations, des fonctions permettant de tester si un nombre a *de fortes chances* d'être un nombre premier.

En arithmétique, on dispose d'un théorème dû à Fermat intéressant pour les nombres premiers :

Théorème 1 (Petit théorème de fermat)**.** *Si p est un nombre premier, alors pour tout entier a vérifiant $1 \leqslant a < p$, $a^{p-1} \equiv 1 \pmod{p}$.*

Sa réciproque est vraie également. En revanche, pour un a donné, il est tout à fait possible que $a^{p-1} \equiv 1 \pmod{p}$ même si p n'est pas premier. C'est toutefois suffisamment rare pour que l'on puisse en faire un test qui fonctionne das la majorité des cas.

Ainsi, si un entier k vérifie $a^{k-1} \equiv 1 \pmod{k}$, alors k a de fortes chances d'être premier. Inversement, si a^{k-1} est congru à une autre valeur que 1 modulo k, alors k est, à coup sûr un nombre composite.

On souhaite implémenter une fonction utilisant ce critère (appelé *test de Fermat*) afin de tester si un nombre a de fortes chances d'être premier. Il nous faut donc un moyen de calculer rapidement une expression de la forme $\text{pow\_mod}(a,b,k) = a^b \pmod{k}$ (et sans débordement !), en supposant que a et b sont des entiers positifs, et k un entier supérieur ou égal à 2.

2. Il est donc nécessaire que les exécutions successives soient parfaitement indépendantes : on ne pourrait pas directement tester la vitesse d'un tri par insertion de la sorte, car les exécutions ultérieures seraient plus rapides puisque le tableau serait déjà trié.

3. On rappelle que cette seconde fonction ne permet de déterminer si l'entier passé en argument est premier que si les premiers inférieurs à n ont tous été déterminés précédemment par un appel à l'une des fonctions de la section précédente.

4. On supposera le temps de l'appel de fonction supplémentaire négligeable. Précison aussi que, le résultat de la fonction n'étant pas utilisé, et la fonction étant pure, sous certaines conditions, un compilateur aggressif dans son optimisation pourrait envisager de ne pas appeler la fonction du tout. Mais cela ne devrait pas arriver ici.

Pour ce faire, on utilisera les règles suivantes :

- si $b = 0$, alors pow_mod$(a, b, k) = 1$;
- si $b > 0$ et b est pair, alors pow_mod$(a, b, k) =$ pow_mod$(a, b/2, k)^2 \pmod{k}$;
- si $b > 0$ et b est impair, alors pow_mod$(a, b, k) = a \times$ pow_mod$(a, b/2, k)^2 \pmod{k}$.

L'implémentation pose une difficulté : si pow_mod(a, b, k) est nécessairement représentable par un `int` (puisque plus petit que k, les calculs (notamment le carré) peuvent excéder les capacités des `int`. Pour résoudre le problème, on utilisera le type `long int`, capable sur les machines de contenir le produit de n'importe quelle paire d'entiers[5]. Par exemple, le programme suivant calcule le carré de `mango` :

```
int mango = 987654321;

long int carre = mango;
carre = carre * mango;
```

Attention, on ne peut pas écrire directement

```
long int carre = mango * mango; // incorrect
```

car `mango` étant un `int`, le calcul serait effectué sur des `int`, et il provoquerait un débordement avant d'être rangé dans `carre`. En revanche, on peut écrire

```
long int carre = (longint)mango * mango;
```

15. Proposer une implémentation de la fonction `pow` (vu la définition proposée, il semble raisonnable d'envisager une solution sous la forme d'une fonction récursive).

16. En déduire une fonction `test_if_prime_fermat` prenant en argument un `int` k représentant un entier dont on savoir s'il est premier et un `int` a et retournant un booléen indiquant si, d'après le test de Fermat, l'entier a de fortes chances d'être premier.

17. Vérifier, pour $a = 2$ que tous les premiers entre $a+1$ et n sont bien identifiés comme premiers possibles avec ce tests.

18. Déterminer et afficher les entiers composites entre $a+1$ et n qui sont identifiés comme premiers possibles pour $a = 2$. Ces entiers sont qualifiés de « pseudo-premiers ».

19. Chercher la séquence des pseudo-premiers dans l'encyclopédie des nombres premiers et trouver les autres noms que l'on donne à cette séquence d'entiers.

20. Comparer la vitesse d'exécution (et le résultat) de `test_if_prime_fermat` sur les quatre entiers 999789210, 999999863, 999999873 et 999999883 à ce que l'on a obtenu avec les tests déterministes.

5. Mais pas nécessairement le produit d'un triplet d'entiers, attention !

21. Comparer l'efficacité de la méthode pour différentes valeurs de a. Les « pseudo-premiers » obtenus sont-ils les mêmes ?

6 Spirale d'Ulam

Durant un exposé « très long et très ennuyeux », de ses propres dires, le mathématicien Stanislaw Ulam s'est amusé à placer les entiers dans le plan de façon à obtenir une sorte de spirale, et à marquer ceux d'entre eux qui étaient premiers. Le dessin ainsi obtenu porte le nom de *spirale d'Ulam* (ou d'*horloge d'Ulam*).

43	42	41	40	39	38	37
44	21	20	19	18	17	36
45	22	7	6	5	16	35
46	23	8	1	4	15	34
47	24	9	2	3	14	33
48	25	10	11	12	13	32
49	26	27	28	29	30	31

Les fichiers `basic_ppm` permettent, rappelons-le, de créer une image vide de couleur noire de hauteur h et de largeur l en écrivant `init_image(h, l, black)`, de mettre un point blanc à l'intersection de la ligne i et la colonne j en écrivant `set_pixel_color(i, j, white)` et d'enregistrer l'image en utilisant la fonction `write_image("result.ppm")`.

22. Créer une image de taille 255×255 correspondant à la spirale d'Ulam pour les entiers de 1 à $255^2 = 65025$ (inclus).

Ce qui est intéressant dans cette image est qu'elle fait apparaître des structures, notamment des droites contenant un grand nombre de nombres premiers. Une d'elle correspond à la séquence de nombres $k^2 - k + 41$, k étant un entier positif.

23. Déterminer le nombre d'entiers de la forme $k^2 - k + 41$ compris entre 0 et n (exclu), ainsi que le nombre d'entre eux qui sont premiers.

24. Modifier le programme générant la spirale d'Ulam pour que ces premiers apparaissent dans une couleur différente.