Real time application to compare Quick sort and Merge sort

(PROJECT REPORT)

Curriculum for applied learning

Data Structures and Algorithms-CSE2003

19BCE0731-ANIKET PARAMHANS SINGH

19BCE0740-HITEN RAI VATSA

19BCE0743-B ADITYA KRISHNA

(B.Tech in Computer Science and Engineering)

School of Computing Science and Engineering (SCOPE)

FACULTY-PROF. RAGHAVAN R

ABSTRACT

The aim of this project is to show the comparison of Quick sort and Merge sort techniques. The program is menu driven to fit to the convenience of the user, depending on how many words he wants to enter, and sort them through the sorting technique of his/her choice. In the end I will be showing which sorting is better, through the respective worst and best case time complexity.

CONTENTS

INTRODUCTION

Real life problem: - A person wanted to organize foreign words that he needed to translate so that he could look them up in a dictionary in order, without jumping from section to section.

Methodology: - Two sorting techniques will be used, quick sort and merge sort, and the average and worst case time complexity will be compared for the sorting techniques.

1.) Merge Sort

It divides the inputted list into two halves, then halves those halves, and continues doing this until it gets to a base case and starts working its way back up merging base cases together, sorting as it goes until the list is whole again.

2.) Quick Sort

Quicksort is a recursive sorting routine that works by partitioning the array so that items with smaller keys are separated from those with larger keys and recursively applying itself to the two groups.

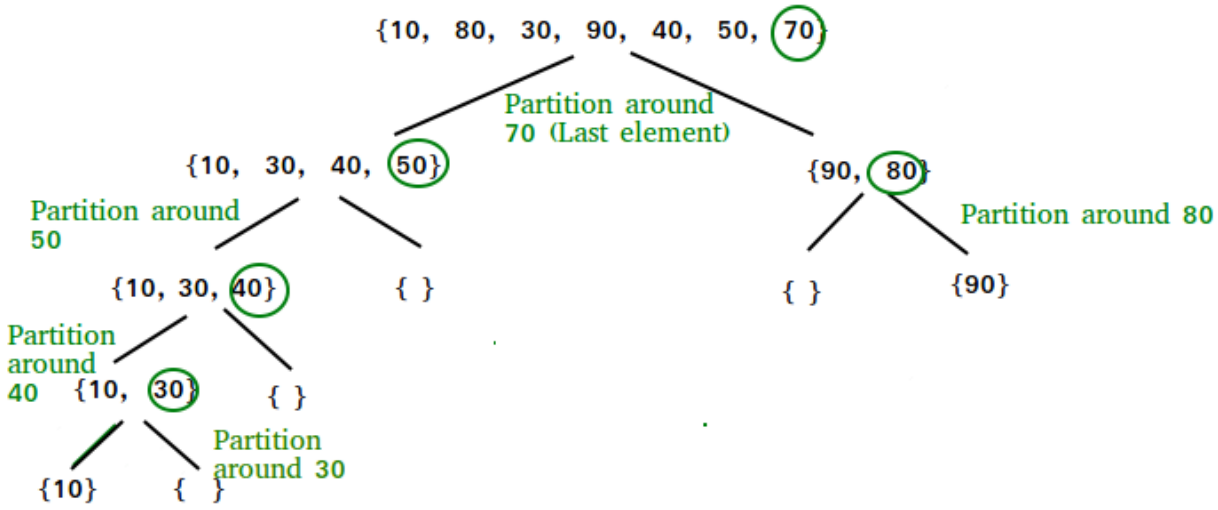
Algorithm for Quick sort

```
int partition (int arr[], int low, int high)
{
   int pivot = arr[high];       // pivot
   int i = (low - 1);  // Index of smaller element

   for (int j = low; j <= high- 1; j++)
   {
      // If current element is smaller than the pivot
      if (arr[j] < pivot)
      {
         i++;    // increment index of smaller element
         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);
   }
}
```


{10, 80, 30, 90, 40, 50, 70}
Partition around
70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around
50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition
around
40
{10, 30}
{ }
Partition
around 30
{10}
{ }

Algorithm for Merge sort

```
void merge(int 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], 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)
    {
        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)
    {
```

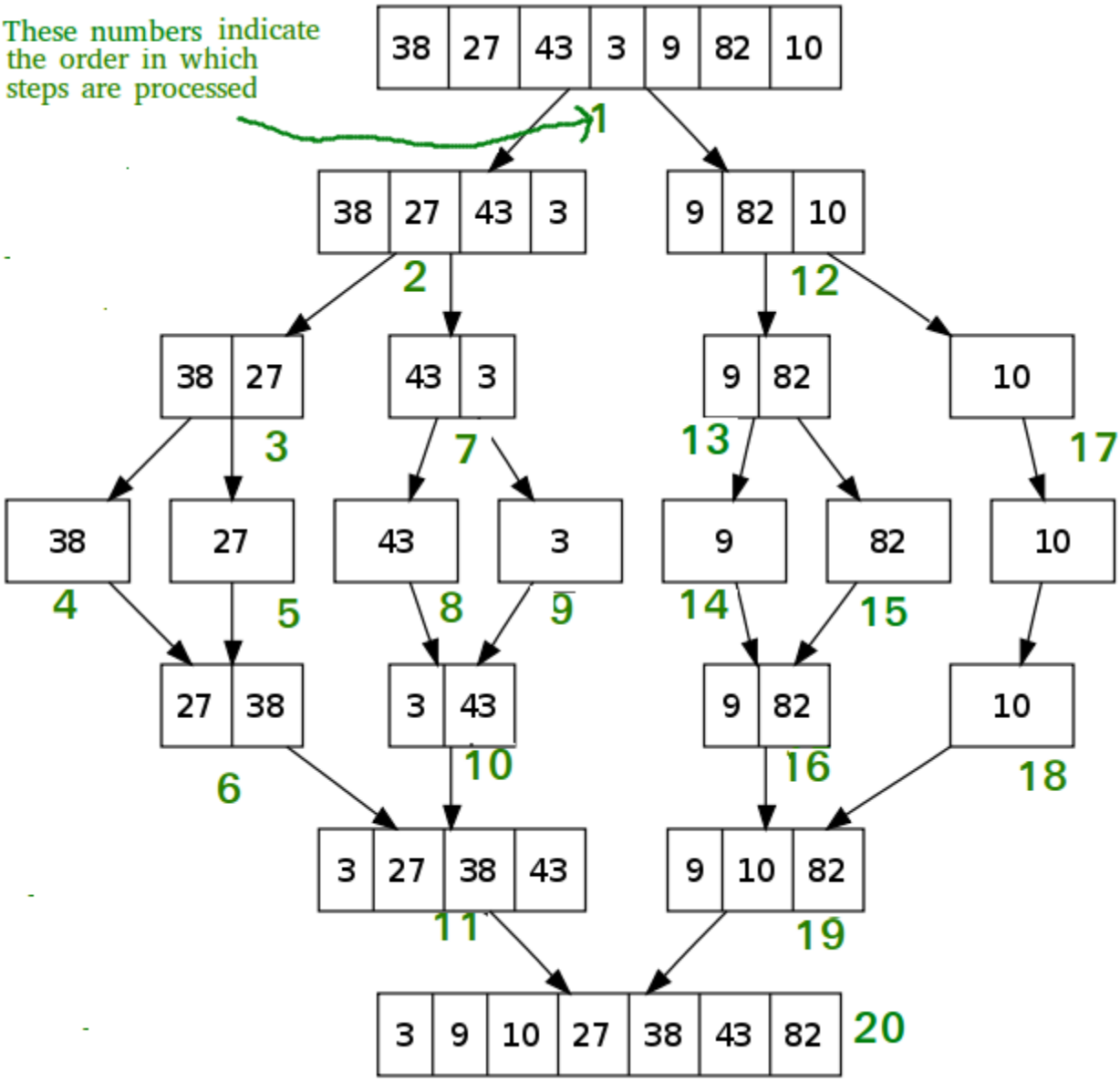
```
        arr[k] = L[i];
        i++;
        k++;
    }

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

/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // 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);
    }
}
```


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

DISADVANTAGES OF MERGE SORT AND QUICK SORT

- MERGE SORT
  - Slower comparatives to the other sort algorithms for smaller tasks.
  - Uses more memory space to store sub-elements of the initial split list.
  - Less efficient than other sorts.

- QUICK SORT
  - Its running time can differ depending on the contents of the array.
  - It is not stable.
  - The worst-case performance is similar to average performance of the bubble, insertion or selection sort.

C code for the program

```
/* Declaration Of Header Files */
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>//for string operations
char *a[100],*b[100];
int l,k,No2=0,j,i;
char *t,*m;
int  Scan(char *a[100]);
int  QuickS(char *a[100],int,int);
void Disp1();
void Split(int,int);
void MergeS(int,int,int);
void Disp2();
/* Start Of Main Program */
int main()
{
   int No1 = 0;
   printf("\nWelcome to 'Quick' And 'Merge' types of sorting program :");
   do
```

```
{
   printf("\nPlease select one of the following options :");
   printf("\n1. Quick Sorting");
   printf("\n2. Merge Sorting");
   printf("\n3. Exit from menu");
   printf("\nPlease enter your choice here : ");
   scanf("%d",&No1);
   switch(No1)
   {
      case 1: No2 = (Scan(a));
         l = 0;
         QuickS(a , l , No2);              // Call To The Function.
         Disp1();
         break;
      case 2: No2 = (Scan(a));
         Split(0,No2-1);
         Disp2();
         break;
      case 3: exit (0);
         break;
      default:printf("\nSorry! Invalid choice. ");
   }
}
```

```
    while(No1 <= 3);
    getch();
}
int Scan(char *a[100])
{
    int i;
    printf("\nPlease enter the limit of array A : ");
    scanf("%d",&No2);
    printf("\nPlease enter the strings for array A : ");
    for(i=0;i<No2;i++)
    {
        scanf("%s",&a[i]);
    }
    for(i=0;i<No2;i++)
    {
            printf("%s",a[i]);
            }
    return (No2);
}
int QuickS(char *a[],int l,int No2)
{
    if(l >= No2)
        return(0);
```

```
k=l;
i=l;
j=No2;
m=a[i];
do
{
   do
   {
      i++;
   }
   while(a[i]<=m && i<No2);
   do
   {
      j--;
   }
   while(a[j]>=m && j>l);
   if(i<j)
   {
      t=a[i];
      a[i]=a[j];
      a[j]=t;
   }
}
```

```
    while(i<j);


    t=a[j];
    a[j]=m;
    a[k]=t;
    QuickS(a,l,j);
    QuickS(a,j+1,No2);
    return(0);
}
void Disp1()
{
    printf("\nQuick Sorted array 'A' is as follows : ");
    for(i=0;i<No2;i++)
    {
        printf("%s\n",a[i]);
    }
}
void MergeS(int low,int mid,int high)
{
    int i,h,j,k;
    i=low;
    h=low;
    j=mid+1;
```

```
while(h <= mid && j <= high)
{
   if(a[h] < a[j])
   {
      b[i] = a[h];
      h++;
   }
   else
   {
      b[i] =  a[j];
      j++;
   }
   i++;
}
if( h > mid )
{
   for(k=j ; k <= high ; k++)
   {
      b[i] = a[k];
      i++;
   }
}
else
```

```
    {
        for(k=h ; k<= high ; k++)
        {
            b[i] =a[k];
            i++;
        }
    }
    for(k=low ; k <= high ; k++)
        a[k] = b[k];
}

void Split(int low,int high)
{
    int mid;
    if(low<high)
    {
        mid = (low+high)/2;
        Split(low,mid);
        Split(mid+1,high);
        MergeS(low,mid,high); // Call to the function.
    }
}
void Disp2()
```

```
{
    printf("\nMerge Sorted array 'A' is as follows : ");
    for(i=0;i<No2;i++)
    {
        printf("%s\n",a[i]);
    }
}
```

Output for the Problem

```
Welcome to 'Quick' And 'Merge' types of sorting program :
Please select one of the following options :
1. Quick Sorting
2. Merge Sorting
3. Exit from menu
Please enter your choice here :
```

```
Welcome to 'Quick' And 'Merge' types of sorting program :
Please select one of the following options :
1. Quick Sorting
2. Merge Sorting
3. Exit from menu
Please enter your choice here :
1
Please enter the limit of array 'A' :
5
Please enter the strings for array A :
australia
apple
mango
russia
pripyat
Quick Sorted array 'A' is as follows :
apple
australia
mango
pripyat
russia
Press 1 to continue :
1
Welcome to 'Quick' And 'Merge' types of sorting program :
Please select one of the following options :
1. Quick Sorting
2. Merge Sorting
3. Exit from menu
Please enter your choice here :
2
Please enter the limit of array 'A' :
5
Please enter the strings for array A :
australia
apple
mango
russia
pripyat
Merge Sorted array 'A' is as follows :
apple
australia
mango
pripyat
russia
Press 1 to continue :
0
--------------------------------
```

Worst and Average case for Quick sort

WORST CASE ANALYSIS:

Now consider the case, when the pivot happened to be the least element of the array, so that

we had $k = 1$ and $n - k = n - 1$. In such a case, we have:

$T(n) = T(1) + T(n - 1) + \alpha n$

Now let us analyse the time complexity of quick sort in such a case in detail by solving the recurrence

as follows:

$T(n) = T(n - 1) + T(1) + \alpha n$

$= [T(n - 2) + T(1) + \alpha(n - 1)] + T(1) + \alpha n$

(Note: I have written $T(n - 1) = T(1) + T(n - 2) + \alpha(n - 1)$ by just substituting $n - 1$ instead

of n. Note the implicit assumption that the pivot that was chosen divided the original subarray of

size $n - 1$ into two parts: one of size $n - 2$ and the other of size 1.)

$= T(n - 2) + 2T(1) + \alpha(n - 1 + n)$ (by simplifying and grouping terms together)

$= [T(n - 3) + T(1) + \alpha(n - 2)] + 2T(1) + \alpha(n - 1 + n)$

$= T(n - 3) + 3T(1) + \alpha(n - 2 + n - 1 + n)$

$= [T(n - 4) + T(1) + \alpha(n - 3)] + 3T(1) + \alpha(n - 2 + n - 1 + n)$

$= T(n - 4) + 4T(1) + \alpha(n - 3 + n - 2 + n - 1 + n)$

$= T(n - i) + iT(1) + \alpha(n - i + 1 + ... + n - 2 + n - 1 + n)$ (Continuing likewise till the ith step.)

$= T(n - i) + iT(1) + \alpha(\Sigma j=0 \text{ to } i-1(n - j))$

Now clearly such a recurrence can only go on until $i = n - 1$

So, substitute $i = n - 1$ in the above equation, which gives us:

$T(n) = T(1) + (n - 1)T(1) + \alpha(\Sigma j=0 \text{ to } n-2(n - j)$

$= nT(1) + \alpha(n(n - 2) - (n - 2)(n - 1)/2)$

$=O(n^2)$.

AVERAGE CASE ANALYSIS

pivotIndex is O(last-first).

Suppose that we partition n elements into sub-arrays of length i and (n-i).

Time T to sort the n elements is then:

$T(n)=T(i)+T(n-i)+c \cdot n$

This kind of formula is called a recurrence relation. They are very common in describing the performance of recursive routines.

Because i is equally likely to be any value from 0 to n-1, the average (expected) value of T(i) is

$E(T(i))=1n\sum j=0n-1T(j)$

Since n-i can take on the same values as i, and all such values are equally likely,

$E(T(n-i))=E(T(i))$

On average, then $T(n)=2n(\sum j=0n-1T(j))+c \cdot n$

Multiply through by n:

$n \cdot T(n)=2 \cdot (\sum j=0n-1T(j))+c \cdot n2$

n is just a variable, so this must be true no matter what value we plug in for it. Try replacing n by n-1.

$(n-1) \cdot T(n-1)=2 \cdot (\sum j=0n-2T(j))+c \cdot (n-1)2$

So now both of these are true:

$n \cdot T(n)=2 \cdot (\sum j=0n-1T(j))+c \cdot n2(n-1) \cdot T(n-1)=2 \cdot (\sum j=0n-2T(j))+c \cdot (n-1)2$

Subtract the 2nd equation from the first:

$nT(n)-(n-1)T(n-1)=2T(n-1)+2cn-c$

Collect the T(n-1) terms together and drop the -c term:

$nT(n)=(n+1)T(n-1)+2cn$

Next we apply a standard technique for solving recurrence relations. Divide by n(n+1) and "telescope":

$T(n)n+1=T(n-1)n+2cn+1T(n-1)n=T(n-2)n-1+2cnT(n-2)n-1=T(n-3)n-2+2cn-1⋮$

Note that most of the terms on the left will have appeared on the right in the previous equation, so if we were to add up all these equations, these terms would appear on both sides and could be dropped:

$T(n)n+1=T(1)2+2c\sum j=3n+11j$

As n gets very large, $\sum j=3n+11j$ approaches $\ln(n)+\gamma$, where γ is Euler's constant, 0.577...

Hence

$T(n)n+1=T(1)2+2c\cdot\ln(n)+2c\gamma=\ln(n)+c2=O(\ln(n))$

and so $T(n)=O(n\cdot\log(n))$

Worst and Average case for Merge sort

```
void MergeS(int low,int mid,int high)
{
   int i,h,j,k;
   i=low;
   h=low;
   j=mid+1;
   while(h <= mid && j <= high)
   {
      if(a[h] < a[j])
      {
         b[i] = a[h];
         h++;
      }
      else
      {
         b[i] =  a[j];
         j++;
      }
      i++;
```

```
    }
    if( h > mid )
    {
        for(k=j ; k <= high ; k++)
        {
            b[i] = a[k];
            i++;
        }
    }
    else
    {
        for(k=h ; k<= high ; k++)
        {
            b[i] =a[k];
            i++;
        }
    }
    for(k=low ; k <= high ; k++)
        a[k] = b[k];
}

void Split(int low,int high)
{
```

```
    int mid;
    if(low<high)
    {
        mid = (low+high)/2;
        Split(low,mid);
        Split(mid+1,high);
        MergeS(low,mid,high); // Call to the function.
    }
}
```

The time taken for one step is log (n), and since n steps are there, the worst and average cases for merge sort is O (nlog (n)).

Quick sort vs merge sort

1. **Partition of elements in the array** :
   In the merge sort, the array is parted into just 2 halves (i.e. n/2).
   whereas
   In case of quick sort, the array is parted into any ratio. There is no compulsion of dividing the array of elements into equal parts in quick sort.
2. **Worst case complexity** :
   The worst case complexity of quick sort is O(n2) as there is need of lot of comparisons in the worst condition.
   whereas
   In merge sort, worst case and average case has same complexities O(n log n).
3. **Usage with datasets** :
   Merge sort can work well on any type of data sets irrespective of its size (either large or small).
   whereas
   The quick sort cannot work well with large datasets.
4. **Additional storage space requirement** :
   Merge sort is not in place because it requires additional memory space to store the auxiliary arrays.
   whereas
   The quick sort is in place as it doesn't require any additional storage.
5. **Efficiency** :
   Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets.
   whereas
   Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets.
6. **Sorting method** :
   The quick sort is internal sorting method where the data is sorted in main memory.
   whereas
   The merge sort is external sorting method in which the data that is to be sorted cannot be accommodated in the memory and needed auxiliary memory for sorting.
7. **Stability** :
   Merge sort is stable as two elements with equal value appear in the same order in sorted output as they were in the input unsorted array.
   whereas
   Quick sort is unstable in this scenario. But it can be made stable using some changes in code.
8. **Preferred for** :
   Quick sort is preferred for arrays.
   whereas
   Merge sort is preferred for linked lists.
9. **Locality of reference** :
   Quicksort exhibits good cache locality and this makes quicksort faster than merge sort (in many cases like in virtual memory environment).

CONCLUSION

Selection of pivot is very important because depending on the selection of pivot we get worst case or best case time complexity.

If the pivot is selected as the first element or the last element from the array we will get time complexity of $O(n^2)$.

Quick sort is faster than Merge sort.

Unlike Merge sort, this algorithm does not need extra linear memory and extra work for copying to the temporary array and back.

The algorithm can be used for sorting data in the computer memory (this algorithm is called internal sorting).