DAA Assignment No. : 6
Minimizing the sum of roots of a given polynomial

Lovely Digra
IIT2017090

Abhishek Vishwakarma
IIT2017091

Anuj Raj
IIT2017092

***Abstract*—This paper introduces us to an algorithm to find a polynomial by rearranging the coefficients of the given polynomial such that sum of its roots is minimum. We solve the problem by using fundamental theorem of algebra and Vieta's formulas. While designing our problem we calculated the time complexity for the algorithm which comes to be O(n) and plotted graphs and found the nature of graph is straight line.**

I. Introduction

Any general polynomial of degree n:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + ...$$

Fundamental theorem of Algebra says that, in a n-degree polynomial, total number of roots is n,real or complex not necessarily distinct. Vieta's formulas relate the polynomial's coefficients a_k to signed sums as follows :

$$x_1 + x_2 + ... + x_n = -\frac{a_{n-1}}{a_n}$$

For further discussion a_n and a_{n-1} is referred as a and b respectively. We are given an array of coefficients of the polynomial. If the polynomial is of degree n then the array contains n+1 elements one for constant. Except the first element, any element can be a zero. Also length of the array is greater than 1.

II. Algorithm Design and Explanation

[H]

The problem of minimizing the sum of roots is converted into maximizing the ratio $\frac{b}{a}$. For this, b should be the maximum value in the array and a should be the minimum value and also $a.b > 0$. So the coefficients of array is divided in three set : positive coefficients, negative coefficients and zero coefficients. This type of grouping will result in the following four cases. a and b are swapped according to the following cases. Note that the non-zero a_n constraint is followed.
Case 1: When the number of positive coefficients and the number of negative coefficients both are greater than or equal to 2. In this case, we will find a maximum and minimum from positive elements and from negative elements also and we will check which is smallest among $-$(maxPos)/(minPos) and $-$(abs(maxNeg))/ (abs(minNeg)) and print the answer after swapping accordingly.

Case 2: When the number of positive coefficients is greater than equal to 2 but the number of negative coefficients is less than 2. In this case, we will consider the case of the maximum of positive and minimum of positive elements only. Because if we picked up one from positive elements and the other from negative elements, the result of -b/a will be a positive value which is not minimum (as we require a large negative value).

Case 3: When the number of negative coefficients is greater than equal to 2 but the number of positive coefficients is less than 2. In this case, we will consider the case of the maximum of negative and minimum of negative elements only. Because if we picked up one from positive elements and the other from negative elements, the result of -b/a will be a positive value which is not minimum (as we require a large negative value).

Case 4: When both the counts are less than or equal to 1. In this case whatever value of a and b are selected the product will be negative or zero , so the maximum possible value of the ratio can zero. Thus this case can be further divided in two sub cases. case 4.1: Atleast one coefficient is zero: If there is a zero coefficient in the array , b is swapped with zero. for example :
input: 1 -2 0 0
output: -2 0 1 0
i.e. if given polynomial is $x^3 - 2x^2$
output polynomial is $-2x^3 + x$ with sum zero.
case 4.2:There are two coefficient. None of them is zero. The answer is found by simple comparision.

III. Experimental Study

For the experimental analysis of this algorithm, we will take no.of coefficients of a polynimial(n) as an input and plot a graph of no. of coefficients vs execution time. A .dat file is generated. For graph refer Fig 1.

IV. Time Complexity and Discussion

The General Time Complexity of the above algorithm comes to be : O(n), where n is number of coefficients of the given polynomial.

Best Time Complexity: Omega(n) , when all the coefficients are positive or negative and the maximum and minimum value among them are first and second term respectively.

Worst Time Complexity: O(n) , when all the coefficients are positive or negative and the maximum and minimum value among them are nth and (n-1)th term respectively.

Space Complexity: O(n), as the number of coefficients is n . Therefore, nth size array or vector is used to store these coefficients.

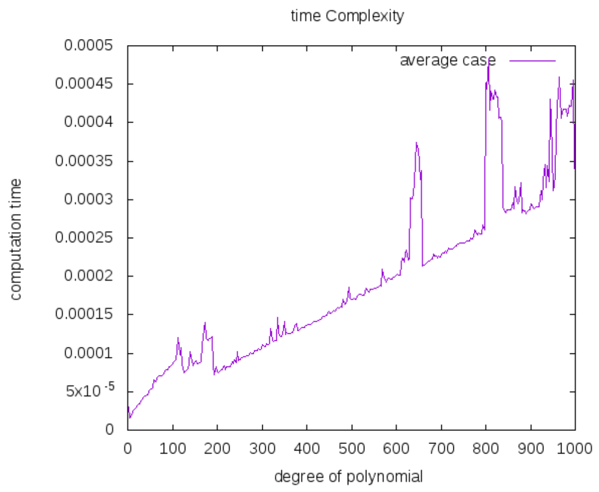

Fig. 1: Fig 1

V. CONCLUSION

We designed an algorithm to find a polynomial by rearranging the coefficients of the given polynomial such that sum of its roots is minimum. We also analyzed the algorithm that resulted in general time complexity of O(n) and plotted the graph number of coefficients vs time elapsed and found that the nature of graph is straight line.

Algorithm 1 PseudoCode for Approach

```
procedure $getMinSum(intarr[], intn)$
    vector<int> res;
    //Store coefficients of final polynomial
    vector<int> pos;
    //Store indices of positive no
    vector<int> neg;
    //Store indices of negative no
    vector<int> zero;
    //Store indices of occurrence of zero
    for all $i = 0$ to n do
        if $A[i] < 0$ then
            neg.push_back(i);
        else
            pos.push_back(i);
    if $pos.size() >= 2$ && $neg.size() >= 2$ then
        findRatio_of_both();
    else if $pos.size() >= 2$ then
        findRatio_of_pos();
    else if $neg.size() >= 2$ then
        findRatio_of_neg();
    else
        findRatio();
    //Print coefficients of final polynomial
    for all $i = 0$ to res.size() do
        $Print(res[i])$
```

References

[1] https://www.geeksforgeeks.org/minimize-sum-roots-given-p/
[2] https://en.wikipedia.org/wiki/Vieta
[3] https://www.mathsisfun.com/algebra/polynomials-sums-prod

Algorithm 2 Case: 1

```
//For pos.size() >= 2 && neg.size() >= 2
procedure $findRatio\_of\_both()$
    $posMax$ ← maximium positive number
    $posMin$ ← minimum positive number
    $posMaxIdx$ ← index of max +ve number
    $posMinIdx$ ← index of min +ve number
    $negMax$ ← maximium negative number
    $negMin$ ← minimum negative number
    $negMaxIdx$ ← index of max -ve number
    $negMinIdx$ ← index of min -ve number
    //Store value of ratios
    $posVal$ ← -1.0*(posMax / posMin);
    $negVal$ ← -1.0*(negMax /negMin);

    //Store coefficients of final polynomial in res
    if $posVal < negVal$ then
        res.push_back(arr[posMinIdx]);
        res.push_back(arr[posMaxIdx]);
        for all $i = 0$ to n do
            if $i \neq posMinIdx \&\& \imath \neq posMaxIdx$ then
                res.push_back(arr[i]);
    else
        res.push_back(arr[negMinIdx]);
        res.push_back(arr[negMaxIdx]);
        for all $i = 0$ to n do
            if $i \neq negMinIdx \&\& \imath \neq negMaxIdx$ then
                res.push_back(arr[i]);
```

Algorithm 3 Case: 2

```
// For pos.size() >= 2
Input: vector res,
vector pos,
int arr[],
int n
procedure $findRatio\_of\_pos()$
    $posMax$ ← maximium positive number
    $posMin$ ← minimum positive number
    $posMaxIdx$ ← index of max +ve number
    $posMinIdx$ ← index of min +ve number
    res.push_back(arr[posMinIdx]);
    res.push_back(arr[posMaxIdx]);
    for all $i = 0$ to n do
        if $i \neq posMinIdx$ && $i \neq posMaxIdx$ then
            res.push_back(arr[i]);
```

Algorithm 4 Case: 3

```
//For neg.size() >= 2
Input: vector res,
vector neg,
int arr[],
int n
procedure $findRatio\_of\_neg()$
    $negMax$ ← maximium negative number
    $negMin$ ← minimum negative number
    $negMaxIdx$ ← index of max -ve number
    $negMinIdx$ ← index of min -ve number
    res.push_back(arr[negMinIdx]);
    res.push_back(arr[negMaxIdx]);
    for all $i = 0$ to n do
        if $i \neq negMinIdx$ && $i \neq negMaxIdx$ then
            res.push_back(arr[i]);
```

Algorithm 5 Case: 4

```
//For pos.size <=1 && neg.size <=1
Input: vector res,
vector zero,
int arr[],
int n
procedure $findRatio()$
    if $zero.size() == 0$ then
        //Store value of ratios
        $val1$ ← -1.0*(arr[1] / arr[0]);
        $val2$ ← -1.0*(arr[0] / arr[1]);
        if $val1 <= val2$ then
            res.push_back(arr[0]);
            res.push_back(arr[1]);
        else
            res.push_back(arr[1]);
            res.push_back(arr[0]);
    else
        res.push_back(arr[0]); //rest as it is
        res.push_back(0); // b = 0
        for all $i = 0$ to n do
            if $i \neq zero[0]$ then
                res.push_back(arr[i]);
```