DM #1 — Tests (presque) naïfs de (presque) primalité (avec solutions)

À terminer pour le mercredi 1er octobre 2025

Un nombre entier naturel p est dit *premier* s'il possède exactement deux diviseurs : 1 et lui-même. Autrement dit, $p \geq 2$ est premier ss.'il n'existe aucun $d \in [\![2, p[\![$ (l'ensemble (éventuellement vide) des entiers naturels supérieurs ou égaux à 2 et strictement inférieurs à p) t.q. $d \equiv 0 [p]$ (le reste dans la division entière de p par d est nul). La liste des 58 premiers nombres premiers est par exemple disponible sur : https://oeis.org/A000040. L'objectif de ce sujet est d'écrire de petites fonctions permettant de déterminer si un (petit) nombre est premier (et dans un second temps, s'il est *peut-être* premier).

`uint64_t`. On utilisera le type entier non signé `uint64_t` pour représenter les nombres manipulés, ce qui permettra de représenter tous les nombres de l'intervalle $[\![0, 2^{64} - 1]\!]$ (l'ensemble des entiers naturels supérieurs ou égaux à 0 et inférieurs ou égaux à $2^{64} - 1$). Pour pouvoir être utilisé, ce type nécessite l'inclusion du fichier d'en-tête `stdint.h` ; pour plus d'informations à son sujet, vous pouvez par exemple vous référer aux notes de cours.

Avant de commencer à programmer

1. On suppose deux variables `x` et `y` de type `uint64_t`, contenant chacune la valeur $1\,099\,511\,627\,776 = 2^{40}$. Le résultat de l'évaluation de l'expression `x * y` est-il bien défini ? Si oui, que vaut il ? Mêmes questions si `x` et `y` était de type signé `int64_t`, donc la plus grande valeur représentable est $9\,223\,372\,036\,854\,775\,807 = 2^{63} - 1$?

> Dans les deux cas la valeur du produit « calculé dans $\mathbb{Z}$ » n'est pas représentable par le type de l'expression, puisque 2^{80} est plus grand à la fois que $2^{64} - 1$ et $2^{63} - 1$.
> Dans le premier cas le résultat reste cependant bien défini car `uint64_t` est un type non signé, et `x * y` s'évalue donc au reste de la division euclidienne de 2^{80} par 2^{64}, soit $2^{16} = 65536$.
> Dans le second cas le type de l'expression est non signé, or les dépassement de capacité sur des entiers non signés ne sont pas définis. L'évaluation de l'expression est donc non définie.

2. Quelle est l'entier maximal b tel que b^2 est représentable par une valeur de type `uint64_t` ?

> Il faut que b soit tel que $b^2 < 2^{64}$, donc $b < 2^{32}$. Autrement dit, la valeur maximale de b est $2^{32} - 1 = 4\,294\,967\,295$.

Un test de primalité excessivement naïf

3. Écrivez une fonction C de signature :

```
bool is_prime_very_naive(uint64_t x)
```

telle que `is_prime_very_naive(x)` renvoie `true` (resp. `false`) si x est premier (resp. composé). Cette fonction devra utiliser l'algorithme consistant simplement à appliquer littéralement la définition d'un nombre premier, c'est à dire à tester si x possède un diviseur dans $[\![2, x[\![$.

> On propose :

```
bool is_prime_very_naive(uint64_t x)
{
    if (x < 2)
    {
        return false;
    }

    for (uint64_t t = 2; t < x; t++)
    {
        if (x % t == 0)
```

```
        {
            return false;
        }
    }

    return true;
}
```

4. Votre fonction met-elle plus de temps à détecter les nombres premiers ou les nombres composés ?

On décide qu'un nombre est composé quand il possède un facteur non trivial ; un nombre n'est décidé premier que quand il ne possède *aucun* tel facteur, et pour conclure il faut donc avoir testé *tous* les facteurs potentiels. On met donc plus de temps à détecter les nombres premiers avec cette fonction, qui correspond au cas où l'on n'est jamais sorti de la boucle de façon anticipée avec un `return false`.

5. Écrivez une fonction C de signature :

   `bool test_vn(void)`

   qui teste votre fonction `is_prime_very_naive` pour au moins 6 arguments différents dont certains sont premiers et d'autres non. Cette fonction devra renvoyer `true` (resp. `false`) si tous les tests se sont déroulés avec succès (resp. si au moins l'un des tests a échoué). Elle devra également afficher des messages d'erreur pour chaque teste ayant échoué, et un message victorieux si aucun d'entre eux n'a échoué. Par exemple, un affichage possible en cas d'échecs est :

```
test_vn: NOK: 29 is prime
test_vn: NOK: 65536 is not prime
```

   et en cas de succès :

```
test_vn: all tests successful
```

On propose :

```c
bool test_vn(void)
{
    bool all_okay = true;

    if (!is_prime_very_naive(2))
    {
        fprintf(stderr, "%s: NOK: 2 is prime\n", __func__);
        all_okay = false;
    }
    if (!is_prime_very_naive(29))
    {
        fprintf(stderr, "%s: NOK: 29 is prime\n", __func__);
        all_okay = false;
    }
    if (!is_prime_very_naive(257))
    {
        fprintf(stderr, "%s: NOK: 257 is prime\n", __func__);
        all_okay = false;
    }
    if (!is_prime_very_naive(997))
    {
        fprintf(stderr, "%s: NOK: 997 is prime\n", __func__);
        all_okay = false;
    }
    if (!is_prime_very_naive(65537))
    {
        fprintf(stderr, "%s: NOK: 65537 is prime\n", __func__);
        all_okay = false;
    }
    if (is_prime_very_naive(169))
    {
```

```
        fprintf(stderr, "%s: NOK: 169 is not prime\n", __func__);
        all_okay = false;
    }
    if (is_prime_very_naive(256))
    {
        fprintf(stderr, "%s: NOK: 256 is not prime\n", __func__);
        all_okay = false;
    }
    if (is_prime_very_naive(65536))
    {
        fprintf(stderr, "%s: NOK: 65536 is not prime\n", __func__);
        all_okay = false;
    }

    if (all_okay)
    {
        printf("%s: all tests successful\n", __func__);
    }

    return all_okay;
}
```

6. Testez.

Une variante un tout petit peu meilleure

On peut remarquer que si x est composé alors il possède au moins un diviseur $\leq \sqrt{x}$, et il est donc inutile de chercher un hypothétique diviseur si aucun n'a été trouvé dans l'intervalle $[\![2, \lfloor\sqrt{x}\rfloor]\!]$ (où $\lfloor\sqrt{x}\rfloor$ désigne la partie entière (la « troncation ») de $\sqrt{x}$).

7. Dans quels cas cette remarque permet elle d'améliorer la performance de l'algorithme utilisé dans `is_prime_very_naive`, et de combien ?

   Ceci n'améliore `is_prime_very_naive` que dans le cas où l'argument est un nombre premier (puisque sinon on en trouvera de toutes façons un facteur $\leq \sqrt{n}$, ce qui permettra de sortir de la boucle pour cette valeur d'indice au plus tard). Dans ce cas d'un nombre premier, passe d'un coût de $\approx n$ itérations (de la boucle) à $\approx \sqrt{n}$.

8. Écrivez une fonction C de signature :
   `bool is_prime_naive(uint64_t x)`
   de mêmes spécifications que `is_prime_very_naive` et qui met en œuvre l'amélioration identifiée à la question précédente.

   ATTENTION. Nous ne disposons pas (encore) de moyen efficace pour calculer $\lfloor\sqrt{x}\rfloor$; vous devez donc trouver une façon de réaliser le test de sortie de boucle sans explicitement faire ce calcul. (Et si vous savez comment, vous n'êtes pas autorisé à le faire !)

   ATTENTION. Pensez à bien prendre en compte les risques de dépassement de capacité et de *wraparound*.

   INDICE. Essayez d'exploiter le fait que pour x et y de même type entier non signé, l'évaluation de `x / y` et `x % y` ne peut *jamais* entraîner de *wraparound*.

   On propose :

```
bool is_prime_naive(uint64_t x)
{
    if (x < 2)
    {
        return false;
    }

    for (uint64_t t = 2; t <= x / t; t++)
    {
        if (x % t == 0)
```

```
        {
            return false;
        }
    }

    return true;
}
```

9. Écrivez une fonction C de signature :
`bool test_compare_vn_n(uint64_t bnd)`
qui compare les résultats de `is_prime_very_naive` et `is_prime_naive` pour tous les entiers de l'intervalle $[\![2, bnd[\![$, et affiche des messages d'erreur en cas d'incohérence et un message victorieux si tous les tests se sont déroulés avec succès. Par exemple, un affichage possible en cas d'échecs est :
`test_compare_vn_n: NOK: is 2 prime or not??`
et dans ce cas la fonction devra également renvoyer `false` ; un affichage possible en cas de succès est :
`test_compare_vn_n: all 100000 tests successful`
et dans ce cas la fonction devra renvoyer `true`.

On propose :
```
bool test_compare_vn_n(uint64_t bnd)
{
    for (uint64_t i = 0; i < bnd; i++)
    {
        if (is_prime_very_naive(i) != is_prime_naive(i))
        {
            fprintf(stderr, "%s: NOK: is %lu prime or not??\n", __func__, i);
            return false;
        }
    }

    printf("%s: all %lu tests successful\n", __func__, bnd);
    return true;
}
```

10. Écrivez une fonction C de signature :
`bool test_n_limit(void)`
qui teste que votre fonction `is_prime_naive` décide correctement que $18\,446\,744\,073\,709\,551\,557 = 2^{64} - 59$ est un nombre premier (et qui affiche un message etc.).
Si tout se passe bien, ce test doit pouvoir être passé avec succès en quelques minutes.
REMARQUES.

— Pour éviter un éventuel avertissement à la compilation, il peut être nécessaire d'écrire le littéral représentant 18 446 744 073 709 551 557 avec un suffixe `ULL` (par exemple), c'est à dire de l'écrire comme `18446744073709551557ULL`.
— Une cause d'erreur possible de votre fonction est qu'elle tombe dans une boucle infinie (pourquoi ?) ; dans un terminal, il peut alors être utile d'utiliser le programme `timeout` (s'il est installé) afin d'exécuter votre programme pour une durée maximale avant d'être « tué » (ce qui vous évite de passer votre temps à le mesurer). On peut par exemple faire `> timeout 240 ./dm1` pour exécuter le programme `dm1` pendant au plus 240 secondes ; si un message victorieux n'est pas affiché à la fin de l'exécution, cela veut dire que le programme a été tué et n'a donc pas réussi à déterminer que $2^{64} - 59$ est premier en moins de quatre minutes :(
Toujours dans un terminal, vous pouvez également choisir d'interrompre manuellement l'exécution de votre programme en utilisant la combinaison de touches `Ctrl+C` (mais soyez patient·e ; c'est *normal* que ce test prenne du temps)

On propose :
```
bool test_n_limit(void)
{
```

```
    if (!is_prime_naive(1844674407370955155))
    {
        fprintf(stderr, "%s: NOK: %lu is prime\n", __func__,
                18446744073709551557UL);
        return false;
    }

    printf("%s: all tests successful\n", __func__);
    return true;
}
```

11. Testez.

Soit x et e deux nombres naturels, et N un nombre naturel non nul, on définit l'*exponentiation modulaire* de x à e modulo N comme le nombre $r \in [\![0, N-1]\!]$ tel que $x^e \equiv r [N]$, autrement dit : le reste positif de la division entière de N par x^e.

Exponentiation modulaire lente

12. Écrivez une fonction C de signature :
`uint64_t slow_expmod(uint64_t x, uint64_t e, uint64_t n)`
telle que pour $n \in [\![1, 4294967295]\!]$, `slow_expmod(x, e, n)` renvoie le résultat de l'exponentiation modulaire de x à e modulo n. Cette fonction devra utiliser l'algorithme naïf consistant à « réduire modulo » et multiplier x par lui-même *e-1* fois.
CONSEIL. Utilisez `assert` pour vérifier les préconditions sur les arguments de `slow_expmod`. (En cas de besoin, consultez les notes de cours pour quelques informations de base à propos d'`assert`.)

On propose :
```
uint64_t slow_expmod(uint64_t x, uint64_t e, uint64_t mod)
{
    assert(mod > 0);
    assert(mod < 4294967296); // pour éviter tout wraparound

    uint64_t r = 1;
    x = x % mod;

    for ( ; e > 0; e--)
    {
        r = (r * x) % mod;
    }

    return r;
}
```

13. Écrivez une fonction C de signature :
`bool test_se(void)`
qui teste votre fonction `slow_expmod` sur au moins une petite dizaine d'arguments variés (et qui renvoie `true` etc.).

On propose :
```
bool test_se(void)
{
    bool all_okay = true;

    if (slow_expmod(2, 0, 256) != 1)
    {
```

```
        fprintf(stderr, "%s: 2^1 ~ 1 [256]\n", __func__);
        all_okay = false;
    }
    if (slow_expmod(2, 1, 256) != 2)
    {
        fprintf(stderr, "%s: 2^1 ~ 2 [256]\n", __func__);
        all_okay = false;
    }
    if (slow_expmod(2, 8, 256) != 0)
    {
        fprintf(stderr, "%s: 2^8 ~ 0 [256]\n", __func__);
        all_okay = false;
    }
    if (slow_expmod(2, 8, 500) != 256)
    {
        fprintf(stderr, "%s: 2^9 ~ 256 [500]\n", __func__);
        all_okay = false;
    }

    if (all_okay)
    {
        printf("%s: all tests successful\n", __func__);
    }

    return all_okay;
}
```

14. Testez.

Exponentiation modulaire rapide

L'algorithme d'exponentiation rapide (sous toutes ses formes) joue un rôle assez fondamental en informatique ; nous aurons peut-être l'occasion de le rencontrer une dizaine de fois cette année… Il peut se formuler de façon *récursive* ou *itérative*, et c'est pour l'instant cette dernière que nous allons considérer. Celle-ci exploite simplement les deux observations suivantes :

— si l'on écrit $e_k, \dots, e_0 \in \{0,1\}$ les chiffres de e *en base 2* (c'est à dire que $e = \sum_{i=0}^{k} e_i 2^i$), on a dans $\mathbb{Z}$ l'égalité $x^e = \prod_{i=0}^{k} x^{e_i 2^i}$ (avec la convention qu'un produit vide vaut 1) ; cette égalité est également valable « modulo N », c'est à dire que l'on a $x^e \equiv \prod_{i=0}^{k} x^{e_i 2^i} [N]$.

— la suite des puissances $x, x^2, x^4, \dots x^{2^k}$ peut se calculer en seulement k produits par mises au carré successives de x, x^2, etc. (et encore une fois, cela reste valable « modulo N »).

EXEMPLE. Soit $e = 10 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$, on a $x^{10} = x^{2^3} \times x^{2^1}$. On peut donc calculer l'exponentiation rapide de x à 10 modulo N en calculant successivement (modulo N) x^{2^1}, x^{2^2}, x^{2^3} (ce qui ne nécessite que 3 produits) et en « accumulant » (modulo N) les seules valeurs x^{2^1} et x^{2^3} dans une variable renvoyée à la fin de l'itération (ce qui ne nécessite qu'un produit), pour un total de $4 < 9$ produits.

15. Écrivez une fonction C de signature :

```
uint64_t fast_expmod(uint64_t x, uint64_t e, uint64_t n)
```

On propose :

```
uint64_t fast_expmod(uint64_t x, uint64_t e, uint64_t mod)
{
    assert(mod > 0);
    assert(mod < 4294967296);

    uint64_t r = 1;
    uint64_t pow = x % mod;
```

```
    for ( ; e > 0; e /= 2)
    {
        if (e % 2 == 1)
        {
            r = (r * pow) % mod;
        }
        pow = (pow * pow) % mod; // parfois inutile
    }

    return r;
}
```

de spécifications identiques à `slow_expmod`, mais utilisant l'algorithme itératif d'exponentiation rapide esquissé ci-dessus.

16. Écrivez une fonction C `test_compare_s_f` qui teste votre fonction `fast_expmod` en la comparant de façon appropriée à `slow_expmod`.

On propose :

```
bool test_compare_s_f(uint64_t bnd)
{
    for (uint64_t mod = 1; mod < bnd; mod++)
    {
        for (uint64_t x = 0; x < mod; x++)
        {
            for (uint64_t e = 0; e <= mod; e++)
            {
                uint64_t s = slow_expmod(x, e, mod);
                uint64_t f = fast_expmod(x, e, mod);
                if (s != f)
                {
                    fprintf(stderr,
                            "%s: NOK: %lu^%lu [%lu] ~ %lu "
                            "or %lu or what?\n",
                                    __func__,  x, e, mod,   s,    f);
                    return false;
                }
            }
        }
    }

    printf("%s: all %lu tests successful\n", __func__, (bnd - 1) * (bnd - 1) * bnd);
    return true;
}
```

17. Testez.

Test de primalité « de Fermat »

Un énoncé du « petit théorème de Fermat » est le suivant : soit p un nombre premier, pour tout $x \in [\![1, p-1]\!]$, alors $x^{p-1} \equiv 1\,[p]$. Par contraposée, pour x entier > 2, s'il existe $w \in [\![1, n-1]\!]$ tel que $w^{x-1} \equiv a \neq 1\,[n]$, alors n n'est pas premier, et w est un *témoin* (*witness*) de cette non-primalité.

18. Écrivez une fonction C de signature :

`bool maybe_is_prime_ltf(uint64_t x, uint64_t t)`

telle que `maybe_is_prime_ltf(x, t)` teste (au plus) t témoins potentiels de non primalité (par exemple distribués arithmétiquement), et conclut que x est peut-être premier si aucun témoin n'est trouvé.
REMARQUE. Il serait plus opportun de choisir aléatoirement les témoins potentiels (mais c'est un petit peu plus délicat à faire correctement en C) ; on obtiendrait alors un *algorithme probabiliste* de type « Monte-Carlo », notion que vous étudierez l'année prochaine.

On propose :

```
bool maybe_is_prime_ltf(uint64_t x, uint64_t t)
{
    if (x < 2)
    {
        return false;
    }

    if (t >= x)
    {
        t = x - 1;
    }

    uint64_t step = x / (t + 1);

    for (uint64_t w = step; w < x; w += step)
    {
        if (fast_expmod(w, (x - 1), x) != 1)
        {
            return false;
        }
    }

    return true;
}
```

En général, ce test de primalité est assez performant car pour la plupart des nombres composés il existe beaucoup plus de témoins de non-primalité pour le critère testé que de facteurs (qui sont les témoins de non-primalité pour l'algorithme naïf du début du sujet). Il existe cependant des nombres composés dits « de Carmichael » pour lesquels ce n'est pas le cas : n est « de Carmichael » si l'on a $m^{n-1} \equiv 1[n]$ pour tout $m \in [\![2, n-1]\!]$ premier avec n. Dans ce cas, les témoins de non-primalité pour le test « de Fermat » coïncident avec les témoins de non-primalité pour le test naïf qui, plus simple, serait plus rapide. La liste des 35 premiers nombres « de Carmichael » est par exemple disponible sur : https://oeis.org/A002997.

19. Essayez de retrouver expérimentalement des nombres « de Carmichael » en utilisant vos fonctions déjà écrites. Par exemple, vous pouvez chercher (mais ce n'est pas une obligation) à générer une sortie semblable à :

```
test_ltf: 29341 is maybe prime but is not; so it goes...
test_ltf: 46657 is maybe prime but is not; so it goes...
test_ltf: 115921 is maybe prime but is not; so it goes...
test_ltf: 162401 is maybe prime but is not; so it goes...
test_ltf: 252601 is maybe prime but is not; so it goes...
test_ltf: 294409 is maybe prime but is not; so it goes...
test_ltf: 314821 is maybe prime but is not; so it goes...
test_ltf: 334153 is maybe prime but is not; so it goes...
test_ltf: 340561 is maybe prime but is not; so it goes...
test_ltf: 399001 is maybe prime but is not; so it goes...
test_ltf: 410041 is maybe prime but is not; so it goes...
test_ltf: 488881 is maybe prime but is not; so it goes...
test_ltf: 512461 is maybe prime but is not; so it goes...
test_ltf: 530881 is maybe prime but is not; so it goes...
test_ltf: 552721 is maybe prime but is not; so it goes...
test_ltf: 599997 numbers tested for possible primality
          with up to 10 arithmetically distributed
          witnesses
```

On propose :

```c
void test_ltf(uint64_t bnd, uint64_t t)
{
    for (uint64_t i = 2; i < bnd; i++)
    {
        bool naive_prime = is_prime_naive(i);
        bool ltf_prime = maybe_is_prime_ltf(i, t);

        if (naive_prime && !ltf_prime)
        {
            fprintf(stderr, "%s: NOK: %lu is prime "
                            "but is not maybe prime!\n", __func__, i);
        }
        else if (ltf_prime && !naive_prime)
        {
            printf("%s: %lu is maybe prime "
                   "but is not... so it goes...\n", __func__, i);
        }
    }

    printf("%s: %lu numbers tested for possible primality "
           "with up to %lu arithmetically"
           " distributed witnesses\n", __func__, bnd - 3, t);
}
```

N.B. En modifiant légèrement le test « de Fermat » de façon astucieuse, on peut obtenir le test de primalité « de Miller-Rabin », très utilisé en pratique car ne souffrant pas du même défaut : il n'existe pas de nombres composés ayant « peu » de témoins de non-primalité pour ce test, ce qui fait qu'on en trouvera toujours *probablement* un « assez rapidement » (et à supposer qu'une conjecture extrêmement célèbre soit vraie on peut *toujours* en trouver un assez rapidement).