**PRACTICAL # 03**

**OBJECT:**

The run-time analysis and comparison of a traditional sorting algorithm and a fast sort algorithm

**THEORY:**

Traditional sort algorithms typically take O(N^2) steps to sort a list of N elements. That means if the number of elements is 10, the algorithm will require 100 steps in worst case to sort this data. And if there are 1 million elements to sort, it will require square of 1 million steps for sorting! This is something that requires high computation power of machine.

The fast sort algorithms typically have worst-case complexity of O(N \* log N) to sort a list of N elements. While the traditional slow sort algorithms have worst-case complexity of O(N^2) to sort a list of N elements. As the number of elements grow, the difference in execution time becomes more significant.

A fast sort algorithm like quick sort uses divide and conquer strategy to minimize the number of steps to sort the data. This reduces the complexity significantly.

**CODE:**

The program below defines two algorithms. A traditional selection sort(slow sort) algorithm and a fast sort(quick sort) algorithm in C++.

The selection sort starts with a loop nested in other. The algorithm starts with first index on the array and checks all the following elements of the array. It substitutes the minimum element in first position. In second iteration of the main loop, it starts with second index and repeats the same process to find the second minimum element in the array and substitutes it. This continues till the end of the array.

The quick sort starts with a pivot element at the starting index of the array. It partitions the array in two sub-arrays. These two sub-arrays are sorted in a way that one sub-array has all the elements less than the pivot element and the other sub-array has all its elements greater than the pivot element. The pivot element at this point is already in its sorted position and will not be processed further. Now the quick sort calls itself recursively starting with these two sub-arrays, ignoring the pivot element.

*#include <iostream>*

*#include <time.h>*

*#include <iomanip>*

*using namespace std;*

*//Selection Sort*

*int\* selectionSort(int arr[], int lastIndex){*

*int temp=0;*

*for(int i=0; i<**=lastIndex; i++ ){*

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

*if(arr[i] > arr[j] ){*

*//swap elements*

*temp = arr[i];*

*arr[i] = arr[j];*

*arr[j] = temp;*

*}*

*}*

*}*

*return arr;*

*}*

*//array partition function for quick sort*

*int partition(int arr[], int start, int end){*

*int pivotElement = arr[start], pivotIndex = start, i = start;*

*for(;i<=end; i++){*

*if(arr[i] < pivotElement ){*

*arr[pivotIndex] = arr[i];*

*arr[i] = arr[pivotIndex + 1];*

*arr[pivotIndex+1] = pivotElement;*

*pivotIndex++;*

*}*

*}*

*return pivotIndex;*

*}*

*//Quick Sort*

*int\* quickSort(int arr[], int start, int end){*

*if(start <end){*

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

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

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

*}*

*return arr;*

*}*

*//Main function*

*int main() {*

*clock\_t begin, end;*

*double time\_spent;*

*//set the precision of the real number*

*cout.precision(20);*

*//Unsorted array*

*int arr[] = { 111, 14, 4, 6, 8, 34, 123, 54, 555, 320,*

*449, 389, 3, 56, 78, 545, 657, 26, 339, 112};*

*cout<<"\n\nSELECTION SORT";*

*begin = clock(); //start time tracking*

*int\* sArr = selectionSort(arr, 19);*

*end = clock(); //stop time tracking*

*time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;*

*cout<< "\ntime spent = "<<time\_spent;*

*cout<<"\n\nQUICK SORT";*

*begin = clock(); //start time tracking*

*int\* qArr = quickSort(arr, 0, 19);*

*end = clock(); //stop time tracking*

*time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;*

*cout<< "\ntime spent = "<<time\_spent<<endl;*

*return 0;*

*}*

The complexity of algorithm increases significantly with nesting of the loops. If there is one loop, the complexity is in some order of O(N). In case of one nested loop, the complexity is in order of O(N^2). Thus the number of nested loops defines the power of N. Where N is the number of data elements and the complexity is the number of steps to process that data.

In the above example, the selection sort has one loop nested inside the other. Thus it requires the square number of iterations of the number of data elements to sort the array. The quick sort, on the other hand, uses divide and conquer strategy and has only one loop to solve the sorting problem. The performance difference between these two types of algorithms grows as the number of data elements to sort increases.

**ACTIVITY:**

1. Execute the program given above in any C++ compiler. Check the number of steps executed, changing the array values and the search number.

2. Execute the program given above in any C++ compiler. Check by changing the array values and the search number.

**REVIEW QUESTIONS**

1. What is the advantage of sorting information?

1. What is the run-time complexity of a slow sort algorithm?
2. How can we reduce the runtime complexity of a sorting algorithm?
3. What is the worst case complexity of quick sort algorithm?
4. How the quick sort optimizes the performance for sorting?
5. What differentiates fast sort algorithm from slow sort?