Fast integer operations

Abhiram K.
Dheeraj D.
Shrivathsa P.
Suhas G.

March 15, 2020

Abstract

Integer operations such as factorial and factorization of large numbers are used almost everywhere. Hence, the search for the most efficient algorithm—in terms of both time and space—to evaluate such integer operations is a widely researched area. In this report, we first present a few cycle detection algorithms. We then use these algorithms to demonstrate a fast factorisation algorithm. In the last section, we discuss some algorithms to compute factorials quickly. We have also provided implementations of the said algorithms using Python.

Contents

1 Cycle detection algorithms

For a given set S with finite cardinality N, we may define N^N distinct maps from the set to itself. Suppose f is one such function chosen at random. We define a sequence of elements $\{x_i\}_{i=0}^{\infty}$ as follows: $x_0 \in S$ and $x_{i+1} = f(x_i)$ holds for all $i \geq 1$. As S has finite cardinality, the sequence must cycle at some point. We use μ to denote the index of the first element in the sequence $\{x_i\}_{i=0}^{\infty}$ that occurs more than once. We define T—the period—to be the minumum number such that $x_{\mu+T} = x_\mu$ holds. For obvious reasons, a "good" pseudorandom sequence generated this way should have large μ and T values. Cycle detection algorithms are algorithms that help contrive the values $\{\mu, T\}$ from an iterated sequence such as the one defined above.

There are many established cycle-detection algorithms with varying time and space complexities. It is preferable to minimise the number of f evaluations in the algorithm to make the algorithm more efficient. It is also preferred that the algorithm minimises the space used as memory. An obvious algorithm is to keep track of each element of the sequence and check for repetitions as we move along the sequence. However, this is not feasible for large values of μ and T, as it requires huge amounts of memory for the algorithm to work. Floyd and Brent are two cycle detection algorithms that use constant memory. In Sections 1.1 and 1.2, we talk about these two algorithms. In Section 2.1, we use these algorithms to obtain an algorithm that factorises a given number quickly. Most of the work presented in these sections is derived from [1].

1.1 Floyd's algorithm

We give the pseudo-code first, and then elaborate on it.

```
$x = x_0$   $y = x_0$   $i = 0$
    **repeat**   $i = i + 1$   $x = f(x)$   $y = f(y)$
    **until**   $x = y$
```

Using the above algorithm, we get the output i to be a multiple of the period T. We may then obtain the exact period T by finding the index of the first term after x_i that equals x_i and then subtracting i from the index. By checking $x_{j+T} = x_j$ for each j starting from zero until the equality is reached, we may obtain μ for the sequence.

This algorithm uses two pointers to keep track of elements. Both pointers move through the sequence at different speeds. It is common to call the slower one the "tortoise" and the faster one the "hare". We shall adhere to this terminology. With both of them starting from the first term x_0, at each stage the tortoise moves by one step and the hare by two. Thus, after i steps, the tortoise is at x_i and the hare is at x_{2i}. We note that the gap between the hare and the tortoise increases by one after each step of the process. Hence, when the tortoise is at position x_j for some $j \geq \mu$, and the gap j is a multiple of T, we have the two pointers to point to the same value and the process stops.

We observe that the value of j above is equal to μ if μ divides T. Otherwise, it is given by the value $T(1 + \lfloor \mu/T \rfloor)$. In any case, it is bounded by $\mu + T$. Now, the algorithm evaluates f thrice in each step, once for the tortoise and twice for the hare, until the inital output. Thus, the number of f evaluations, which we shall call the "work" of an algorithm, is bounded by $3(\mu + T)$. Further, we note that for $x_j = x_{2j}$ to hold, j must be greater than or equal to both μ and T. Thus, the worst-case work of the algorithm is greater than $3\max(\mu, T)$, which serves as an upper bound for the work of Brent's algorithm, which we will be showing in the next section. The benefit of Floyd's algorithm is that it takes up $O(1)$ storage space.

1.2 Brent's algorithm

Brent's algorithm is on average 36% faster than Floyd's algorithm [1], and as mentioned in the previous section, we will be showing that the worst-case work is also better than Floyd's algorithm. We first give the pseudo-code for the algorithm.

```
$x = x_0$   $y = x_1$   $r = 1$ $i = 0$
    while   $x \neq y$
        repeat   $x = y$   $i = 0$   $r = 2r$
            repeat   $i = i + 1$   $y = f(y)$
            until   $i = r$
```

The i outputted by the above algorithm gives the exact value of T. We may then find μ the same way we found it in Floyd's algorithm.

The algorithm above uses the constant two as a parameter. However, Brent's algorithm includes a whole family of algorithms with two being replaced by any suitable number. In fact, it has been shown that the algorithm is most efficient when the parameter is approximately equal to 2.4771. But most implementations use 2 instead as it simplifies calculations and has a better worst-case work-rate. We now explain the algorithm.

This algorithm also involves using two pointers, which we call the tortoise and the hare as before, with the process stopping when the two pointers point to the same value after the initial stage. At the start, both the pointers point at x_0. Subsequently, the tortoise does not move while the hare moves a number of steps greater than or equal to the given parameter (in our case two). Then, the tortoise "leaps" to where the hare is. In the next stage, the tortoise remains fixed once again, while the hare traverses the sequence until the number of steps it covers in this stage is greater than equal to the square of the parameter. This process continues, with the length of the sequence covered by the hare increasing exponentially (i.e, $r, r^2, r^3, \dots r^m$ where r is the parameter of the algorithm) each stage.

We find the work of the algorithm when the parameter is equal to two. Now, the values the tortoise takes is $x_0, x_2, x_6, \dots x_{2^n-2}$. We try to find a bound for n. Assume that $T \geq \mu$ holds. For the process to stop, the length the hare covers in the last stage has to be equal to T. Thus, the permissible number of steps for the hare to move before the tortoise's next leap is greater than or equal to T. The first time this happens is when the tortoise is at some term between x_T and x_{2T}. Hence, in this case, $2^n - 2$ is bounded by $2T$, and the work by $2T + T$. Now, suppose that $\mu \geq T$ holds. Then it is clear that if the tortoise is at any term between x_μ and $x_{2\mu}$ (although the algorithm can terminate before this can happen), the hare covers a distance equal to the period of the cycle before the tortoise makes its next jump, and hence ends the process. As a result, we see that the work is bounded by $2\mu + T$ in this case. Thus, the work of the algorithm is bounded by $2\max(\mu, T) + T$, which is less than or equal to $3\max(\mu, T)$. Thus, Brent's worst-case work is better than Floyd's. We note that Brent's algorithm also uses $O(1)$ storage space.

1.3 Implementations

Floyd's algorithm :

```
def floyd(f, n, start):

    tortoise = f(start)
    hare = f(f(start))

    while tortoise != hare:
        tortoise = f(tortoise)
        hare = f(f(hare))

    tortoise = start

    mu = 0

    while tortoise != hare:
        tortoise = f(tortoise)
        hare = f(hare
        mu += 1

    T = 1
    hare = f(tortoise)
    while hare != tortoise:
        hare = f(hare)
        T += 1

    return mu, T
```

Brent's algorithm :

```
def brent(f, n, start):

    tortoise = start
    hare = f(start)
    power = 1
    T = 1

    while hare != tortoise:
        if power == T:
            T = 0
            tortoise = hare
            power *= 2
        hare = f(hare)
        T += 1

    mu = 0
    tortoise = start
    hare = tortoise

    for i in range(T):
        hare = f(hare)

    while hare != tortoise:
        tortoise = f(tortoise)
        hare = f(hare)
        mu += 1

    return mu, T
```

2 Factorisation

Factorisation of a number is listing all the prime factors of the given number. This seemingly useless operation became extremely important when it became a key component of cryptology, which exploited the fact that factorizing large numbers quickly is extremely difficult. Thus, finding algorithms that efficiently factorise large numbers is a widely researched area. The naive way is to check the divisibility of each positive integer less than the given number. A slightly better algorithm is to check for integers less than or equal to the square root of the given number, as any number can have at most one prime factor greater than its square root. However, this algorithm also proves to be extremely slow for large numbers. In this section, we present an algorithm to factorise numbers that is fairly quick, which uses the cycle detection algorithms discussed in Section 1.

2.1 Pollard's ρ algorithm

Let n be the number we wish to factorise. The idea is to first obtain a pseudo-random sequence of numbers from the set S containing integers from 1 to $n-1$ using a function like $x^2+1 \mod (n)$ or $x^2-1 \mod (n)$. These functions are chosen as they are easy to compute and give a pseudo-random sequence [1]. Then, we apply either Floyd's algorithm or Brent's (preferably Brent's, as it makes the algorithm about 24% faster [1]) to the sequence with termination condition being $\gcd((y-x),n) \geq 1$, where y and x are elements pointed by the hare and the tortoise respectively. We then get a factor n unless $y-x=0 \mod (n)$ holds, in which case the algorithm fails (although this case is uncommon) and we may have to choose a different pseudo-random sequence. We then repeat this procedure on $\gcd((y-x),n)$ and $n/\gcd((y-x),n)$ to obtain further factors.

The core idea behind the algorithm is that a pseudo-random sequence generated from a smaller set cycles quicker than a sequence generated from a larger set. Thus, if p is a factor of n, then the sequence generated by $x^2+1 \mod (p)$ cycles faster than $x^2+1 \mod (n)$, hence the termination condition more often than not is when $\gcd((y-x),n) < n$.

The algorithm uses only a small amount of space and the expected running time is of order $O(\sqrt{p})$ where p is the smallest prime factor of n [7].

The reason the algorithm is called Pollard's "ρ" algorithm is because the eventual cycling of the pseudo-random sequence used in the algorithm looks like ρ. This is depicted in Figure 1.

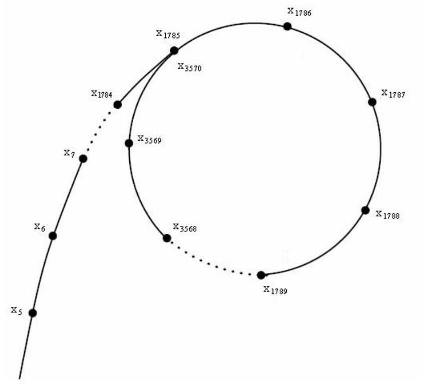

Figure 1: Cycling in pseudo-random numbers looks like the greek symbol "ρ"
Source: Wikipedia [7]

2.2 Algorithm for factorisation

For large numbers, Pollard's ρ algorithm is an efficient algorithm to find a non trivial factor. We can factorise small numbers by simply going through the list of primes below the said number. We can use these two approaches to efficiently factorise large numbers.
Here we call a number large if it is larger that 10^{16} and small otherwise. First, we check for divisibility by small primes, say below 5000. Then we recursively find factors based on the size of the number and factorise it. We present an implmentation using this approach in the nest section.

2.3 Implementations

Pollard's ρ algorithm for finding a non trivial factor :

```
def pollard_rho(n, seed = 2):
    tortoise = seed
    hare = (seed**2+1)
    power = 1
    T = 1
    factor = 1
    while factor == 1:
        while T <= power and factor == 1:
            if power == T:
                T = 0
                tortoise = hare
                power *= 2
                factor = gcd(tortoise - hare, n)
            T += 1
            hare = (hare**2+1)
            factor = gcd(tortoise-hare, n)

    return factor
```

Factorisation using Pollard's ρ algorithm : The first function **factorise** factorises n using all the primes below n while the second function **factorise_p** uses Pollard's ρ algorithm to find non trivial factors of large numbers and uses **factorise** for small numbers.

```
def factorise(n):
    factors = []
    while n%10 == 0:
        factors += [2, 5]
        n = n//10
    while n%10 in [2, 4, 6, 8]:
        factors.append(2)
        n = n//2
    while sum([int(x) for x in str(n)])%3 == 0:
        factors.append(3)
        n = n//3

    primes = [2]+primesfrom3to(floor(sqrt(n)+1)).tolist()

    for p in primes:
        while n%p == 0:
            factors.append(p)
            n = n/p
    if n != 1:
        factors.append(n)
    return factors


primes = prime_range(2, 5000)


def factorise_p(n, check = 0):
    temp = []
    if not check:#loops through primes only once despite recursion
        for p in primes:
            while n%p == 0:
                n = n//p
                temp.append(p)

    if n > 10**16:
        factor = pollard_rho(n, seed = 2)
        i = 1
        while factor == n:
            factors = pollard_rho(n, seed = 2+i)
            i += 1
        factors = [factor, n//factor]
        for divisor in factors:
            temp += factorise_p(divisor, check = 1)
    else:
        temp += factorise(n)

    return temp
```

3 Factorials

We define the factorial of a natural number n to be the product $n(n-1)\dots 2.1 = \prod_{i=1}^{i=n} i$. It is denoted by $n!$. We have $0! = 1$ by convention. In this section, we look at some algorithms that calculate the factorial of a given number.

3.1 A note on some simple algorithms

One obvious way to compute the factorial of a number n is to do it as in the definition: recursively or iteratively by multiplying all the numbers less that or equal to n. However, this requires a lot of multiplications. An improvement on this algorithm is to compute $n!$ when n is even by grouping the end terms as follows:

$$n! = (1.n)(2.(n-1))\dots(\frac{n}{2}.(\frac{n}{2}+1) \tag{1}$$

We use the identity $n! = n(n-1)!$ when n is odd. Observe that

$$(i+1)(n-i) - i(n-i+1) = n - 2i \tag{2}$$

holds. Hence, we can iteratively find the difference between consecutive terms in equation 1. This enables us to find the consecutive terms of equation 1, starting from n, by using addition instead of multiplication. Thus, the number of multiplications required to compute $n!$ is reduced by half.

Here is an example: to calculate 8!, we start with 8. To this, we add the successive differences to get the sequence

$$8, 8+(8-2), 8+(8-2)+(8-4), 8+(8-2)+(8-4)+(8-6)$$

that is

$$8, 14, 18, 20$$

On multiplying these terms, we get $8.14.18.20 = 40320 = 8!$

3.2 Swing numbers

The swing factorial of a number is given by the ratio $\frac{n!}{\lfloor n/2 \rfloor !^2}$ and is denoted by $n\wr$. If n is even, then $n\wr = \binom{n}{\lfloor n/2 \rfloor}$ holds, otherwise $n\wr = \binom{n}{\lfloor n/2 \rfloor}(\frac{n+1}{2})$ holds. Hence, it is always an integer.

We now prove a useful property of swing numbers.

Theorem 1. *Let $l_p(n\wr)$ denote the exponent of a prime p in the prime factorisation $n\wr$. Then, we have the following equation:*

$$l_p(n\wr) = \sum_{k\geq 1} \lfloor \frac{n}{p^k} \rfloor \mod 2 \tag{3}$$

Proof. This proof is taken from [2]. From Legendre's theorem on the prime factorisation of $n!$, we have

$$\begin{aligned} l_p(n\wr) &= l_p\left(\frac{n!}{\lfloor n/2\rfloor!^2}\right) \\ &= l_p(n!) - 2l_p(\lfloor n/2\rfloor!) \\ &= \sum_{k\geq 1}\lfloor n/p^k\rfloor - 2\sum_{k\geq 1}\lfloor\lfloor n/2\rfloor/p^k\rfloor \\ &= \sum_{k\geq 1}(\lfloor n/p^k\rfloor - 2\lfloor\lfloor n/p^k\rfloor/2\rfloor) \end{aligned}$$

Now, using the result $m - 2\lfloor m/2\rfloor = m \bmod 2$ for all integers m, we get the result stated. $\square$

Some more properties of swing numbers can be found in [2].

3.3 Factorials using swing numbers

We observe that $n! = \lfloor \frac{n}{2}\rfloor!^2 n\wr$. This recursion allows us to implement a divide-and-conquer algorithm to compute factorials. Once we have a list of primes below n (this can be achieved by using prime number sieves, such as the Eratosthenes sieve) it is quite easy to compute $n\wr$ using Theorem 1.

3.4 A different approach : Moessner's algorithm

Numbers of the form $\frac{n(n+1)}{2}$, where n is a natural number, are called Triangular numbers. Moessner's algorithm uses these numbers to generate factorials. To compute $n!$, the first step of the algorithm is to write down numbers from 1 to $T_n = \frac{n(n+1)}{2}$ and mark all T_k where $k \leq n$. Then, we add up the numbers that have not been marked so far, writing the running sum in the appropriate spots. We then circle the first, third, sixth, ... numbers of this new sequence. We repeat this procedure until we end up with a single number, which turns out to be the factorial of n. For a proof of this fact, the interested reader can look at [5]. We demonstrate this procedure when $n = 4$:

First step	:	$\boxed{1}$	2	$\boxed{3}$	4	5	$\boxed{6}$	7	8	9	$\boxed{10}$
Second step	:		$\boxed{2}$		6	$\boxed{11}$		18	26	$\boxed{35}$	
Third step	:				$\boxed{6}$			24	$\boxed{50}$		
Fourth step	:							$\boxed{24}$			

This algorithm is very fast when n is small. However when n is large, there are faster ways (such as using the swing numbers) of computing $n!$.

3.5 Implementations

Calculating swing numbers:

```
def swing(n):
    primes1 = prime_range(sqrt(n)+1)
    primes2 = prime_range(sqrt(n)+1, n//3+1)
    primes3 = prime_range(n//2+1, n+1)
    prod = mul(primes3)
    for p in primes1:
        exp = 0
        num = n//p
        while num != 0:
            exp += num%2
            num = num//p
        prod *= p**exp
    for p in primes2:
        if n//p&1:
            prod *= p
    return prod
```

Factorial using swing numbers:

```
def swing_factorial(n):
    prod = swing(n)
    mult = 2
    while n != 1:
        n = n>>1
        prod *= swing(n)**mult
        mult *= 2
    return prod
```

The following code is taken from [3].

```
def factorialPS(n):

    small_swing = [
        1,1,1,3,3,15,5,35,35, 315, 63, 693, 231, 3003, 429, 6435, 6435,
        109395,12155,230945,46189,969969,88179,2028117,676039,16900975,
        1300075,35102025,5014575,145422675,9694845,300540195,300540195 ]

    def swing(n):
        if n < 33: return small_swing[n]
        sqrtn = n.sqrt()
        factors = prime_range(n // 2 + 1, n + 1)

        for prime in prime_range(3, sqrtn + 1):
            p, q = 1, n

            while True:
                q //= prime
                if q == 0: break
                if q & 1 == 1:
                    p *= prime

            if p > 1: factors.append(p)

        R = prime_range(sqrtn + 1, n // 3 + 1)
        factors += filter(lambda x: (n // x) & 1 == 1, R)

        return mul(factors)

    def odd_factorial(n):
        if n < 2: return 1
        return (odd_factorial(n//2)**2)*swing(n)

    if n == 0: return 1
    if n < 0 : return 0
    if n < 20: return mul(range(2, n + 1))

    N, bits = n, n
    while N != 0:
        bits -= N & 1
        N >>= 1

    return odd_factorial(n)*(2**bits)
```

Factorials using Moessner's algorithm : This implementation is based on the one found in [4].

```
def moessner(n):
    s = matrix.zero(n+1).list()
    s[0] = 1
    for m in range(1, n+1):
        for k in range(m, 0, -1):
            for i in range(1, k+1):
                s[i] += s[i-1]
    return s[n]
```

4 Acknowledgement

References

[1] Richard P. Brent
https://maths-people.anu.edu.au/ brent/pd/rpb051i.pdf

[2] P. Luschny
http://www.luschny.de/math/factorial/SwingIntro.pdf

[3] P. Luschny
http://www.luschny.de/math/factorial/SwingFactorialSagePython.html

[4] P.Luschny
http://www.luschny.de/math/factorial/csharp/FactorialAdditiveMoessner.cs.html

[5] Dexter Kozen and Alexandra Silva.
https://www.cs.cornell.edu/ kozen/Papers/Moessner.pdf

[6] John H. Conway and Richard K. Guy.
The Book of Numbers. Copernicus.

[7] Wikepedia