Barrett Reduction Algorithm

Use:

- It is used for repeated computation of ab mod n where ($0 \leq ab < n^2$) using modular reductions than use long division which is faster as it requires only multiplication ,subtraction and shifts
- Assumption n is not power of 2;$n \geq 3$ otherwise computation becomes too simple

Algorithm

Precomputation:

- Factor k such that $2^k > n$, smallest choice is $[(\log(n) \text{ base } 2)]$ (ceil)
- Factor $r = [4^k/n]$(ceil)

Reduction:

- For calculating x mod n such that $(0 \leq x < n^2)$ calculate $t = x - [x*r/4^k]*n$
- If $t < n$ then x mod n = t
- If $t > n$ then x mod n = $t - n$

Algorithm 1 The Cooley-Tukey NTT algorithm

INPUT: A vector $\boldsymbol{x} = [x_0, \ldots, x_{n-1}]$ where $x_i \in [0, p-1]$ of degree n (a power of 2) and modulus $q = 1 \bmod 2n$

INPUT: Precomputed table of $2n$-th roots of unity $\boldsymbol{g}$, in bit reversed order

OUTPUT: $\boldsymbol{x} \leftarrow NTT(\boldsymbol{x})$

```
function NTT(x)
    t ← n/2
    m ← 1
    while m < n do
        k ← 0
        for i ← 0; i < m; i ← i + 1 do
            S ← g[m + i]
            for j ← k; j < k + t; j ← j + 1 do
                U ← x[j]
                V ← x[j + t].S mod q
                x[j] ← U + V mod q
                x[j + t] ← U − V mod q
            k ← k + 2t
        t ← t/2
        m ← 2m
    return
```

Use of Barrett in NTT

- U,V here represent the Even and Odd part of the NTT equation
- Recursive calling of function NTT function can also be used instead of 2 loops
- Barrett or Montgomery algorithm can be used for optimizing the modular operations performed while computing V and computation of g vector

Recursive Algorithm of NTT

```python
def number_theoretic_transform(f, p, omega):
    n = len(f)
    if n == 1:
        return f

    f_even = number_theoretic_transform(f[::2], p, (omega * omega) % p)
    f_odd = number_theoretic_transform(f[1::2], p, (omega * omega) % p)

    w = 1
    n_half = n // 2
    result = [0] * n

    for i in range(n_half):
        even = f_even[i]
        odd = (f_odd[i] * w) % p

        result[i] = (even + odd) % p
        result[i + n_half] = (even - odd) % p

        w = (w * omega) % p

    return result
```

Code for Barrett

BARRET ALGORITHM

Barrett.py > ...

```python
"""Barrett Redution Algorithm"""
import time
from csv import writer
from random import randint
start_time = time.time()
n = 3**23
k = n.bit_length()#finds the bit length of the modulus
r = (1<<k*2)//n#right shift 1 by bit length *2 to compute 4^k
def bar(x) -> int:
    t = x - ((x*r)>>k*2)#shift left by 2k to divide by 4^k
    if t<n :
        return(t)
    else:
        return(t-n)
for _ in range (1,100):
    num = randint(3**23,3**46)
    print(f"{bar(num)}")
time_passed = time.time() - start_time
print(f" {time_passed} seconds ")
List = [time_passed]
with open('time_bar.csv', 'a') as f_object:
    writer_object = writer(f_object)
    writer_object.writerow(List)
    f_object.close()
```

TRADITIONAL

normal_mult.py > ...

```python
import time
from csv import writer
from random import randint
start_time = time.time()
n = 3**23
for _ in range (1,100):
    num = randint(3**23,3**46)
    print((num%n))
time_passed =time.time()-start_time
print(f"{time_passed} seconds")
List = [time_passed]
with open('time_norm.csv', 'a') as f_object:
    writer_object = writer(f_object)
    writer_object.writerow(List)
    f_object.close()
```

Time complexity

- Number of test cases : 80
- Barrett Reduction on average : 0.014517248 seconds
- Traditional on average: 0.017941624 seconds

References

- https://www.nayuki.io/page/barrett-reduction-algorithm
- https://en.wikipedia.org/wiki/Barrett_reduction
- https://eprint.iacr.org/2017/727.pdf
- https://www.nayuki.io/page/number-theoretic-transform-integer-dft