# Question 4

*// We can have multiple approaches for finding the pivot element. I have used the approach in which the first element of the partition is the pivot element always.*

#include <iostream>

#include <vector>

using namespace std;

int *partition*(*vector*<int> &*arr*, int *start*, int *end*)

{

    int pivot = *arr[start]*; *// Pivot is our first element of the array*

*// First we count how many numbers are less than the pivot in the array*

    int count = 0;

    for (int i = *start*; i <= *end*; i++)

    {

        if (*arr[*i*]* < pivot)

        {

            count++;

        }

    }

    int pivotIndex = *start* + count;

*swap*(*arr[start]*, *arr[*pivotIndex*]*);

    int i = *start*;

    int j = *end*;

    while (i < pivotIndex && j > pivotIndex)

    {

        while (*arr[*i*]* < pivot) *// No need to swap case1*

        {

            i++;

        }

        while (*arr[*j*]* > pivot) *// No need to swap case2*

        {

            j--;

        }

        if (i < pivotIndex && j > pivotIndex)

        {

*swap*(*arr[*i*]*, *arr[*j*]*);

        }

    }

    return pivotIndex;

}

void *quickSort*(*vector*<int> &*arr*, int *start*, int *end*)

{

*// Base Case*

    if (*start* >= *end*)

    {

        return;

    }

*// Making a partition*

    int p = *partition*(*arr*, *start*, *end*);

*// Recursive calls for sorting the left and right partitions*

*quickSort*(*arr*, *start*, p - 1);

*quickSort*(*arr*, p + 1, *end*);

    return;

}

*/\**

*Best Case and Worst case complexity of Quick sort??*

*=> Best Case O(nlogn)*

*When the first element happens to be close to the median value. So the array gets divided into roughly equal halves*

*Example: [5, 2, 8, 1, 9, 3, 7] - first element 5 is near the middle value*

*[5, 2, 8, 1, 9, 3, 7]  pivot = 5*

*→ [2, 1, 3] | 5 | [8, 9, 7]  ← Balanced!*

*The recursion tree has log n levels (because we're dividing by 2 each time)*

*At each level, we do O(n) work (partitioning all elements)*

*Total: log n levels × O(n) work per level = O(n log n)*

*=> Worst Case O(n^2)*

*When the pivot is the smallest or the largest element. i.e the array is sorted or reverse sorted, or when all the elements of the array are identical.*

*[1, 2, 3, 4, 5]  pivot = 1 (smallest!)*

*→ [] | 1 | [2, 3, 4, 5]  ← Extremely unbalanced!*

*Then: [2, 3, 4, 5]  pivot = 2*

*→ [] | 2 | [3, 4, 5]  ← Still unbalanced!*

*Then: [3, 4, 5]  pivot = 3*

*→ [] | 3 | [4, 5]*

*Then: [4, 5]  pivot = 4*

*→ [] | 4 | [5]*

*5 levels for 5 elements = O(n²)*

*\*/*

int *main*()

{

    cout *<<* *endl*

*<<* " \_\_  \_\_         \_\_   \_\_    \_\_\_  ||   \_\_  \_\_         \_\_  \_\_\_ \_\_\_ \_\_   ||   \_\_  \_\_         \_\_   \_\_   \_\_    " *<<* *endl*

*<<* "  \_)  \_)|  \_\_  /  \\ |\_  /|   /  ||    \_)  \_)|  \_\_  /  \\   /   /  \_)  ||    \_)  \_)|  \_\_  /  \\ /\_\_  (\_\_) /|" *<<* *endl*

*<<* " /\_\_ \_\_)|(     \\\_\_/ \_\_) \_|\_ /   ||   /\_\_ \_\_)|(     \\\_\_/  /   /  /\_\_  ||   /\_\_ \_\_)|(     \\\_\_/ \\\_\_) (\_\_) \_|\_" *<<* *endl*

*<<* *endl*;

*vector*<int> arr1 = {5, 2, 8, 1, 9, 3, 7};

*vector*<int> arr2 = {1, 2, 3, 4, 5};

    for (auto i : arr1)

    {

        cout *<<* i *<<* "  ";

    }

    cout *<<* *endl*

*<<* "(Best Case) -> Sorting this array in which median value i.e 5 becomes the pivot" *<<* *endl*;

*quickSort*(arr1, 0, arr1.*size*() - 1);

    for (auto i : arr1)

    {

        cout *<<* i *<<* "  ";

    }

    cout *<<* *endl*

*<<* *endl*;

    for (auto i : arr2)

    {

        cout *<<* i *<<* "  ";

    }

    cout *<<* *endl*

*<<* "(Worst Case) -> Sorting this array which is already sorted." *<<* *endl*;

*quickSort*(arr1, 0, arr2.*size*() - 1);

    for (auto i : arr2)

    {

        cout *<<* i *<<* "  ";

    }

    cout *<<* *endl*

*<<* *endl*;

}

# Output

A screenshot of a computer program

AI-generated content may be incorrect.

# Question 5

*/\**

*One dimensional???*

*-> This means we are working on a single number line , therefore we do not have points here in 2D space, in which we have x and y coordinates both.*

*\*/*

#include <iostream>

#include <vector>

using namespace std;

*// Creating a class here as we have to return the cloesest distance + the pair of points. So we will return the whole object instead.*

class *closestPair*

{

public:

    double number1;

    double number2;

    double distance;

*closestPair*() {};

*closestPair*(double *n1*, double *n2*, double *d*) : *number1*(*n1*), *number2*(*n2*), *distance*(*d*) {}

};

*// This fnction makes every negative distance positive , as distance is always positive.*

double *findAbsoluteValue*(double *x*)

{

    if (*x* < 0)

    {

        return -*x*; *// E.g if x is -5, it does -(-5) so it becomes 5.*

    }

    return *x*;

}

*// Asal function ye hai*

*closestPair* *findClosestPair*(*vector*<double> &*numberList*, int *start*, int *end*)

{

    int size = *end* - *start*;

    if (size <= 1)

    {

        return *closestPair*(0, 0, 1e308); *// Base case 1*

    }

    if (size == 2)

    {

        double dist = *findAbsoluteValue*(*numberList[start* + 1*]* - *numberList[start]*);

        return *closestPair*(*numberList[start]*, *numberList[start* + 1*]*, dist); *// Base Case 2*

    }

    int middle = *start* + (*end* - *start*) / 2;

*// Recursive Calls for left and right halves*

*closestPair* leftResult = *findClosestPair*(*numberList*, *start*, middle);

*closestPair* rightResult = *findClosestPair*(*numberList*, middle, *end*);

*// Find now which half has the smaller distance.*

*closestPair* currentBest;

    if (leftResult.distance <= rightResult.distance)

    {

        currentBest *=* leftResult; *// Yahan pr deep copy shallow copy wala chakkar nhi hosakta because we don't have any array or a pointer so that's why no need of a copy constructor.*

    }

    else

    {

        currentBest *=* rightResult;

    }

*/\*Now we have current best. But suppose this was the list {1,9,10,20}*

*Left half: [1, 9] → closest pair is (1, 9) with distance = 8*

*Right half: [10, 20] → closest pair is (10, 20) with distance = 10*

*So current\_best.distance = 8*

*But wait! What about the pair (9, 10)?*

*9 is in the left half*

*10 is in the right half*

*Distance = 1 ← This is actually the closest pair!*

*=> So now we have to find the cross-half closest pair possibilities.*

*\*/*

    for (int i = *start*; i < middle; i++)

    { *// Every number in left half*

        for (int j = middle; j < *end*; j++)

        { *// Every number in right half*

            double cross\_boundary\_distance = *findAbsoluteValue*(*numberList[*j*]* - *numberList[*i*]*);

            if (cross\_boundary\_distance < currentBest.distance)

            {

*// Found a better pair that crosses the boundary!*

                currentBest.distance = cross\_boundary\_distance;

                currentBest.number1 = *numberList[*i*]*;

                currentBest.number2 = *numberList[*j*]*;

            }

        }

    }

    return currentBest;

}

int *main*()

{

    cout *<<* *endl*

*<<* " \_\_  \_\_         \_\_   \_\_    \_\_\_  ||   \_\_  \_\_         \_\_  \_\_\_ \_\_\_ \_\_   ||   \_\_  \_\_         \_\_   \_\_   \_\_    " *<<* *endl*

*<<* "  \_)  \_)|  \_\_  /  \\ |\_  /|   /  ||    \_)  \_)|  \_\_  /  \\   /   /  \_)  ||    \_)  \_)|  \_\_  /  \\ /\_\_  (\_\_) /|" *<<* *endl*

*<<* " /\_\_ \_\_)|(     \\\_\_/ \_\_) \_|\_ /   ||   /\_\_ \_\_)|(     \\\_\_/  /   /  /\_\_  ||   /\_\_ \_\_)|(     \\\_\_/ \\\_\_) (\_\_) \_|\_" *<<* *endl*

*<<* *endl*;

*vector*<double> list = {2.5, -3, 4, 10, 6};

*closestPair* c = *findClosestPair*(list, 0, list.*size*() - 1);

    cout *<<* "distance: " *<<* c.distance *<<* *endl*

*<<* "Pair: ( " *<<* c.number1 *<<* "," *<<* c.number2 *<<* " )" *<<* *endl*;

}

# Output

A screenshot of a computer

AI-generated content may be incorrect.