|  |  |
| --- | --- |
| **Experiment No.** | **2** |
| **Aim** | **Experiment on finding the running time of an algorithm.(Quick sort and Merge sort).** |
| **Name** | **Atharva Gite** |
| **UID No.** | **2021300038** |
| **Class & Division** | **COMPS-A(C)** |

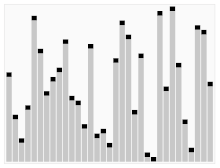
**Theory:**

**1. Quick Sort:**

Quicksort is an efficient, general-purpose sorting algorithm. Quicksort was developed by British computer scientist Tony Hoare in 1959 and published in 1961, it is still a commonly used algorithm for sorting.

Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called partition-exchange sort. The sub-arrays are then sorted recursively. This can be done in place, requiring small additional amounts of memory to perform the sorting.

Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. Most implementations of quicksort are not stable, meaning that the relative order of equal sort items is not preserved



Animated Representation of Quick-Sort

Inventor: Tony Hoare

Worst complexity: n^2

Average complexity: n\*log(n)

Best complexity: n\*log(n)

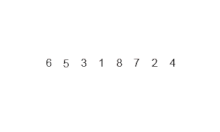
Method: Partitioning

Stable: No

**2. Merge Sort:**

Merge sort is an efficient, general-purpose, and comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the order of equal elements is the same in the input and output.

Merge sort is a divide-and-conquer algorithm that was invented by John von Neumann in 1945. A detailed description and analysis of bottom-up merge sort appeared in a report by Goldstine and von Neumann as early as 1948.



Inventor: John von Neumann

Worst complexity: n\*log(n)

Average complexity: n\*log(n)

Best complexity: n\*log(n)

Method: Merging

Stable: Yes

**CODE:**

a)Code to Generate Random Numbers

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int main () {

   FILE \*fp;

   fp = fopen("numbers.txt", "w");

   long long int i, n;

   n = 100000;

   srand((unsigned) time(&t));

   for( i = 0 ; i < n ; i++ )

   {

   int num=rand()%100;

   num=num\*100000;

   num+=rand();

   fprintf(fp, "%d\n", num);

   }

   return(0);

}

b)Main code

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include<stdint.h>

#include <sys/time.h>

#define N 100000

#define BLOCK\_SIZE 100

int partition (int arr[], int low, int high)

{

    int pivot = arr[high];

    int i = (low - 1);

    for (int j = low; j <= high- 1; j++)

    {

        if (arr[j] < pivot)

        {

            i++;

            int temp = arr[i];

            arr[i] = arr[j];

            arr[j] = temp;

        }

    }

    int temp = arr[i + 1];

    arr[i + 1] = arr[high];

    arr[high] = temp;

    return (i + 1);

}

void quick\_sort(int arr[], int low, int high)

{

    if (low < high)

    {

        int pi = partition(arr, low, high);

        quick\_sort(arr, low, pi - 1);

        quick\_sort(arr, pi + 1, high);

    }

}

void merge(int arr[], int l, int m, int r) //l=left m-middle r=right

{

    int i, j, k;

    int n1 = m - l + 1;

    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (j = 0; j < n2; j++)

        R[j] = arr[m + 1 + j];

    i = 0;

    j = 0;

    k = l;

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        } else {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;

    }

    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;

    }

}

void merge\_sort(int arr[], int l, int r)

{

    // left < right

    if (l < r) {

        int m = l + (r - l) / 2;

        merge\_sort(arr, l, m);

        merge\_sort(arr, m + 1, r);

        merge(arr, l, m, r);

    }

}

int main() {

    FILE \*fp,\*ptrm,\*ptrq;

    struct timeval start, end;

    int arr[N],k,n,j,i;

    fp = fopen("numbers.txt", "r");

    k=0;

    // Read each number from the file and store it in the array

    while (fscanf(fp, "%d", &n) == 1)

    {

        arr[k++] = n;

    }

    // Close the file

    fclose(fp);

    ptrm= fopen("merge\_sort\_time.txt", "w");

    ptrq= fopen("quick\_sort\_time.txt", "w");

    //clock start

    for (i=500;i<1001;i++)

    {

        double merge\_sort\_time = 0;

        double quick\_sort\_time = 0;

        // 1\*100= 100 blocks, 2\*100=200 blocks and so on

        int size;

        size=i\*BLOCK\_SIZE;

        // block arr to find out combien de time for sorting 0-99,0-199 and so on

        int block\_arrm[size];

        int block\_arrq[size];

        for(j = 0; j<size ; j++)

        {

        block\_arrm[j] = arr[j];

        block\_arrq[j]=arr[j];

        }

            gettimeofday(&start, NULL);

            merge\_sort(block\_arrm,0,size-1);

            gettimeofday(&end, NULL);

            uint64\_t delta\_usm = (end.tv\_sec - start.tv\_sec) \* 1000000 + (end.tv\_usec - start.tv\_usec);

            gettimeofday(&start, NULL);

            quick\_sort(block\_arrq,0,size-1);

            gettimeofday(&end, NULL);

            uint64\_t delta\_usq = (end.tv\_sec - start.tv\_sec) \* 1000000 + (end.tv\_usec - start.tv\_usec);

            fprintf(ptrm, "%lu\n",delta\_usm);

            fprintf(ptrq, "%lu\n",delta\_usq);

            }

    return 0;

}

**GRAPHS:**

**Result:**  For smaller inputs, merge sort is more efficient. Overall, quick sort is faster than merge sort for randomized data, particularly on larger distributions.