**WORKSHEET 2.1**

**1. Aim:**

Write a program to implement merge sort and quick sort and compare their worst case complexities.

**2. Problem Description:**

To implement merge sort and quick sort and compare their worst case complexities.

**For Merge Sort –**

**3. Algorithm:**

**Mergesort(a,beg,end):**

Step 1: Repeat steps 2 to 3 when mergesort is recursively called.

Step 2: Check if beg<end then:

1. Calculate mid i.e. mid=(beg+end)/2;

2. mergesort(a,beg,mid);

3. mergesort(a,mid+1,end);

4. mergingsortedsubarrays(a,beg,mid,mid+1,end);

Step3: Return

**Mergingsortedsubarrays(a,beg,mid,mid+1,end):**

Step 1: Repeat steps 2 to 11 when mergingsortedsubarrays is called.

Step 2: Set lb=beg,lr=mid,rb=mid+1,rr=end,na=lb,nb=rb,nc=lb

Step 3: Repeat steps 4 & 5 while(na<=lr)&(nb<=rr)

Step 4: if a[na]<a[nb] then:

set c[nc]=a[na] and na=na+1

else

set c[nc]=a[nb] and nb=nb+1

Step 5: nc=nc+1

Step 6: execute step 7 if(na>lr)

Step 7: Repeat while nb<=rr

Set c[nc]=a[nb] and nc=nc+1 and nb=nb+1

Step 8: Execute step 9if (na<lr)

Step 9:Repeat while na<=lr

Set c[nc]=a[na] and nc=nc+1 and na=na+1

Step 10:Repeat for k=lb to rr

Set a[k]=c[k]

Step 11: Return

**4. Computational Complexity:-**

* The major work is done by the merge procedure which has complexity of O(n). The merge procedure is called from merge sort procedure after the array is divided into 2 halves and each half has been sorted.
* In each of the recursive calls, one for left half and one for right half, the array is divided into halves, thus dividing the array into 4 segments. At each level, the no. of segment doubles. Therefore, the total divisions are log n. Hence the total number of comparisons are O(n logn).
* The only disadvantage of merge sort is that it uses an extra temporary array of the same size as that of input array, to merge two halves. The elements of the temporary array are copied back to the original array before next merging.

**5. Pseudo Code :-**

procedure mergesort( var a as array )

if ( n == 1 ) return a

var l1 as array = a[0] ... a[n/2]

var l2 as array = a[n/2+1] ... a[n]

l1 = mergesort( l1 )

l2 = mergesort( l2 )

return merge( l1, l2 )

end procedure

procedure merge( var a as array, var b as array )

var c as array

while ( a and b have elements )

if ( a[0] > b[0] )

add b[0] to the end of c

remove b[0] from b

else

add a[0] to the end of c

remove a[0] from a

end if

end while

while ( a has elements )

add a[0] to the end of c

remove a[0] from a

end while

return c

end procedure

**6. Source Code:**

#include<iostream>

using namespace std;

void mergesort(int a[],int beg,int end);

void mergingsortedsubarrays(int a[7],int lb,int lr,int rb,int rr);

int main()

{

int i,n=7,a[7]={7,4,9,1,3,2,8};

cout<<endl<<"After Sorting array elements are :”;

for(i=0;i<7;i++)

cout<<a[i]<<" ";

mergesort(a,0,n-1);

cout<<endl<<"After Sorting array elements are :”;

for(i=0;i<=6;i++)

cout<<a[i]<<" ";

return 0;

}

void mergesort(int a[],int beg, int end)

{

int mid;

if(beg<end)

{

mid=(beg+end)/2;

mergesort(a,beg,mid);

mergesort(a,mid+1,end);

mergingsortedsubarrays(a,beg,mid,mid+1,end);

}

}

void mergingsortedsubarrays(int a[7],int lb,int lr,int rb,int rr)

{

int na,nb,nc,k,c[7];

na=lb;

nb=rb;

nc=lb;

while((na<=lr)&&(nb<=rr))

{

if(a[na]<a[nb])

{

c[nc]=a[na];

na++;

}

else

{

c[nc]=a[nb];

nb++;

}

nc++;

}

if(na>lr)

{

while(nb<=rr)

{

c[nc]=a[nb];

nc++;

nb++;

}

}

else

{

while(na<=lr)

{

c[nc]=a[na];

nc++;

na++;

}

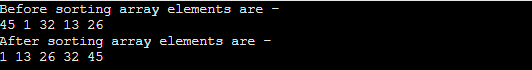
}

for(k=lb;k<=rr;k++)

a[k]=c[k];

}

1. **Screenshot of Output:**



**For Quick Sort -**

1. **Algorithm:**

QUICKSORT (array A, start, end)

1. {
2. if (start < end)
3. {
4. p = partition(A, start, end)
5. QUICKSORT (A, start, p - 1)
6. QUICKSORT (A, p + 1, end)
7. }

}

PARTITION (array A, start, end)

1. {
2. pivot ? A[end]
3. i ? start-1
4. for j ? start to end -1 {
5. do if (A[j] < pivot) {
6. then i ? i + 1
7. swap A[i] with A[j]
8. }}
9. swap A[i+1] with A[end]
10. return i+1

}

**9. Computational Complexity:-**

* Best Case: Ω (N log (N))

The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.

In this case, the algorithm will make balanced partitions, leading to efficient Sorting.

* Average Case: θ ( N log (N))

Quicksort’s average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.

* Worst Case: O(N2)

The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort ) to shuffle the element before sorting.

**10. Pseudo Code :-**

procedure quickSort(left, right)

if right-left <= 0

return

else

pivot = A[right]

partition = partitionFunc(left, right, pivot)

quickSort(left,partition-1)

quickSort(partition+1,right)

end if

function partitionFunc(left, right, pivot)

leftPointer = left

rightPointer = right - 1

while True do

while A[++leftPointer] < pivot do

//do-nothing

end while

while rightPointer > 0 && A[--rightPointer] > pivot do

//do-nothing

end while

if leftPointer >= rightPointer

break

else

swap leftPointer,rightPointer

end if

end while

swap leftPointer,right

return leftPointer

end function

end procedure

**11. Source Code:**

#include <stdio.h>

// Function to swap two elements

void swap(int\* a, int\* b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

// Partition the array using the last element as the pivot

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

{

// Choosing the pivot

int pivot = arr[high];

// Index of smaller element and indicates

// the right position of pivot found so far

int i = (low - 1);

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

// If current element is smaller than the pivot

if (arr[j] < pivot) {

// Increment index of smaller element

i++;

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

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

// The main function that implements QuickSort

// arr[] --> Array to be sorted,

// low --> Starting index,

// high --> Ending index

void quickSort(int arr[], int low, int high)

{

if (low < high) {

// pi is partitioning index, arr[p]

// is now at right place

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

// Separately sort elements before

// partition and after partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// Driver code

int main()

{

int arr[] = { 10, 7, 8, 9, 1, 5 };

int N = sizeof(arr) / sizeof(arr[0]);

printf("Array before perform sorting : \n");

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

printf("%d ", arr[i]);

printf("\n");

// Function call

quickSort(arr, 0, N - 1);

printf("Array after perform sorting : \n");

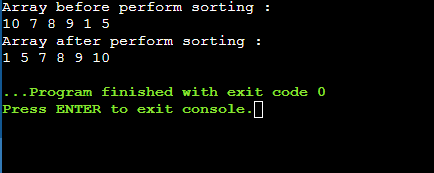
for (int i = 0; i < N; i++)

printf("%d ", arr[i]);

return 0;

}

1. **Screenshot of Output:**



1. **Learning & Outcomes:**

* Learned about the merge sorting and quick sort Algorithm, how it works, How much is the time complexity.
* Learned to create dynamic array using pointers.
* Used divide and conquer approach and learnt it.
* Learnt to find pivot element in quick sort algorithm