**Problem Statement:**

Design and implement the **Quick Sort algorithm** using the **Divide and Conquer method** with the **Median of Medians algorithm** for pivot selection. Determine the time required to sort the list for large values of *n*.

**Brief About the Problem:**

Sorting is one of the most fundamental problems in computer science, with numerous applications in data processing, searching, and optimization. Quick Sort is an efficient, widely used sorting algorithm based on the **Divide and Conquer** approach. However, its performance heavily depends on the choice of pivot.

1. **Traditional Quick Sort:**
   * Selects a pivot (randomly, first/last element, or middle element).
   * Partitions the array into elements **less than**, **equal to**, and **greater than** the pivot.
   * Recursively applies Quick Sort to the subarrays.
   * **Worst-case time complexity:** O(n²) (if pivot selection is poor).
2. **Quick Sort with Median of Medians:**
   * Uses the **Median of Medians algorithm** to find a good pivot, reducing the chance of an unbalanced partition.
   * Ensures a better pivot selection for **near-optimal partitioning**.
   * Results in a **guaranteed O(n log n) worst-case time complexity** instead of O(n²).

The goal is to implement **Quick Sort with Median of Medians**, analyze its time complexity, and compare its performance for large input sizes.

**Algorithm:**

1. **Median of Medians for Pivot Selection:**
   * Divide the array into groups of **5 elements** (or fewer for the last group).
   * Sort each group and pick the median.
   * Recursively find the median of these medians.
2. **Quick Sort Algorithm:**
   * Use the **median of medians** as the pivot.
   * Partition the array into three parts: elements **less than**, **equal to**, and **greater than** the pivot.
   * Recursively sort the **less than** and **greater than** subarrays.
   * Merge the sorted subarrays.

**Time Complexity Analysis:**

| **Case** | **Traditional Quick Sort** | **Quick Sort with Median of Medians** |
| --- | --- | --- |
| Best Case | O(n log n) | O(n log n) |
| Average Case | O(n log n) | O(n log n) |
| Worst Case | O(n²) | **O(n log n)** |

By using **Median of Medians**, we avoid the worst-case O(n²) complexity, making Quick Sort more efficient, especially for **large values of n**.

**Code implementation :-**

#include <iostream>

#include <ctime>

#include <cstdlib>

#include <chrono>

using namespace std;

// Function to find the k-th smallest element using the Median of Medians algorithm

int medianOfMedians(int arr[], int n, int k) {

if (n <= 5) {

// Sort the array and return the k-th element

for (int i = 0; i < n; i++) {

for (int j = i + 1; j < n; j++) {

if (arr[j] < arr[i]) {

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

}

}

}

return arr[k];

}

// Split arr into sublists of 5

int medians[100]; // Assuming a maximum of 100 groups of 5

int medianCount = 0;

for (int i = 0; i < n / 5; i++) {

medians[medianCount++] = medianOfMedians(arr + i \* 5, 5, 2); // Get median of each group

}

if (n % 5 != 0) {

medians[medianCount++] = medianOfMedians(arr + (n / 5) \* 5, n % 5, (n % 5) / 2);

}

// Recursively find the median of medians

return medianOfMedians(medians, medianCount, medianCount / 2);

}

// Quick Sort function

void quickSort(int arr[], int n) {

if (n <= 1) return;

int pivot = medianOfMedians(arr, n, n / 2);

int low[100000], high[100000]; // Assuming maximum size of 100000

int lowCount = 0, highCount = 0;

for (int i = 0; i < n; i++) {

if (arr[i] < pivot) {

low[lowCount++] = arr[i];

} else if (arr[i] > pivot) {

high[highCount++] = arr[i];

}

}

quickSort(low, lowCount);

quickSort(high, highCount);

// Combine the results back into arr

for (int i = 0; i < lowCount; i++) {

arr[i] = low[i];

}

for (int i = 0; i < n; i++) {

if (arr[i] == pivot) {

arr[lowCount + i] = pivot;

}

}

for (int i = 0; i < highCount; i++) {

arr[lowCount + (n - highCount) + i] = high[i];

}

}

// Function to measure time taken to sort a large list

void measureTime(int n) {

int\* arr = new int[n];

srand(static\_cast<unsigned int>(time(0)));

// Fill the array with random integers

for (int i = 0; i < n; ++i) {

arr[i] = rand() % 10000; // Random integers between 0 and 9999

}

auto start = chrono::high\_resolution\_clock::now();

quickSort(arr, n);

auto end = chrono::high\_resolution\_clock::now();

// Calculate duration in nanoseconds

auto duration = chrono::duration\_cast<chrono::nanoseconds>(end - start);

std::cout << "Time taken to sort " << n << " elements: " << duration.count() << " nanoseconds" << std::endl;

delete[] arr; // Free allocated memory

}

int main() {

int n;

std::cout << "Enter the number of elements to sort: ";

std::cin >> n;

measureTime(n);

return 0;

}



