| | |
|---|---|
| **Experiment No.** | **02** |
| **Aim** | **Experiment based on divide and conquer approach.** |
| **Name** | **Pooja Gajendra Patil** |
| **UID No.** | **2022301011** |
| **Class & Division** | **Computer Engineering-A** |

**Theory/Experiment:**

**Quicksort**– It picks an element called as pivot, and then it partitions the given array around the picked pivot element. It then arranges the entire array in two sub-array such that one array holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.The Divide and Conquer steps of Quicksort perform following functions. Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element. Conquer: Recursively, sort two subarrays with Quicksort Combine: Combine the already sorted array.

**Merge sort**– Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging.The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into twoelement lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

main.c data.txt

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <time.h>

void dataInput()
{
        FILE *fptr;
        fptr = fopen("data.txt", "w");
    for (int i = 0; i < 100000; i++)
    {
        int temp = rand() ;
        fprintf(fptr, "%d\n", temp);
    }
        fclose(fptr);
}
void merge(long arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    /* create temp arrays */
    int L[n1] ;
    int R[n2];


    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
```

main.c data.txt

```c
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
    are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
    are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(long arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
```

main.c | data.txt

```c
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

// function to swap elements
void swap(long *a, long *b) {
  int t = *a;
  *a = *b;
  *b = t;
}

// function to find the partition position
int partition(long array[], int low, int high) {

  // select the rightmost element as pivot
  int pivot = array[high];

  // pointer for greater element
  int i = (low - 1);

  // traverse each element of the array
  // compare them with the pivot
  for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {

      // if element smaller than pivot is found
      // swap it with the greater element pointed by i
      i++;

      // swap element at i with element at j
      swap(&array[i], &array[j]);
```

main.c | data.txt

```c
      // swap element at i with element at j
      swap(&array[i], &array[j]);
    }
  }

  // swap the pivot element with the greater element at i
  swap(&array[i + 1], &array[high]);

  // return the partition point
  return (i + 1);
}

void quickSort(long array[], int low, int high) {
  if (low < high) {

    // find the pivot element such that
    // elements smaller than pivot are on left of pivot
    // elements greater than pivot are on right of pivot
    int pi = partition(array, low, high);

    // recursive call on the left of pivot
    quickSort(array, low, pi - 1);

    // recursive call on the right of pivot
    quickSort(array, pi + 1, high);
  }
}

void printArray(int A[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

int main()
{
    dataInput();

```

main.c data.txt

```c
    dataInput();

    FILE *fptr;
    fptr = fopen("data.txt", "r");
    long arr[99999], arr1[99999], arr2[99999];
    for (int i = 0; i < 99999; i++)
    {
        fscanf(fptr, "%8ld", &arr[i]);
    }
    int s = 100;

    printf("Size\tMerge Sort\tQUICK sORT \n");
    for (int i = 0; i <= 1000; i++)
    {
        for (int j = 0; j < 100000; j++)
        {
            arr1[j] = arr[j];
            arr2[j] = arr[j];
        }

        double diff1,diff2;
        struct timespec start, end;
        int i;

        clock_gettime(CLOCK_MONOTONIC, &start);

        int arr_size = sizeof(arr1) / sizeof(arr1[0]);
        mergeSort(arr1, 0, s);
        clock_gettime(CLOCK_MONOTONIC, &end);



        diff1 = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1000000000.0;

       clock_gettime(CLOCK_MONOTONIC, &start);
        quickSort(arr2,0, s);
        clock_gettime(CLOCK_MONOTONIC, &end);
        diff2 = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec)/1000000000.0;
        printf("%d\t%f\t%f\n", s, diff1, diff2);
        s+=100;
```

```c
        diff2 = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec)/1000000000.0;
        printf("%d\t%f\t%f\n", s, diff1, diff2);
        s+=100;
}
        return 0;

}
```

**Conclusion:**

In this experiment we have learnt about how to find the run time of divide and conquer algorithm i.e Merge and Quick sort
Merge sort divides the original array into N subarray of size one each then repeatedly merges two in same order.
Quick sort selects an element as pivot and partitions the array around it. It moves all the element greater than to its right, then recursively sort the subarrays

Space Complexity:

Merge sort → An extra array of size N is. needed to store the merged array. Thus Space complexity is $O(N)$

Quick sort doesnot require extra array but complexity depends on pivot. If we select largest or smallest element as pivot there are total N recursive calls. Thus space complexity for such case is $O(N)$. If we manage to partition the array in equal halves each time the size of recursion tree is log N. So its space complexity is $O(n \log n)$.