Problème de Syracuse

Arthur Jacquin

15 août 2021

Table des matières

1 Introduction

$$\begin{aligned} f :\ \mathbb{N}^* &\longrightarrow \mathbb{N}^* \\ n &\longmapsto \begin{cases} \frac{n}{2} & \text{si } n \equiv 0[2] \\ 3n+1 & \text{si } n \equiv 1[2] \end{cases} \end{aligned}$$

Voici la conjecture (non démontrée) de Syracuse : pour tout entier naturel n non nul, l'ensemble $\left\{k \in \mathbb{N} \mid f^k(n) = 1\right\}$ est non vide donc admet un minimum. On cherche ici une façon efficace de vérifier ce résultat pour un n donné (cela a déjà été fait jusqu'à 2^{68}).

En supposant la conjecture vraie, on définit m telle que :

$$\begin{aligned} m :\ \mathbb{N}^* &\longrightarrow \mathbb{N} \\ n &\longmapsto \min\left\{k \in \mathbb{N} \mid f^k(n) = 1\right\} \end{aligned}$$

2 Circuits logiques - additions

2.1 Additionneur 1 bit

Le bloc `ADD` est défini par le circuit suivant. En binaire, on vérifie aisément que $a_1 + a_2 = b_2 b_1$.

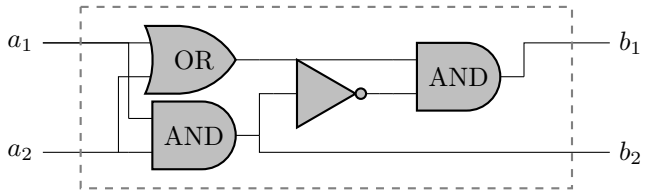

FIGURE 1 – Bloc `ADD`.

On définit le bloc `ADDr` pour prendre en compte une éventuelle retenue. On a $r + a_1 + a_2 = b_2 b_1$.

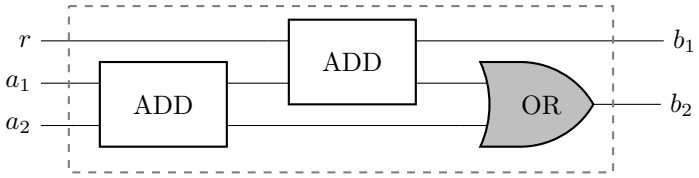

FIGURE 2 – Bloc `ADDr`.

2.2 Additionneur n bits

Le circuit suivant est un additionneur 4 bits et donne $a_4 a_3 a_2 a_1 + b_4 b_3 b_2 b_1 = c_5 c_4 c_3 c_2 c_1$. La généralisation à n bits est immédiate.

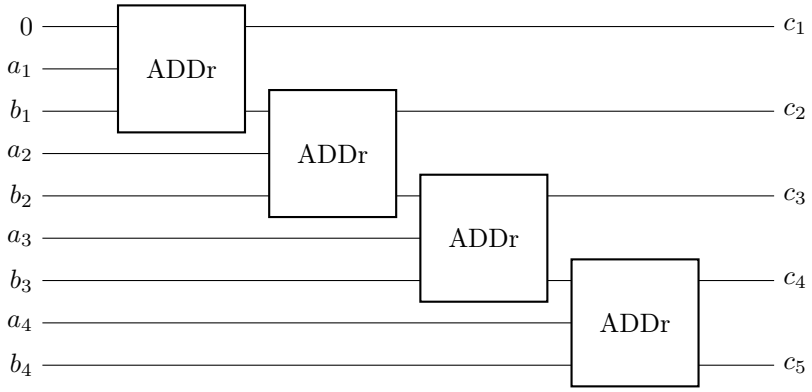

FIGURE 3 – Additionneur 4 bits.

3 Application au problème de Syracuse

Objectif Itérer efficacement dans l'algorithme de Syracuse.

Soit $n = \sum_{i=0}^{l} a_i 2^i \in \mathbb{N}$. Le cas n pair n'est pas très intéressant : la prochaine itération ($f(n) = \frac{n}{2}$) correspond à un simple décalage des bits. Essayons donc d'implémenter l'itération pour n impair ($f(n) = 3n + 1$). On remarque que $f(n)$ est alors pair : on peut donc itérer deux fois en obtenant directement $f^2(n) = \frac{3n+1}{2}$.

On remarque que $3n + 1 = (n) + (2n + 1)$. Ces deux quantités sont très faciles à former : n est déjà formé, $2n + 1$ est (en binaire) la concaténation des bits de n et de 1. On traitera l'addition restante par le même méchanisme qu'en partie 1, ici en supposant $l = 2$.

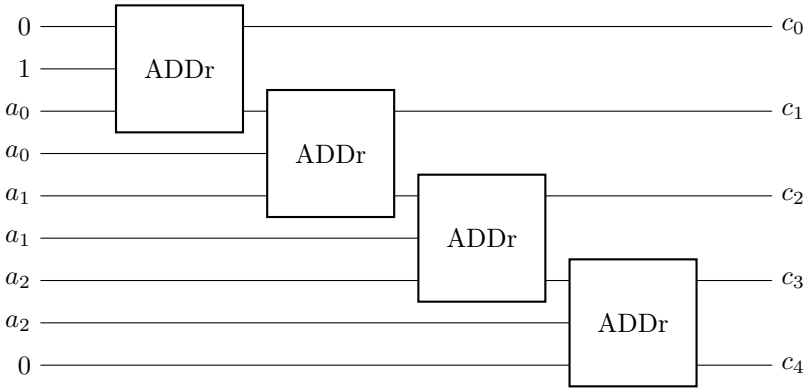

FIGURE 4 – Itérateur $n = \sum a_i 2^i \longmapsto 3n + 1 = \sum c_i 2^i$ pour $l = 2$.

On admet que n est impair, c'est-à-dire que $a_0 = 1$. Enfin, comme $3n + 1$ est pair, $c_0 = 0$. Ces deux informations étant connues, on peut simplifier le circuit. Ignorer c_0 permet également de diviser par 2. La généralisation à un l quelconque est immédiate.

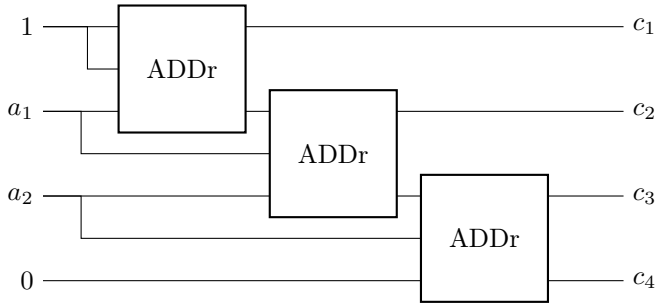

FIGURE 5 – Itérateur $n = \sum a_i 2^i \longmapsto \frac{3n+1}{2} = \sum c_i 2^{i-1}$ pour $l = 2$.

4 Implémentation en Python

On ne s'attache pas à l'optimisation de ce script ou à l'utilisation d'un fonctionnement fin de Python : ce n'est qu'une étape avant une implémentation en C, qui permettra de gérer finement les opérations sur les bits et ainsi de profiter du travail sur les circuits logiques. Par exemple, on ne soulèvera pas d'erreurs (ce méchanisme n'existe pas en C) mais nous renverrons une valeur associée à l'erreur.

Intéressons nous dans un premier temps aux nombres impairs (utilisation de la figure précédente). En ne formant $f(n)$ qu'à partir de la détection d'un 1, on peut se ramener directement au nombre impair $\frac{f(n)}{2^{Val_2(f(n))}}$. En ne manipulant uniquement des nombres impairs, on peut omettre le premier bit (correspondant à la parité). Pour étendre les fonctions aux nombres pairs, il suffit de remarquer que $m(n) = Val_2(n) + m(\frac{n}{2^{Val_2(n)}})$.

Définissons un nombre de bit maximal pour la représentation des entiers : `l = 32`. L'omission des bits de signe et la division systématique par $2^{Val_2(n)}$ permet de manipuler des quantités bien supérieures à 2^l. La fonction `iter_impair` considère donc une représentation binaire privée de son bit de parité d'un impair n et renvoie :

- La représentation binaire privée de son bit de parité de n', résultat des multiples itérations effectuées.
- Le nombre d'itérations effectuées par l'éxécution de la fonction. La fonction se base sur la figure précédente, qui opère déjà 2 itérations, mais chaque zéro ignoré avant d'écrire n' correspond également à une itération.
- Le nombre de bits significatifs de n'. S'il est nul, cela signifie que $n' = 2 * 0 + 1$, donc que l'on a opéré $m(n)$ itérations. S'il est strictement supérieur à l, il y a dépassement de capacité de stockage (*overflow*) : il est nécessaire d'augmenter l pour traiter n.

Enfin, on étends l'utilisation de la fonction à $\mathbb{N}^*$ avec la fonction `m`. Le script de la page suivante permet d'obtenir les résultats suivants :

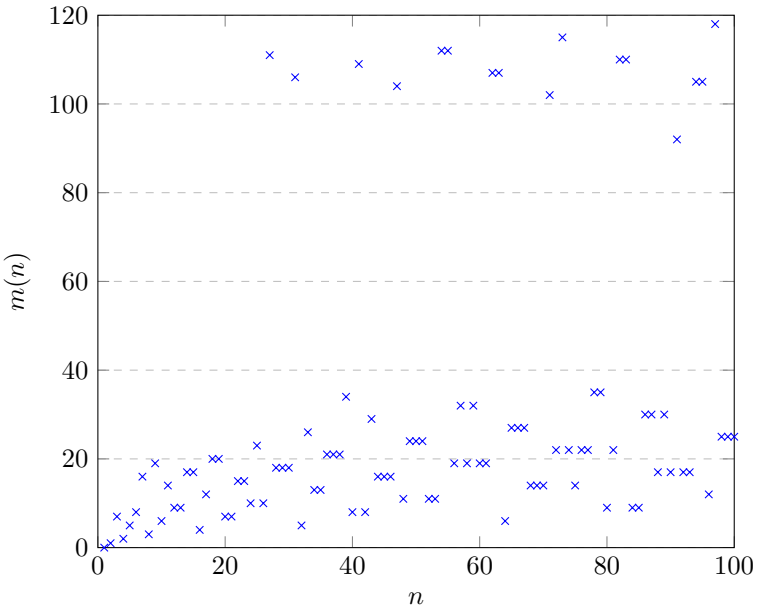

FIGURE 6 – $n \longmapsto m(n)$.

```python
l = 64 # Nombre de bit maximal

def to_bits(n):
    ''' Decomposition binaire (poids des bits croissant) '''
    return [(n >> i)%2 for i in range(l)]

def ADDr(in1, in2, in3):
    ''' Retourne les bits de in1 + in2 + in3 '''
    out2, out1 = divmod(in1 + in2 + in3, 2)
    return out1, out2

def iter_impair(bits):
    ''' Retourne la prochaine iteration impaire de n (impair) '''

    # Bits
    in1 = 1
    in2 = 1
    out = 0

    # Long
    nb_iter = 2
    max_bit = -1
    k = -1

    # Resultat
    res = [0]*l

    # Generation
    for i in range(l):
        (out, in1), in2 = ADDr(in1, in2, bits[i]), bits[i]
        if k == -1:
            if out == 0:
                nb_iter += 1
            else:
                k = 0
        else:
            if out == 1:
                max_bit = k
                res[k] = 1
            k += 1

    out, in1 = ADDr(in1, in2, 0)
    if out == 1:
        max_bit = k
        res[k] = 1

    # Detection d'overflow
    if in1 == 1:
        max_bit = l

    # Resultat
    return res, nb_iter, max_bit + 1

```

```
54 def m(n):
55     assert n > 0 # Verification n non nul
56 
57     res = 0
58     i = n
59     while i%2 == 0:
60         res += 1
61         i >>= 1
62     # i contient n/2^Val2(n) donc est impair
63     if i == 1:
64         return res
65     if i >> l + 1: # Nombre non represente par l bits
66         return 0
67     i_bits = to_bits(i >> 1) # Transformation en bits
68     while True:
69         i_bits, nb_iter, taille = iter_impair(i_bits)
70         res += nb_iter
71         if taille == l + 1:
72             return 0 # Correspond a un overflow
73         elif taille == 0:
74             return res # 1 est atteint !
75 
76 for i in range(1, 101):
77     print(m(i))
```

Listing 1 – syracuse.py

5 Implémentation en C

```c
#include <stdio.h>

unsigned long old, new; /* n et son iteration */
unsigned short int nb_iter; /* Nombre d'iterations de Syracuse dans
    une iteration */
unsigned short int k; /* Indice pour l'ecriture de l'iteration */
unsigned short int i; /* Indice de boucle */

struct { /* ADDr */
    unsigned in1:1; /* Entree 1 */
    unsigned in2:1; /* Entree 2 */
    unsigned in3:1; /* Entree 3 */
    unsigned tmp:1; /* Sortie 1 du premier bloc ADD */
    unsigned out1:1; /* Sortie 1 */
    unsigned out2:1; /* Sortie 2 */
} c;

void ADDr () {
    c.tmp = (c.in2 | c.in3) & (~c.in2 | ~c.in3);
    c.out1 = (c.in1 | c.tmp) & (~c.in1 | ~c.tmp);
    c.out2 = (c.in1 & c.tmp) | (c.in2 & c.in3);
};

struct { /* Suivi de l'ecriture de l'iteration */
    unsigned debute:1;
    unsigned overflow:1;
} bools;

void iter_impair () {
    /* Initialisation des variables */
    c.in1 = 1;
    c.in2 = 1;
    bools.debute = 0;
    k = 0;

    /* Initialisation du resultat */
    nb_iter++;
    nb_iter++;
    new = 0;

    /* Ecriture de l'iteration */
    for (i = 0; i < 8*sizeof(old); i++) {
        c.in3 = old & 1;
        ADDr();
        c.in1 = c.out2;
        c.in2 = old & 1;
        old >>= 1;
        if (bools.debute) {
            new += c.out1 << k;
            k++;
        } else {
```

```
51              if (c.out1) {
52                  bools.debute = 1;
53              } else {
54                  nb_iter++;
55              }
56          }
57      }
58      c.in3 = 0;
59      ADDr();
60      new += c.out1 << k;
61
62      /* Detection d'overflow */
63      bools.overflow = c.out2;
64  };
65
66  int m (unsigned long n) { /* Retourne 0 en cas d'erreur, m(n) sinon */
67      /* Verification n non nul */
68      if (n == 0) {
69          return 0;
70      };
71
72      /* Initialisation des variables */
73      nb_iter = 0;
74      while (~n & 1) { /* Division par Val2(n) (obtention d'un impair)
        */
75          n >>= 1;
76          nb_iter++;
77      };
78      old = n >> 1; /* Suppression du bit de parite */
79
80      /* Iterations */
81      while (old) {
82          iter_impair();
83          old = new;
84          if (bools.overflow) {
85              return 0;
86          };
87      };
88
89      /* Resultat */
90      return nb_iter;
91  };
92
93  int main () {
94      for (int j = 1; j <= 100; j++) {
95          printf("%d\n", m(j));
96      };
97
98      return 0;
99  };
```

Listing 2 – `syracuse.c`

6 Comparaison des performances

```python
def f(n):
    if n % 2 == 1:
        return 3*n + 1
    else:
        return n >> 1

def m_naive(n):
    res = 0
    while n != 1:
        n = f(n)
        res += 1
    return res
```

Listing 3 – `syracuse_naif.py`

```c
long f (long n) {
    if (n & 1) {
        return 3*n + 1;
    } else {
        return n >> 1;
    };
};

long m (long n) {
    int res = 0;
    while (n >> 1) {
        n = f(n);
        res++;
    };
    return res;
};
```

Listing 4 – `syracuse_naif.c`

On mesure le temps de calcul des $m(n)$ avec $n \in [\![1; N]\!]$. Les versions en C sont (à méthode identique) notablement plus rapides, probablement grâce à la compilation. En revanche, les versions naïves sont (à language identique) notablement plus rapides : les efforts d'optimisations ont été inutiles.

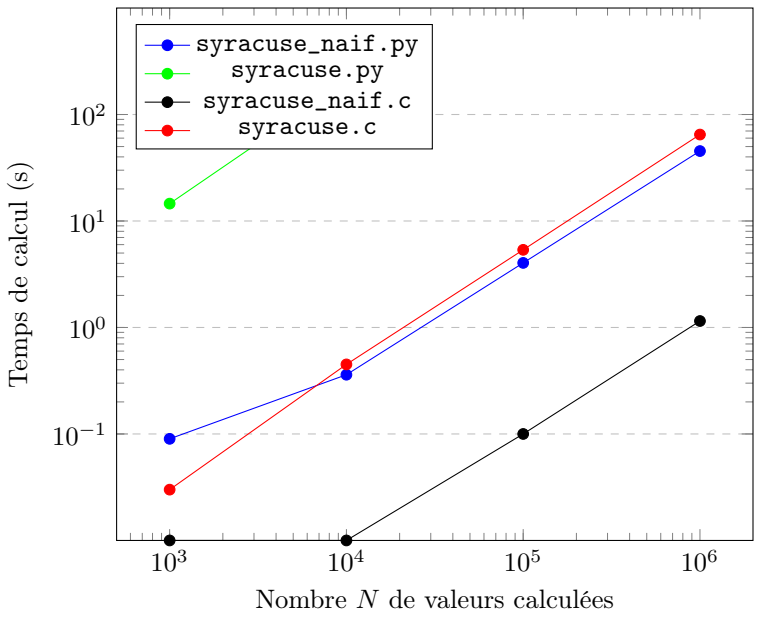

FIGURE 7 – Comparaison des scripts