Algorithm Class Assignment 2 Divide & Conquer

Question 1

Algorithm Design

Question's Main Idea:

- By finding the biggest sum in the number array,
  we can find the needed range for the sub-array

Algorithm Solving:

- Traverse the data array from left to right and record a sum in [0, x]
- If the new sum result is bigger than the older record,
  replace the old result to the new one
- Store the current value of 'x' (targeti), as the traverse
  operation finished, the meaning of 'targeti' is that if the
  biggest sub-array to be [0, targeti], the right boundery must be 'targeti'
- Traverse the data array from right to left and get a similar result of 'targetj'
- If targeti >= targetj, the required sub-array must be [targetj, targeti]
- If targeti < targetj, the required sub-array must in [0, targeti],
  [targeti, targetj], [targetj, size - 1], with the same approach mentioned above,
  find the maximum value between these ranges and get the final result

Code Implementation

```
// Maximum Sub-array Sum Algorithm
```

```
// Using Divide & Conquer Method
// Data Input Assume To Be An Integer Array

#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int i;
        int j;
        int targeti;
        int targetj;
        int size = nums.size();
        int sum = (-999999);
        int temp = 0;

        for (i = 0; i < size; i++) {
            temp = temp + nums[i];

            if (temp > sum) {
                sum = temp;
                targeti = i;
            }
        }

        temp = 0;
        sum = (-999999);

        for (i = size - 1; i >= 0; i--) {
            temp = temp + nums[i];

            if (temp > sum) {
                sum = temp;
                targetj = i;
            }
        }

        if (targeti >= targetj) {
            int result = 0;

            for (j = targetj; j <= targeti; j++) {
                result = result + nums[j];
            }

            return result;
        }

        temp = 0;
        int targetii;
        sum = (-999999);

        for (j = targeti; j >= 0; j--) {
            temp += nums[j];

            if (temp > sum) {
                sum = temp;
```

```
                    targetii = j;
                }
            }

            int result1 = 0;

            for (j = targetii; j <= targeti; j++) {
                result1 = result1 + nums[j];
            }

            temp = 0;
            int targetjj;
            sum = (-999999);

            for (j = targetj; j < size; j++) {
                temp = temp + nums[j];

                if (temp > sum) {
                    sum = temp;
                    targetjj = j;
                }
            }

            int result2 = 0;

            for (j = targetj; j <= targetjj; j++) {
                result2 = result2 + nums[j];
            }

            vector<int> nums1(nums.begin() + targeti, nums.begin() + targetj);

            int result3 = maxSubArray(nums1);

            return max(max(result1, result2), result3);
        }
};

int main() {
    vector<int> integerArray;
    int number;
    int answer;
    Solution s;

    while (cin >> number && number != 0) {
        integerArray.push_back(number);
    }

    /*
    for(int i = 0; i < integerArray.size(); i++) {
        cout << integerArray[i] << endl;
    }
    */

    answer = s.maxSubArray(integerArray);

    cout << answer;
}
```

Question 2

Algorithm Design

<u>Question's Main Idea:</u>

- Find the smallest distance between any arbitrary two points

<u>Algorithm Solving:</u>

- Sort all of the points in the two-dimensional space by the x-axis value
- Keep dividing the points into half and establishing two subgroups
- As soon as the subgroup has the elements smaller or equal to three, find all distance between each point using method of exhaustion
- With the smaller distance found in the left subgroup & right subgroup, compare left smaller and the right smaller in order to find the minimum.
- Check the cross-domain points' distance within two times the minimum with the divide line to be in the middle
- Compare between the smaller cross-domain point distance and the minimum to find the merge minimum and combine these two calculated subgroups

Code Implementation

```cpp
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <cmath>
#define MAX 10000
using namespace std;

// Class 'Point'
class Point {
public:
    // 'x' value of the point
    double x;
    // 'y' value of the point
    double y;

    // Calculate the distance between two points
    double Distance(Point point) {
        double result;
        result = sqrt((point.x - x)*(point.x - x) + (point.y - y)*(point.y - y
        return result;
    }
};

// Initial an array with Point attribute
Point point[MAX];

//  Comparing function for C++ sort using x-axis of the point
bool compareXaxis (Point a, Point b) {
    if (a.x < b.x) {
        return true;
```

```
            return true;
        }
        else {
            return false;
        }
}

// Comparing function for C++ sort using y-axis of the point
bool compareYaxis (Point a, Point b) {
    if (a.y < b.y) {
        return true;
    }
    else {
        return false;
    }
}

// Algorithm 'Divide & Conquer'
double divideNconquer(int L, int R) {
    int i;
    int j;

    // Only one point
    if (L >= R) {
        return MAX;
    }
    // Two or three points
    else if (R - L < 3) {
        double d = MAX;

        // Exhaustion method
        for (i = L; i < R; i++) {
            for (j = i + 1; j <= R; j++) {
                d = min(d, point[i].Distance(point[j]));
            }
        }

        return d;
    }

    int M = (L + R) / 2;

    // Find the smallest distance for each subgroup
    double d = min(divideNconquer(L, M), divideNconquer(M + 1, R));

    if (d == 0) {
        return 0;
    }

    int n = 0;
    Point strip[MAX];

    // Find those points closer to mid (with the distance smaller than current
    // Left side
    for (i = M; i >= L && point[M].x - point[i].x < d; i--) {
        strip[n++] = point[i];
    }
    // Right side
    for (i = M + 1; i <= R && point[i].x - point[M].x < d; i++) {
        strip[n++] = point[i];
```

```
            strip[n++] = point[i];
        }

        // Sort by y-axis
        sort(strip, strip + n, compareYaxis);

        // Find the smallest distance across subgroups
        for (i = 0; i < n; i++) {
            for (j = 1; j <= 3 && i + j < n; j++) {
                d = min(d, strip[i].Distance(strip[i + j]));
            }
        }

        return d;
    }

    int main()
    {
        int n;
        int i;

        // Load in all the points needed
        while (cin >> n && n > 0) {
            // Load in the x & y value of the point
            for (i = 0; i < n; i++) {
                cin >> point[i].x >> point[i].y;
            }

            // Sort by x-axis
            sort(point, point + n, compareXaxis);

            // Execute the algorithm
            double answer = divideNconquer(0, n - 1);

            // Output the answer
            if (answer == 10000) {
                cout << "INFINITY" << endl;
            }
            else {
                cout << fixed;
                cout << setprecision(4);
                cout << answer << endl;
            }
        }

        return 0;
    }
```

Question 3

Textbook 2.6

- Algorithm

1. Split the given data into three equal sub-groups and get two divide index i & j
2. Select the sub-group by comparing the required number with two index

  - smaller than i: range between n (smallest number) and i
  - between i & j: range between i and j
  - bigger than j: range between j and m (biggest number)
3. Recursively execute the function (Split group and select sub-group)
4. Check the required value is whether existed in the data array or not

- **Time Complexity = O(logn)**
  - For the question data set to be a sorted array, the implementation will be similar to binary search yet divide the big problem into 3 smaller subproblem. Thus, we will perform total log3n operations and get the magnitude of logn
- **Space Complexity = O(logn)**
  - Due to the ‘call stack’ operation in the implementation, we will need logn space to perform it

Textbook 2.7

- Algorithm

1. Declare ‘LEFT’ and ‘RIGHT’ varibles which will mark the extreme indices of the array
2. ‘LEFT’ will be assigned to 0 and ‘RIGHT’ will be assigned to (n - 1)
3. Find MID = (LEFT + RIGHT) / 2
4. Call ‘mergeSort’ function on (LEFT, MID) and (MID + 1, REAR)
5. Recursively call the function until LEFT < RIGHT
6. Merge the subproblems till the whole list has been sorted
7. Return the last element of the sorted array and we will find the biggest number

- **Time Complexity = nlog2n = O(nlogn)**
  - In every iteration, we are dividing the big problem into 2 smaller subproblems. Hence this will perform log2n operations and has to be done for n iteration, which results in nlog2n operations total
- **Space Complexity = O(n)**
  - ‘n’ auxiliary space is required in implementation as all the elements are copied into an secondary array

Textbook 2.13

- Algorithm

1. Split the big problem into three sub-groups and find two divide index i & j
   - Divide index ‘i’ to be one-third of the data array
   - Divide index ‘j’ to be two-third of the data array
2. Call ‘mergeSort’ function on (BEGIN, i), (i, j), (j, END)
3. Recursively call the function until those elements in the subgroups are in the correct order

4. Merge the subproblems till the whole list has been in the sorted order
5. Gain the sorted result for our input data

- **Time Complexity = nlog3n = O(nlogn)**
  - In every iteration, we are dividing the big problem into 3 smaller subproblems. Hence this will perform log3n operations and has to be done for n iteration, which results in nlog3n operations total
- **Space Complexity = O(n)**
  - ‘n’ auxiliary space is required in implementation as all the elements are copied into an secondary array

tags: `Algorithm Class`