****

**NAME: -** Darji Akshatkumar Hiteshbhai

**RollNo: -** 23MCD001

**Branch: -** M.tech-CSE**(Data Science)**

**Subject: -** Complexity Theory & Algorithms

**Practical-3**

**Aim:** Randomized quick sort for the input size 10000, 50000 and 100000 for Ascending, Descending & Random order array. Plot the chart of the output data and do the analysis which algorithm is best and justify your reason.

**Code for Randomized Quick Sort-**

#include <bits/stdc++.h>

using namespace std;

int partition\_low(vector<int> &array, int low, int high)

{

    int pivot = array[low];

    int start = low + 1;

    int end = high;

    while (true)

    {

        while (start <= end && array[end] >= pivot)

        {

            end = end - 1;

        }

        while (start <= end && array[start] <= pivot)

        {

            start = start + 1;

        }

        if (start <= end)

        {

            swap(array[start], array[end]);

        }

        else

        {

            break;

        }

    }

    swap(array[low], array[end]);

    return end;

}

int partition\_high(vector<int> &array, int low, int high)

{

    int pivot = array[high];

    int start = low;

    int end = high - 1;

    while (true)

    {

        while (start <= end && array[start] <= pivot)

        {

            start = start + 1;

        }

        while (start <= end && array[end] >= pivot)

        {

            end = end - 1;

        }

        if (start <= end)

        {

            swap(array[start], array[end]);

        }

        else

        {

            break;

        }

    }

    swap(array[start], array[high]);

    return start;

}

int partition\_mid(vector<int> &array, int low, int high)

{

    int mid = (low + high) / 2;

    int pivot = array[mid];

    int start = low;

    int end = high;

    while (true)

    {

        while (array[start] < pivot)

        {

            start = start + 1;

        }

        while (array[end] > pivot)

        {

            end = end - 1;

        }

        if (start >= end)

        {

            return end;

        }

        swap(array[start], array[end]);

        start = start + 1;

        end = end - 1;

    }

}

int partition\_random(vector<int> &array, int low, int high)

{

    int pivot\_index = low + rand() % (high - low + 1);

    int pivot = array[pivot\_index];

    swap(array[pivot\_index], array[high]);

    int start = low;

    int end = high - 1;

    while (true)

    {

        while (start <= end && array[start] <= pivot)

        {

            start = start + 1;

        }

        while (start <= end && array[end] >= pivot)

        {

            end = end - 1;

        }

        if (start <= end)

        {

            swap(array[start], array[end]);

        }

        else

        {

            break;

        }

    }

    swap(array[start], array[high]);

    return start;

}

void quick\_sort(vector<int> &array, int start, int end, int pivot\_choice)

{

    while (start < end)

    {

        int idx;

        switch (pivot\_choice)

        {

        case 0: // Low

            idx = partition\_low(array, start, end);

            break;

        case 1: // High

            idx = partition\_high(array, start, end);

            break;

        case 2: // Mid

            idx = partition\_mid(array, start, end);

            break;

        case 3: // Random

            idx = partition\_random(array, start, end);

            break;

        }

        if (idx - start < end - idx)

        {

            quick\_sort(array, start, idx - 1, pivot\_choice);

            start = idx + 1;

        }

        else

        {

            quick\_sort(array, idx + 1, end, pivot\_choice);

            end = idx - 1;

        }

    }

}

int main()

{

    srand(static\_cast<unsigned int>(time(nullptr)));

    vector<int> sizes = {10000, 50000, 100000};

    for (int size : sizes)

    {

        vector<int> ascending(size);

        vector<int> descending(size);

        vector<int> random(size);

        // Generate ascending order array

        for (int i = 0; i < size; ++i)

        {

            ascending[i] = i + 1;

        }

        // Generate descending order array

        for (int i = 0; i < size; ++i)

        {

            descending[i] = size - i;

        }

        // Generate random order array

        for (int i = 0; i < size; ++i)

        {

            random[i] = rand() % size + 1;

        }

        // Sort with different pivot choices

        for (int pivot\_choice = 0; pivot\_choice < 4; ++pivot\_choice)

        {

            vector<int> ascending\_copy = ascending;

            vector<int> descending\_copy = descending;

            vector<int> random\_copy = random;

            // Sort ascending order array

            clock\_t start\_time = clock();

            quick\_sort(ascending\_copy, 0, ascending\_copy.size() - 1, pivot\_choice);

            clock\_t end\_time = clock();

            double elapsed\_time = double(end\_time - start\_time) / CLOCKS\_PER\_SEC;

            cout << "Array Size: " << size << ", Pivot Choice: ";

            switch (pivot\_choice)

            {

            case 0:

                cout << "Low";

                break;

            case 1:

                cout << "High";

                break;

            case 2:

                cout << "Mid";

                break;

            case 3:

                cout << "Random";

                break;

            }

            cout << "\nAscending Order, Time Taken: " << elapsed\_time << " seconds" << endl;

            // Print the first 200 sorted elements

            cout << "First 200 Sorted Elements: ";

            for (int i = 0; i < min(200, size); ++i)

            {

                cout << ascending\_copy[i] << " ";

            }

            cout << "\n\n";

            // Sort descending order array

            start\_time = clock();

            quick\_sort(descending\_copy, 0, descending\_copy.size() - 1, pivot\_choice);

            end\_time = clock();

            elapsed\_time = double(end\_time - start\_time) / CLOCKS\_PER\_SEC;

            cout << "Array Size: " << size << ", Pivot Choice: ";

            switch (pivot\_choice)

            {

            case 0:

                cout << "Low";

                break;

            case 1:

                cout << "High";

                break;

            case 2:

                cout << "Mid";

                break;

            case 3:

                cout << "Random";

                break;

            }

            cout << "\nDescending Order, Time Taken: " << elapsed\_time << " seconds" << endl;

            // Print the first 200 sorted elements

            cout << "First 200 Sorted Elements: ";

            for (int i = 0; i < min(200, size); ++i)

            {

                cout << descending\_copy[i] << " ";

            }

            cout << "\n\n";

            // Sort random order array

            start\_time = clock();

            quick\_sort(random\_copy, 0, random\_copy.size() - 1, pivot\_choice);

            end\_time = clock();

            elapsed\_time = double(end\_time - start\_time) / CLOCKS\_PER\_SEC;

            cout << "Array Size: " << size << ", Pivot Choice: ";

            switch (pivot\_choice)

            {

            case 0:

                cout << "Low";

                break;

            case 1:

                cout << "High";

                break;

            case 2:

                cout << "Mid";

                break;

            case 3:

                cout << "Random";

                break;

            }

            cout << "\nRandom Order, Time Taken: " << elapsed\_time << " seconds" << endl;

            // Print the first 200 sorted elements

            cout << "First 200 Sorted Elements: ";

            for (int i = 0; i < min(200, size); ++i)

            {

                cout << random\_copy[i] << " ";

            }

            cout << "\n\n";

        }

    }

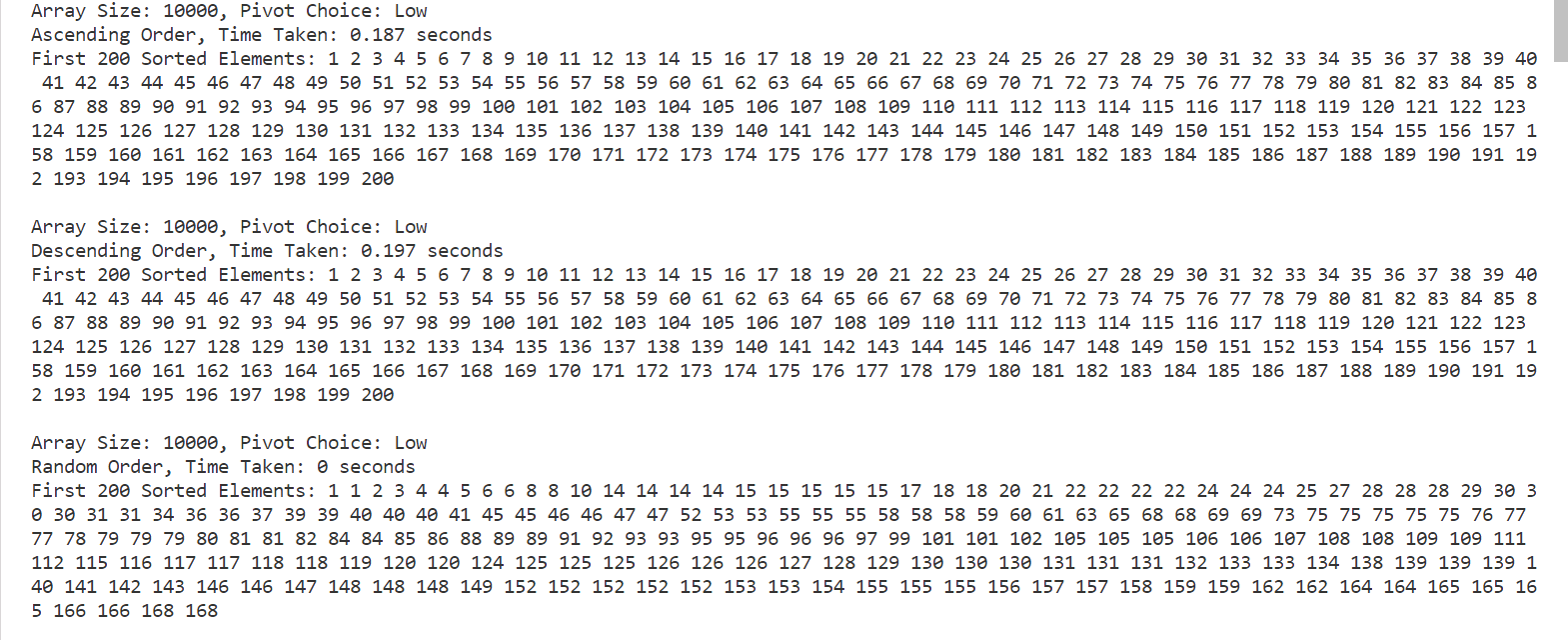
    return 0;

}

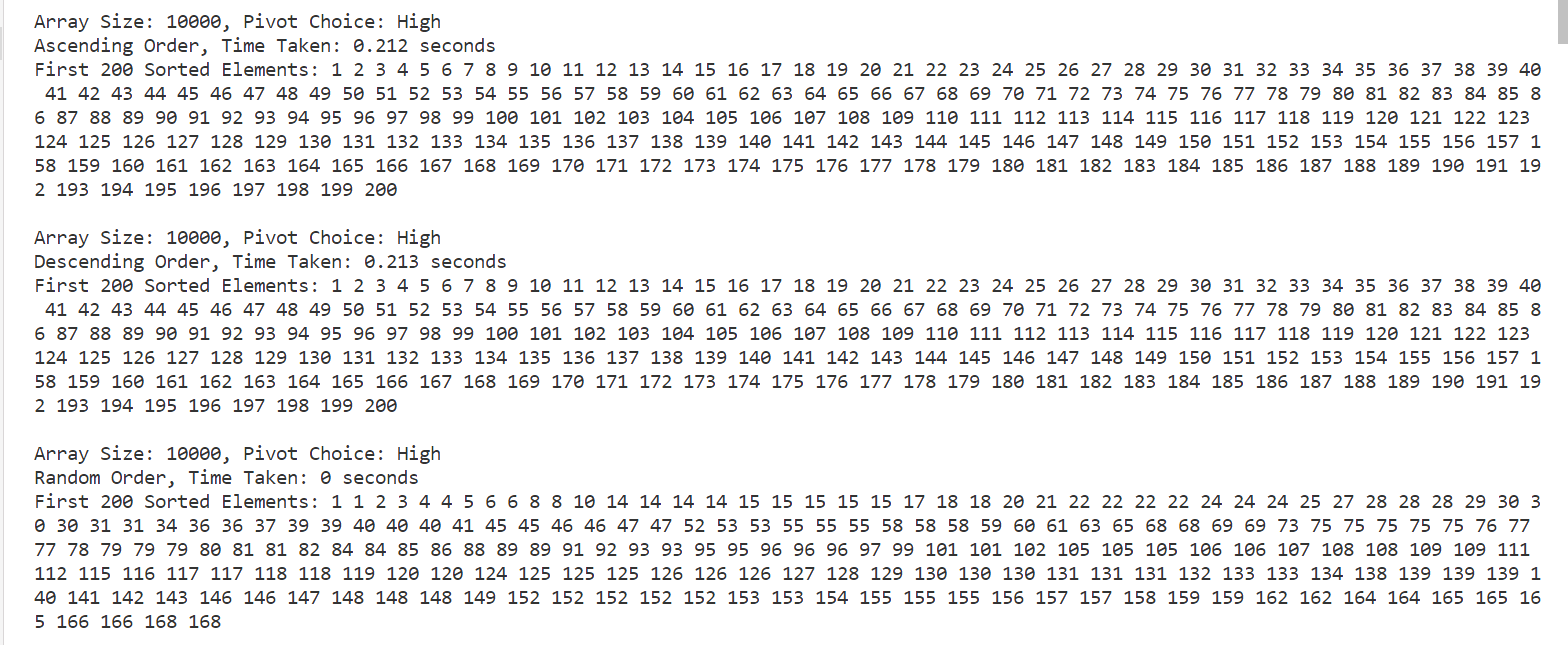
* **Output**

**FOR 10000 Data**

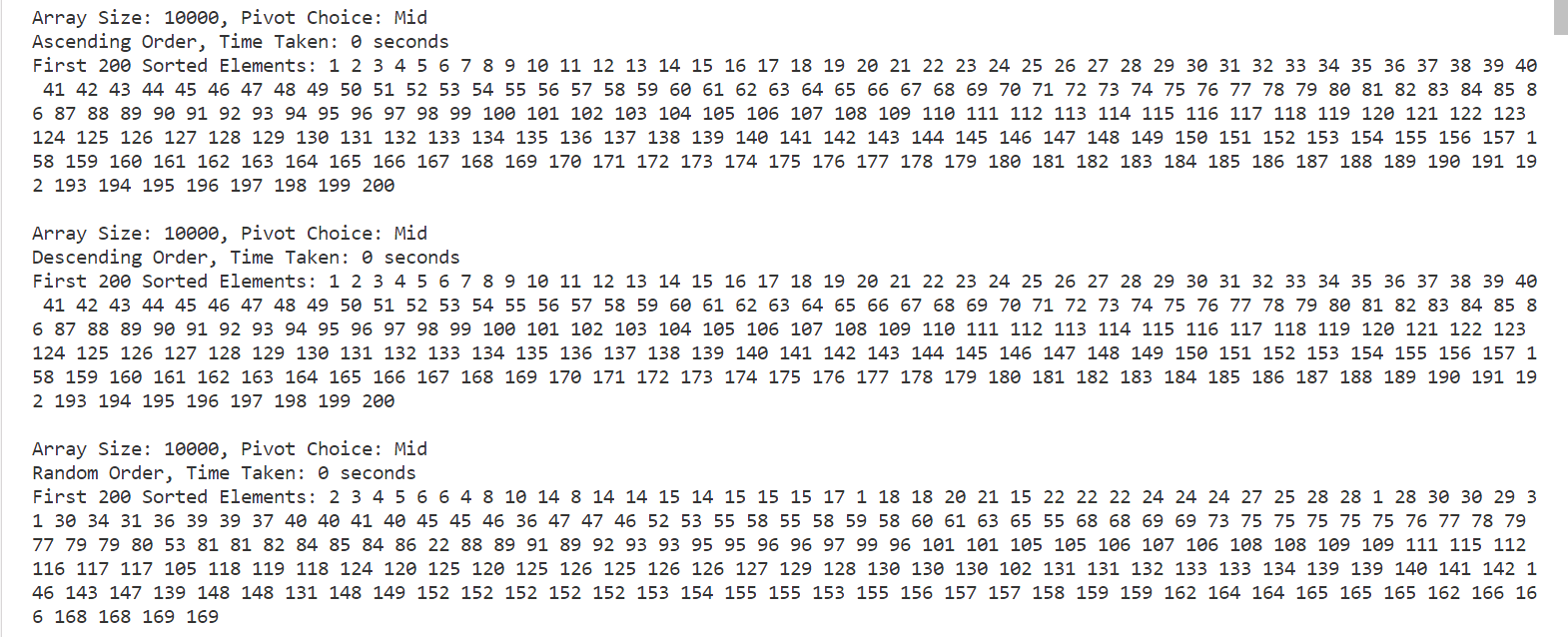
**Pivot Choice LOW**



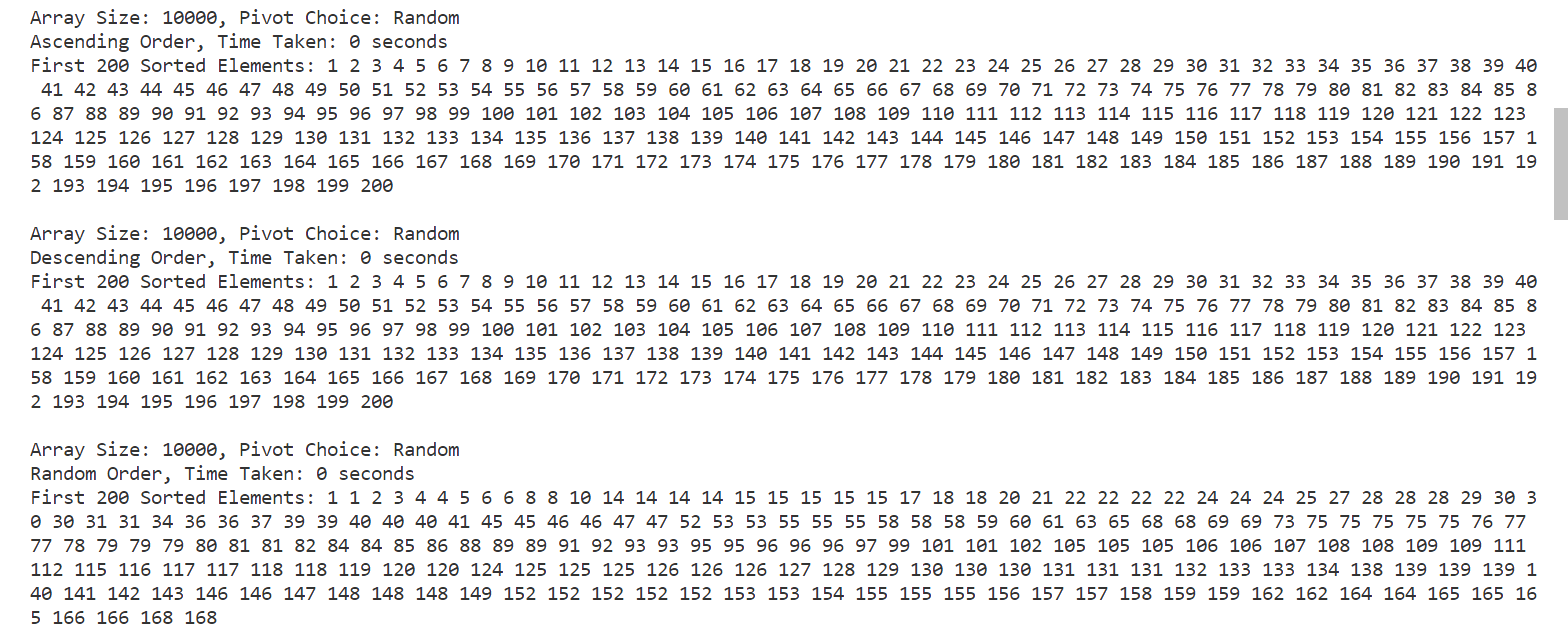
**PIVOT CHOICE HIGH**



**PIVOT CHOICE MID**

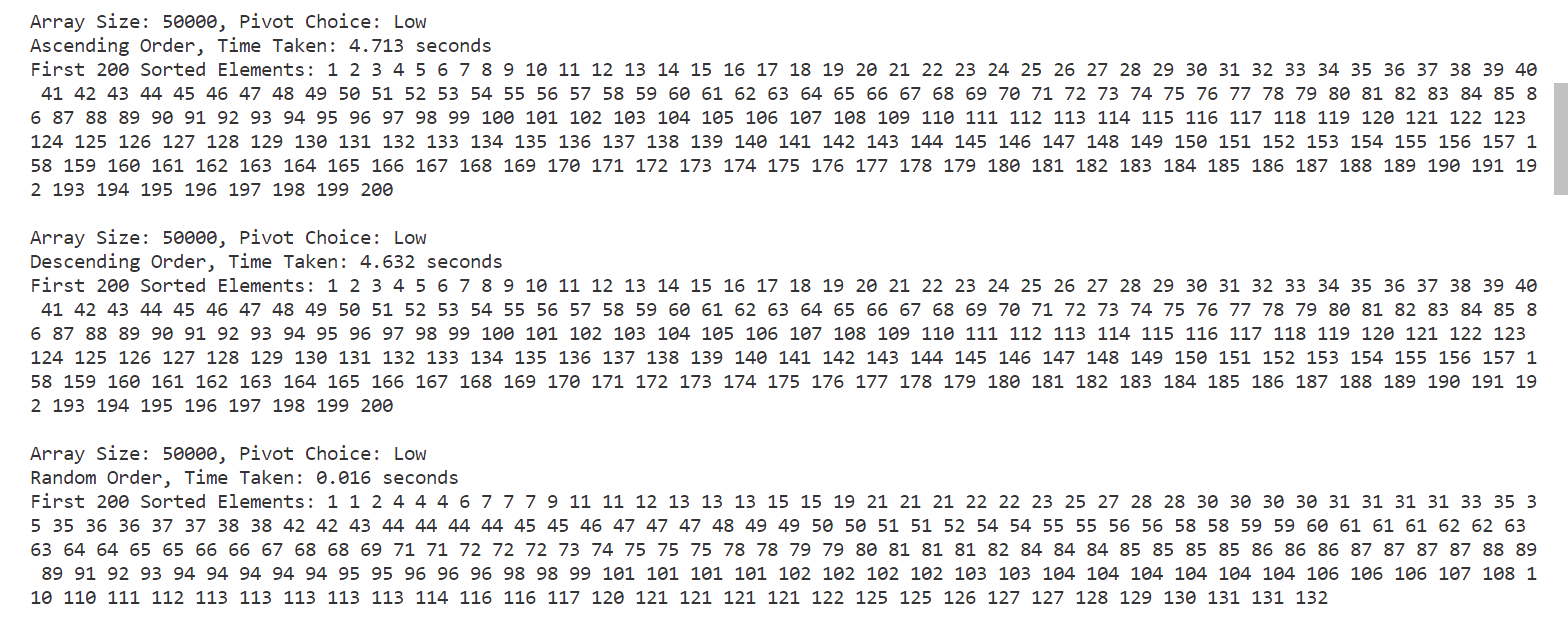


**PIVOT CHOICE RANDOM**

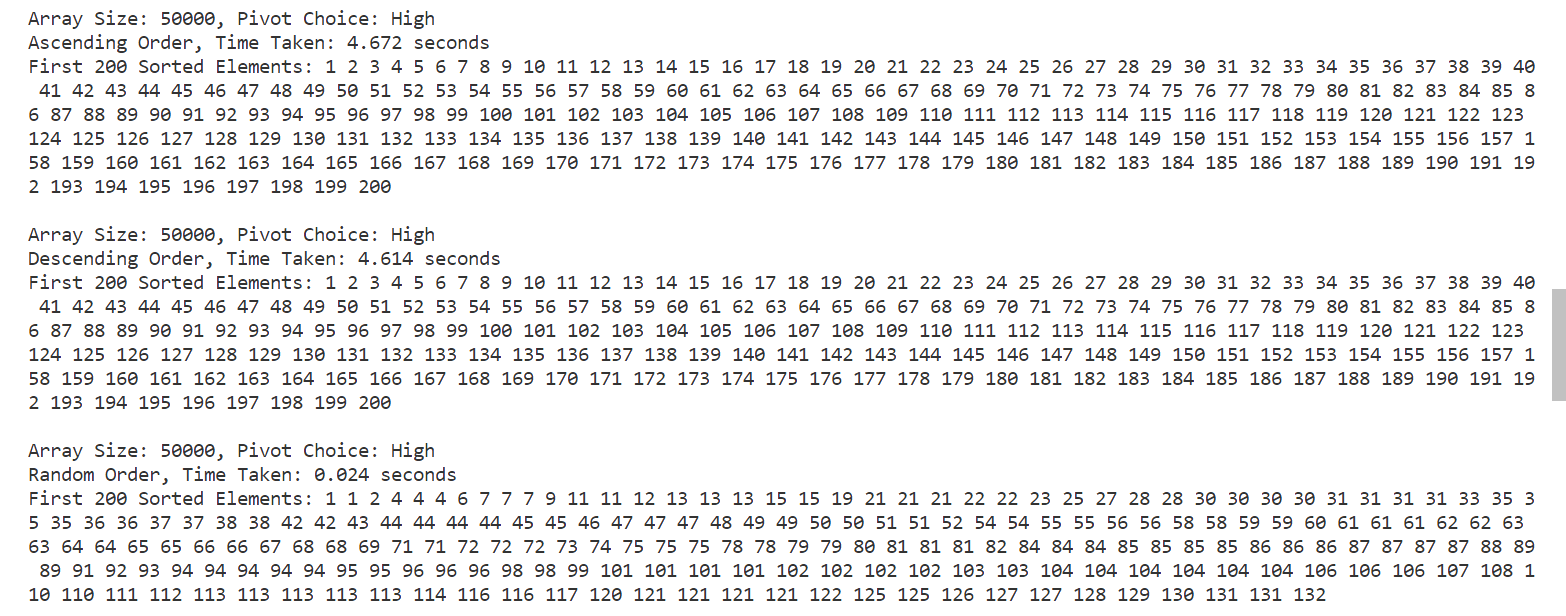


**FOR 50000 Data**

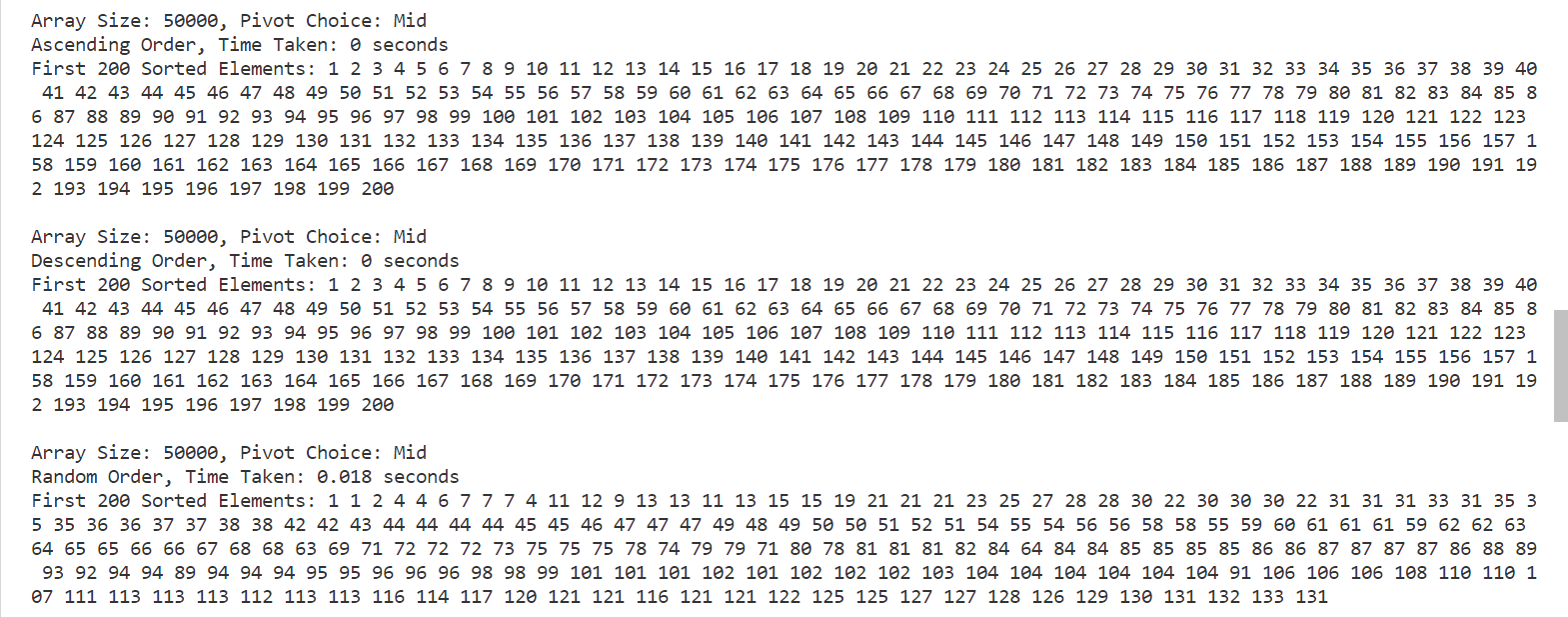
**PIVOT CHOICE LOW**



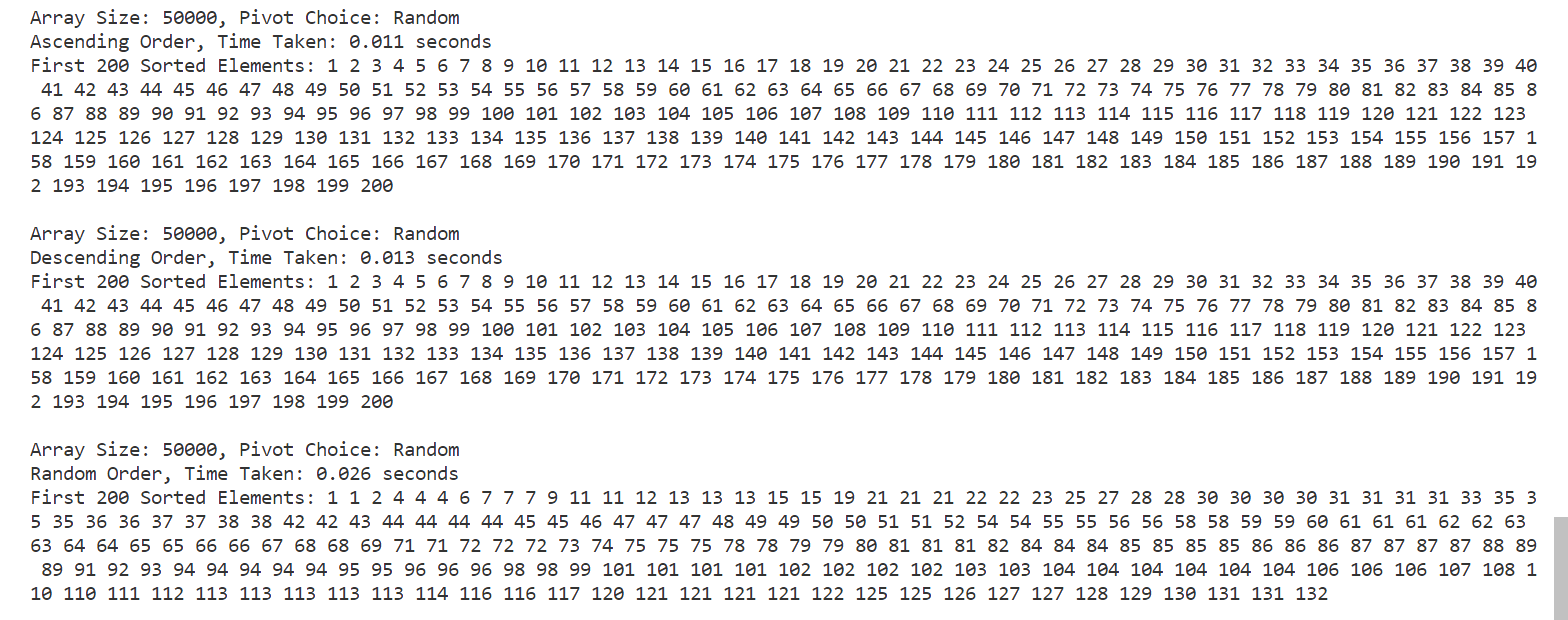
**PIVOT CHOICE HIGH**



**PIVOT CHOICE MID**

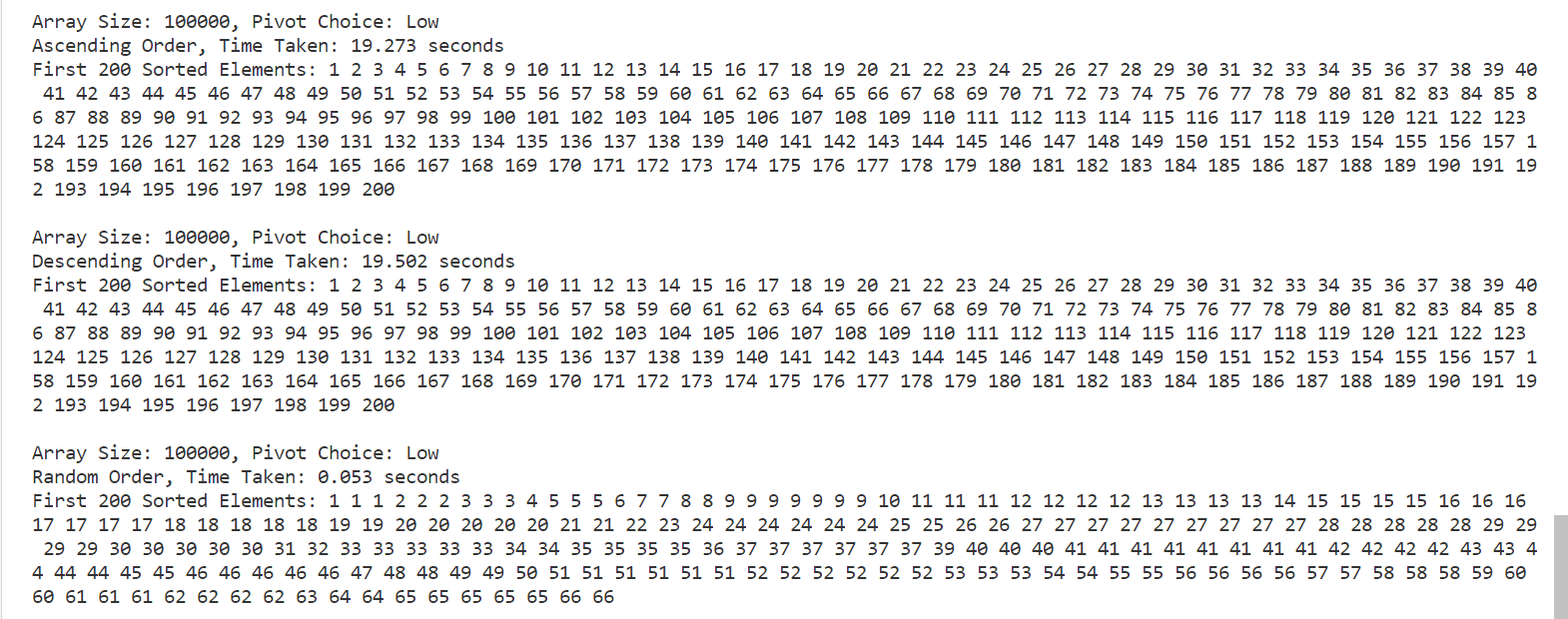


**PIVOT CHOICE RANDOM**

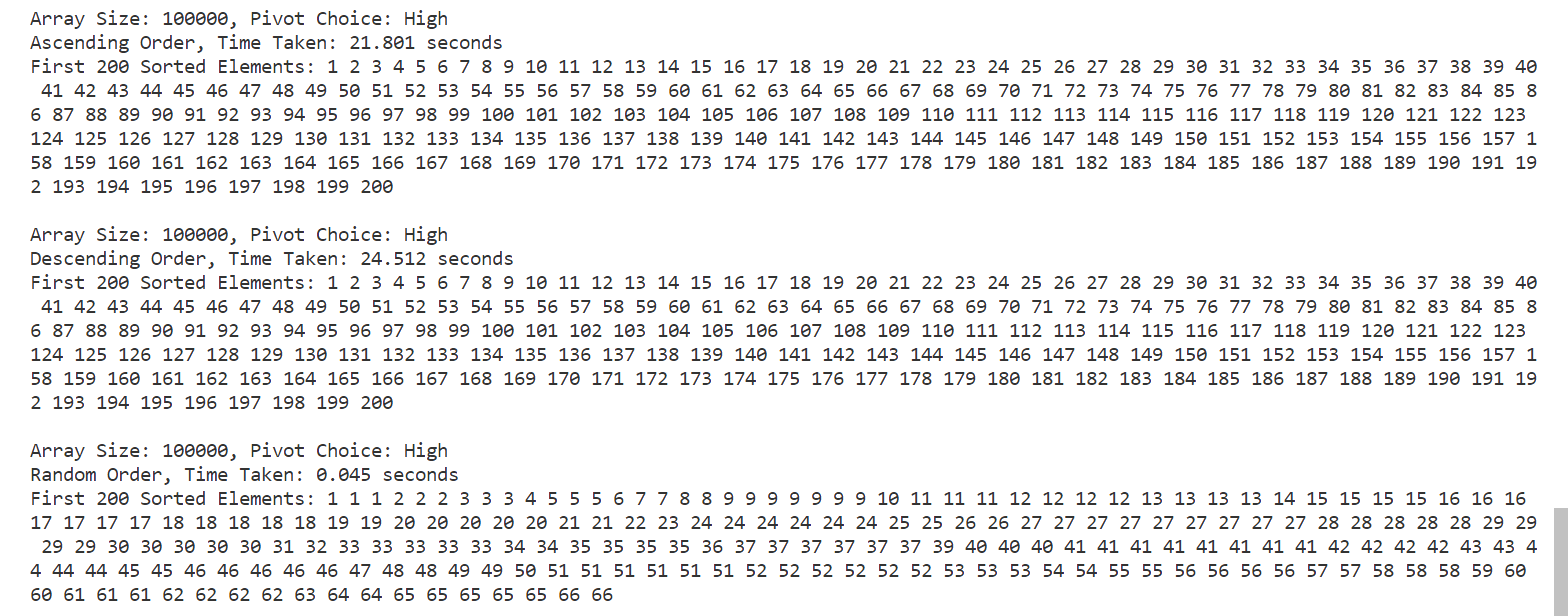


**FOR 100000 Data**

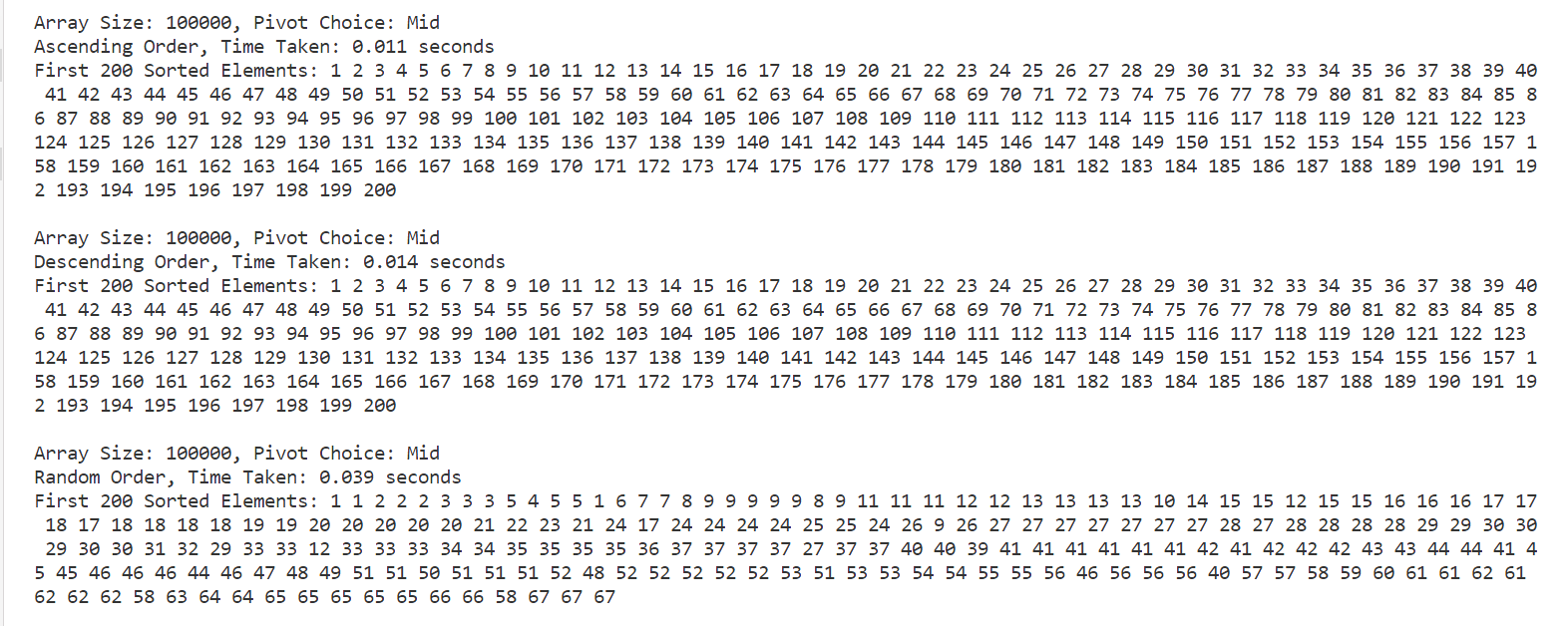
**PIVOT CHOICE LOW**



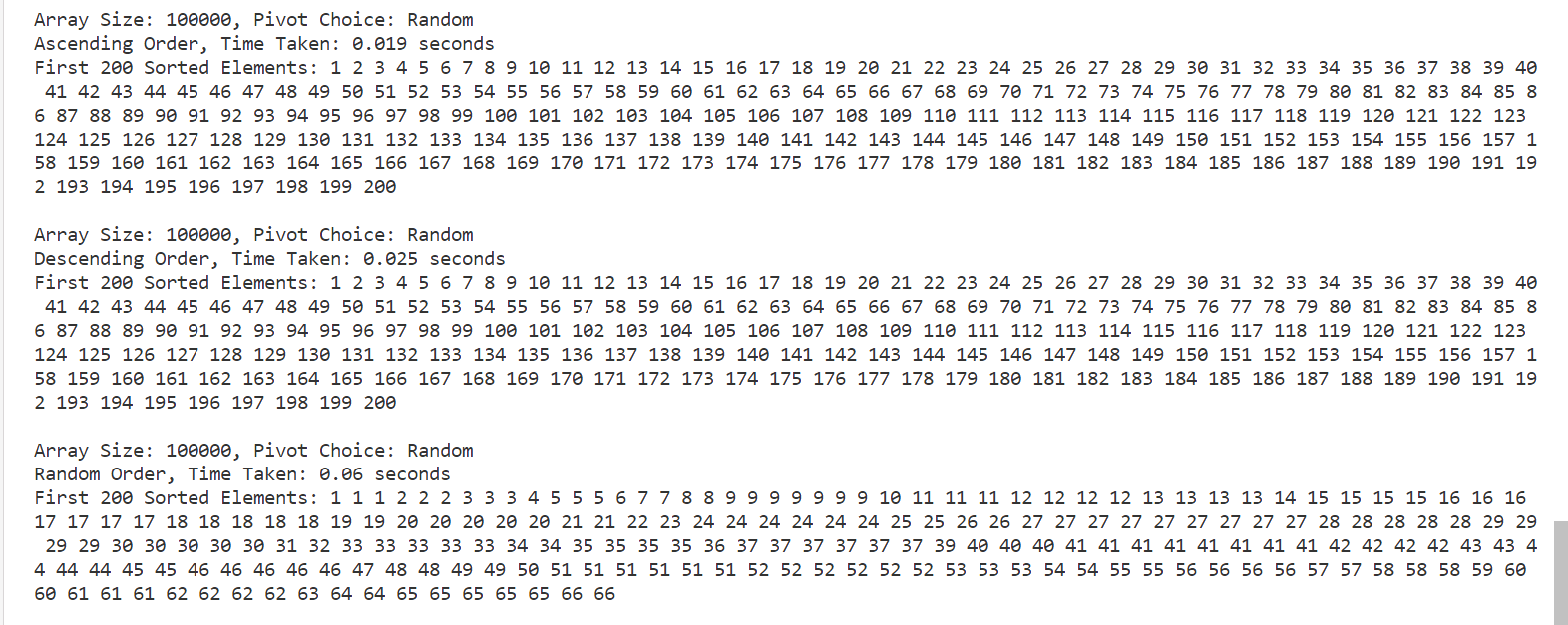
**PIVOT CHOICE HIGH**



**PIVOT CHOICE MID**



**PIVOT CHOICE RANDOM**



* **Graphs & Output Data**

**FOR ASCENDING ORDER DATA**

**FOR DESCENDING ORDER DATA**

* **Analysis**

So the time taken by quick sort for sorting the array in ascending and descending order as first and last pivot choice is more because of the array is already in sorted manner so for the ascending order data if I take first element as a pivot then it is sorted so the pivot at its correct position so the left side of pivot there is nothing and right side of pivot there is entire array so the division made by recursion is unbalanced so its takes more time to sort an array as compare to balanced partition. For the descending order same scenario right side of pivot there is nothing and left side of pivot there is descending order array so it is unbalanced partition so it takes more time. So, if my array is in sorted manner like in ascending or descending order then it takes more time to sort as compare to random order array for the first and last choice of pivot.

If I take random array and pivot as first so in most of cases the pivot elements correct position is in the array is somewhere middle index so it makes the balanced partition that takes less time than the unbalanced partition that makes in ascending and descending order array.

And for the pivot choice mid and random it makes the balanced partition so it takes lesser time then the pivot choice high and low in all such cases.

So, conclusion is Mid and Random pivot choice is better for sorting an array using quick sort