Design and Analysis of Algorithms Assignment - 4

Department of Information Technology,

Indian Institute of Information Technology, Allahabad 211015, India

AAMIN CHAUDHARI(IIT2019206),SUBHASH BALLA(IIT2019207), DHANUSH VASA (IIT2019208)

INTRODUCTION

The problem is to find out the product of two polynomials.Divide and conquer paradigm is used to solve this task.we will see that run time of this algorithm based on divide and conquer approach is efficient compared to the naive algorithm.Usually the approach of Divide and Conquer exhibits the logarithmic time.Divide and conquer algorithm recursively breaks down a problem into two or more sub problems of the same type.

Advantages of Divide and Conquer The advantages of using Divide and conquer algorithm reflects in time and space complexity tremendously than using all the brute force approaches.

Time Efficiency
Complexity of Brute Force is in general high order in nature.We will see that divide and conquer based algorithm reduces the run time from $O(n^2) to O(n^{\log_2 3})$.

ALGORITHM DESIGN

This method is mainly based on divide and conquer. Following is a simple method that divides the given polynomial (of degree n) into two polynomials containing lower degree terms(lower than n/2).

Data Structures Used:

- Arrays to store the set of polynomials and their result.

Assume both A(x),B(x) polynomials are of degree n.

ATTEMPT 1: let,

$$A(x) = a_0 + a_1.x + ... + a_n.x^n$$

$$and$$

$$B(x) = b_0 + b_1.x + ... + b_n.x^n$$

Define

$$A_0(x) = a_0 + a_1.x + ... + a_{[n/2]-1}.x^{[n/2]-1}$$

$$A_1(x) = a_{[n/2]} + a_{[n/2]+1}.x + ... + a_n.xn - [n/2]$$

where [.] is GIF.

Then

$$A(x) = A_0(x) + A_1(x).x^{[n/2]}$$

similarly

$$B(x) = B_0(x) + B_1(x).x^{[n/2]}$$

we have

$$A(x).B(x) = A_0.B_0 + (A_0.B_1 + A_1.B_0).x^{[n/2]} + A_1.B_1.X^{2.[n/2]}$$

we can see that original problem of size n is divided into 4 sub problems of degree n/2.

$$(i.e; computing A_0B_0, A_0B_1, A_1B_0, A_1B_1)$$

After computing them,we simply find

$$A(x).B(x) = A_0.B_0 + (A_0.B_1 + A_1.B_0).x^{[n/2]} + A_1.B_1.X^{2.[n/2]}$$

which is simply polynomial addition (kind of) and hence takes O(n) time. so the recurrence we get is

$$T(n) = 4.T(n/2) + O(n)$$

solving this recurrence, we get

$$T(n) = O(n^2)$$

But this is not what we wanted since naive algorithm also works in $O(n^2)$.

ATTEMPT 2: we are computing 4 sub problems.CAN WE DO ANY BETTER?
consider ,

$$X = A_0.B_0$$
$$Y = A_1.B_1$$
$$Z = (A_0 + A_1).(B_0 + B_1)$$

required is

$$(A_0.B_1 + A_1.B_0) = Z - X - Y$$

hence,by only computing 3 sub problems (X,Y,Z) we can get

$$A(x).B(x) = X + (Z - X - Y).X^{[n/2]} + Y.x^{2.[n/2]}$$

the recurrence relation(discussed in analysis part) what we get now is ,

$$T(n) = 3.T(n/2) + O(n)$$

solving this recurrence,we get

$$T(n) = O(n^{log_2 3}) = O(n^{1.585})$$

which is obviously better than $O(n^2)$.

Code Implementation

```
#include<bits/stdc++.h>
using namespace std;
#define IOS ios_base::sync_with_stdio(false),cin.tie(NULL)
#define mod 1000000007
#define  pb push_back
typedef long long ll;
typedef unsigned long long ull;
typedef  long double ld;
typedef vector<int> vi;
typedef vector<bool> vb;
typedef vector<vi >vvi;
typedef vector<ll> vll;
typedef vector<vll >vvll;

ll n, m;
ll maximumdegree=10,large_num=1000000;
vll result(1111111), a(1001), b(1001);



void print_poly(vll& a, ll deg) {

        for (ll i = 0; i < deg; i++)cout << a[i] << "*x^" << i << "+ ";
```

```
        cout << a[deg] << "*x^" << deg << "\n";
}

void build_poly(vll& a, ll st, ll en, vll& result) {

        for (ll i = st; i <= st + en; i++)result[i - st] = a[i];
}

void poly_add(vll& a, vll& b, ll deg, ll bias1, ll bias2, vll& result) {

        for (ll i = 0; i <= deg; i++) {
                result[i] = (i - bias1 >= 0 ? a[i - bias1] : 0) +
                                (i - bias2 >= 0 ? b[i - bias2] : 0);
        }

}

void self_sub(vll& a, vll& b, ll deg) {

        for (ll i = 0; i <= deg; i++)a[i] -= b[i];
}

void poly_mult(vll& a, vll& b, ll deg, vll& result) {

        vll a0(2 * deg + 1), a1(2 * deg + 1);
        vll b0(2 * deg + 1), b1(2 * deg + 1);
        vll  a0b0(2 * deg + 1), a1b1(2 * deg + 1);
        vll  a0a1(2 * deg + 1), b0b1(2 * deg + 1);
        vll  a0b1_a1b0(2 * deg + 1);



        //base cases-->
        if (deg == 0 || deg == 1) {
                if (deg == 0)result[0] = a[0] * b[0];
                else{
                        result[0] = a[0] * b[0];
            result[1] = a[0] * b[1] + a[1] * b[0];
            result[2] = a[1] * b[1];
            }
                return;
        }

        ll deg_0 = deg / 2 - 1, deg_1 = deg - deg / 2;

    build_poly(a, 0, deg_0, a0);
    build_poly(b, 0, deg_0, b0);
    build_poly(a, deg_0 + 1, deg_1, a1);
    build_poly(b, deg_0 + 1, deg_1, b1);

        poly_mult(a0, b0, deg_0, a0b0);
        poly_mult(a1, b1, deg_1, a1b1);
        poly_add(a0, a1, max(deg_0, deg_1), 0, 0, a0a1);
    poly_add(b0, b1, max(deg_0, deg_1), 0, 0, b0b1);
        poly_mult(a0a1, b0b1, max(deg_0, deg_1), a0b1_a1b0);
        self_sub(a0b1_a1b0, a0b0, deg);
    self_sub(a0b1_a1b0, a1b1, deg);
        poly_add(a0b0, a0b1_a1b0, 2ll * deg, 0, deg / 2, result);
        poly_add(result, a1b1, 2ll * deg, 0, 2ll * (deg / 2), result);

}



int main() {
        IOS;
        srand(time(0));

        n = rand() % maximumdegree + 1, m = rand() % maximumdegree + 1;

        for (ll i = 0; i <= n; i++)a[i] = rand() % large_num + 1 ;
        for (ll i = 0; i <= m; i++)b[i] = rand() % large_num + 1;
```

```
        cout << "a polyomial is ---->\n"; print_poly(a, n);
        cout << "\n";
        cout << "b polynomial is ---->\n"; print_poly(b, m);
        cout << "\n";

        poly_mult(a, b, max(n, m), result);

        cout << "resultant polynomial is ---->\n";
        print_poly(result, n * m + 1);
}
```

ALGORITHM ANALYSIS

Assume

$$n = 2^h$$

the recurrence we got from improved algorithm is

$$\begin{aligned} T(n) &= 3T(n/2) + O(n) \\ &= 3T(n/2) + c.n \end{aligned}$$

for some c greater than 0.

$$\begin{aligned} &= 3[3T(n/2^2) + c.n/2] + c.n \\ &= 3^2T(n/2^2) + (1 + 3/2).c.n \\ &\vdots \\ &= 3^hT(n/2^h) + \sum_{j=0}^{h-1} (3/2)^j.c.n \end{aligned}$$

we know ,

$$\begin{aligned} 3^h &= 2^{(log_2 3).h} = (2^h)^{log_2 3} = n^{log_2 3} \\ \sum_{j=0}^{h-1} (3/2)^j &= ((3/2)^h - 1)/((3/2) - 1) \\ &= 2.(n^{log_2 3}/n - 1) \\ &= 2(n^{log_2 3 - 1} - 1) \end{aligned}$$

hence,

$$\begin{aligned} T(n) &= n^{log_2 3} + O(n^{log_2 3}) \\ &= O(n^{log_2 3}) \approx O(n^{1.585}) \end{aligned}$$

Hence Time complexity=

$$O(n^{log_2 3})$$

The space occupied by polymult function of degree n in stack memory is That function recursively calls 3 subproblems of half of actual size one at a time.
so maximum space occupied at a time in stack is

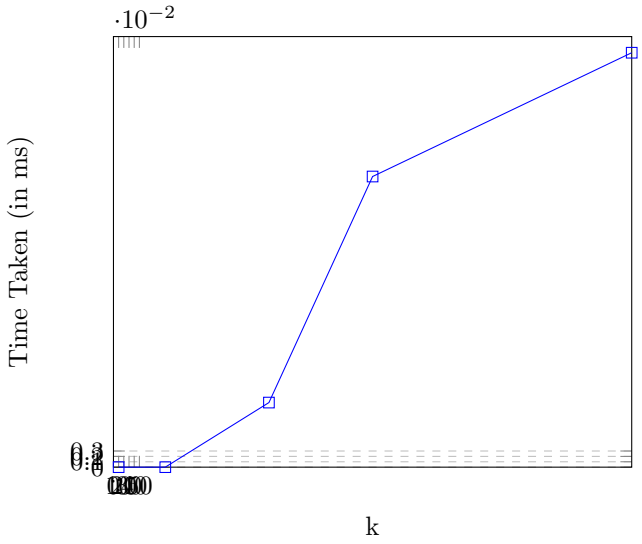
$$\begin{aligned} &O(n) + O(n/2) + O(n/2^2) + ... + O(n/2^{log_2 n}) \\ &= O(2.n.(1 - 1/n)) \\ &= O(n) \end{aligned}$$

Hence Auxillary space=**O(n)**.

TIME AND SPACE GRAPH PLOTTING

TIME ANALYSIS: Following is the graph representing the time complexity of the algorithm.

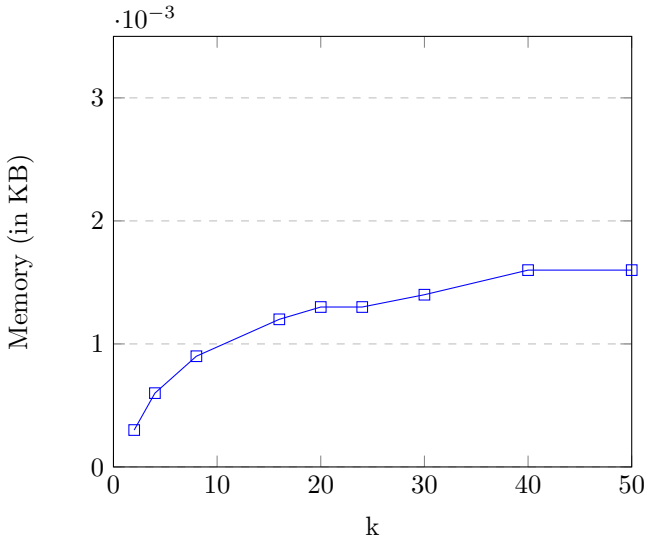
k	Time Taken (in ms)
10	0
100	0
300	0.012
500	0.054
1000	0.077



SPACE ANALYSIS: Following is the graph representing the space complexity of the algorithm.

By the experimental analysis, we found that in case of optimized approach, on increasing the powers of polynomial (increasing the size of memory allocation) the graph is strictly increasing Thus the overall space increases with an increase of memory.Space complexity of this is O(n).

k	memory (in KB)
0	15.24
10	15.24
100	15.31
300	15.32
500	15.368
1000	15.376



CONCLUSION

we can conclude that polynomial multiplication is achieving time complexity of $O(n^{log_2 3})$ and auxillary space of **O(n)**. However ,there exists a more efficient algorithm **(FFT)** which also uses divide and conquer paradigm and achieves **O(n.log(n))** time.

ACKNOWLEDGMENT

We are very much grateful to our Course instructor Mr.Rahul Kala and our mentor, Ms.Tejasvee, who have provided the great opportunity to do this wonderful work on the subject of Design and Analysis of Algorithms, specifically on the algorithmic paradigm of Divide and Conquer.

NOTE

1)One can adjust the maximum degrees of n and m by changing *maximumdegree* variable which is declared a global variable.
2)The submitted code contains comments.One can check what's happening at each step.

REFERENCES

1. Introduction to Algorithms by Cormen,Charles, Rivest and Stein.
   https://web.ist.utl.pt/ fabio.ferreira/material/asa
2. https://en.wikipedia.org/wiki/Karatsuba$_a lgorithm$