IDAA432C Assignment-2 (Group: 41)

Aditya Vallabh	Kiran Velumuri	Neil Syiemlieh	Tauhid Alam
IIT2016517	ITM2016009	IIT2016125	BIM2015003

Question

Assume a given set of unsorted numbers. You have to perform sorting. However, print the sorted numbers along with their original (initial) locations in the given array. Locate a key in the sorted array. Do the necessary experimentation and analysis with your algorithm.

I. INTRODUCTION

To solve this question, we first have to generate a set of random numbers. Then, we have to perform sorting. We have used the merge sort algorithm to sort the numbers. Next, we have to print the sorted numbers along with their original locations in the array.

After sorting, we have to search for a key in the sorted array. For this, we have used binary search algorithm, which on successful search, returns the position of the search key in the unsorted as well as the sorted array.

Using the relevant test cases, we obtain the time complexity of the algorithms for each of best, average and worst cases. Then we plot the time versus n graphs for the above mentioned cases.

II. ALGORITHM DESIGN

The merge sort algorithm divides the array into two halves until we reach the base case, that is, array of size 1 and recursively calls itself for the two halves. After that, the $merge()$ function takes the sorted sub-arrays and merges them to gradually sort the entire array.

Since we are also required to display the original positions of the elements in the sorted array, we have modified the merge sort algorithm by changing the one-dimensional array input into a two-dimensional array, arr. $arr[\_][1]$ contains the elements of the input array while $arr[\_][0]$ contains the corresponding indices of these elements.

The working of the algorithm is explained below:

1) Initialize the left l and right r extremities of the array to the values 0 and $n-1$ respectively before calling $mergesort()$ for the first time.
2) If the array has at least two elements, proceed. Else, stop the current recursive call.
3) Find the middle element m to divide the array into two halves.
4) Now, call $mergesort()$ for the first half of the array.
5) Call $mergesort()$ for the second half of the array.
6) Merge the two halves sorted in the previous steps by calling the $merge()$ function.

Algorithm 1 Merge Sort Algorithm

Input: arr, l, r
if $l < r$ **then**
  $m \leftarrow (l + r)/2$
  $mergeSort(arr, l, m)$
  $mereSort(arr, m + 1, r)$
  $merge(arr, l, m, r)$

The most important part of merge sort is the $merge()$ function which merges the two sorted sub-arrays which are indexed from l to m and $m + 1$ to r, to form the final sorted array. The following steps are involved in the $merge()$ function:

1) Copy the values of elements in the two sub-arrays to the arrays L and R which contain $n1$ and $n2$ elements respectively. Here $n1 = m - l + 1$ and $n2 = r - m$.
2) Use index variables i, j and k to store the position of current element in the arrays L, R and the original array arr, respectively.
3) If $L[i][1]$ is less than or equal to $R[j][1]$, then copy the value of $L[i][0]$ to $arr[k][0]$ and $L[i][1]$ to $arr[k][1]$. Increment i and k.
4) If $L[i][1]$ is greater than $R[j][1]$, then copy the value of $R[j][0]$ to $arr[k][0]$ and $R[j][1]$ to $arr[k][1]$. Increment j and k.
5) Go back to step 3 while both $i < n1$ and $j < n2$
6) Copy the remaining elements left, if any, in the two sub-arrays to arr by incrementing the corresponding index variables.

Algorithm 2 Merge Algorithm

Input: arr, l, m, r
$n1 \leftarrow m - l + 1$
$n2 \leftarrow r - m$
for $i \leftarrow 0$ *to* $n1 - 1$ **do**
  $L[i][0] \leftarrow arr[l + i][0]$
  $L[i][1] \leftarrow arr[l + i][1]$
for $j \leftarrow 0$ *to* $n2 - 1$ **do**
  $R[j][0] \leftarrow arr[m + 1 + j][0]$
  $R[j][1] \leftarrow arr[m + 1 + j][1]$
$i \leftarrow 0$
$j \leftarrow 0$
$k \leftarrow l$
while $i < n1$ AND $j < n2$ **do**
  if $L[i][1] \leq R[j][1]$ **then**
    $arr[k][0] \leftarrow L[i][0]$
    $arr[k][1] \leftarrow L[i][1]$
    $i \leftarrow i + 1$
  else
    $arr[k][0] \leftarrow R[j][0]$
    $arr[k][1] \leftarrow R[j][1]$
    $j \leftarrow j + 1$
  $k \leftarrow k + 1$
while $i < n1$ **do**
  $arr[k][0] \leftarrow L[i][0]$
  $arr[k][1] \leftarrow L[i][1]$
  $i \leftarrow i + 1$
  $k \leftarrow k + 1$
while $j < n2$ **do**
  $arr[k][0] \leftarrow R[j][0]$
  $arr[k][1] \leftarrow R[j][1]$
  $j \leftarrow j + 1$
  $k \leftarrow k + 1$

A. Finding the given key in the sorted array

Now that the array is sorted, we have to search for a particular element in the sorted array. We have used binary search for this purpose, which works as follows:

1) Initialize left l and right r extremities of the array to 0 and $n - 1$ respectively.
2) Find the middle element m in the array which is at $(l + r)/2th$ position - $arr[m][1]$.
3) If the middle element is equal to the search key, return its index m.
4) If the middle element is less than the search key, call binary search for the second half of the array by updating the value of l to $m + 1$.
5) If the middle element is greater than the search key, call binary search for the first half of the array by updating the value of r to $m - 1$;
6) So, if the search key is not present in the array, return -1 indicating that it is not present in the array.

Algorithm 3 Binary Search

Input: arr, n, key
$l \leftarrow 0$
$r \leftarrow n - 1$
while $l \leq r$ **do**
  $m \leftarrow (l + r)/2$
  if $arr[m][1] == key$ **then**
    $return\ m$
  else if $arr[m][1] < key$ **then**
    $l \leftarrow m + 1$
  else
    $r \leftarrow m - 1$
$return - 1$

III. ANALYSIS

A. SORTING AN ARRAY OF NUMBERS

Merge sort can be described by the following recurrence relation, where $T(n)$ is the time taken to perform the computation on an array of size n.

$$T(n) = 2\ T(\frac{n}{2}) + O(n)$$
$$T(1) = O(1)$$

Here, the $T(\frac{n}{2})$ is for the recursive step and the $O(n)$ is for combining the sorted subarrays. We can solve the recurrence relation as follows:

$$\begin{aligned} T(n) &= 2\ T(\frac{n}{2}) + n \\ &= 2\ [2\ T(\frac{n}{4}) + \frac{n}{2}] + n \\ &= 4\ T(\frac{n}{4}) + 2n \\ &= 4\ [2\ T(\frac{n}{8}) + \frac{n}{4}] + 2n \\ &\vdots \\ T(n) &= 2^k\ T(\frac{n}{2^k}) + kn \end{aligned}$$

For $T(1)$:

$$\begin{aligned} &\implies \frac{n}{2^k} = 1 \\ &\implies k = \log_2 n \end{aligned}$$

Entering the value of k in the equation above, we obtain the solution:

$$T(n) = n\ (1 + \log_2 n)$$

The time complexity of the solution is of the order of $n \log n$. This is the case for the best, average and worst cases, because the algorithm will never terminate permaturely no matter the inputs.

B. BINARY SEARCH

The average and worst case time taken to perform binary search on an array of size n be described by:

$$T(n) = T(\frac{n}{2}) + O(1)$$
$$T(1) = O(1)$$

Here, $T(\frac{n}{2})$ is for the recursive step and O(1) is for comparing the key to the middle element. Solving the recurrence relation gives us:

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 1 \\ &= [T(\frac{n}{4}) + 1] + 1 \\ &= T(\frac{n}{4}) + 2 \\ &\vdots \\ T(n) &= T(\frac{n}{2^k}) + k \end{aligned}$$

For $T(1)$:

$$\begin{aligned} &\implies \frac{n}{2^k} = 1 \\ &\implies k = \log_2 n \end{aligned}$$

Entering the value of k in the equation above, we obtain the solution:

$$T(n) = \log_2 n$$

The time complexity of the solution is of the order of $\log n$. This is for the worst and average cases. In the best case, the key is equal to the middle element at the very first try, giving us $\Omega(1)$.

IV. EXPERIMENTAL STUDY

Following are the experimental findings after profiling the data and plotting the relevant graphs using *gnuplot*.

1) Fig. 1 gives the complexity for the Merge Sort algorithm.
2) Fig. 2 gives the complexity for the Binary Search algorithm. It can be seen that the theoretical calculations agree with the experimental data as the average case turns out to be a logarithmic graph.

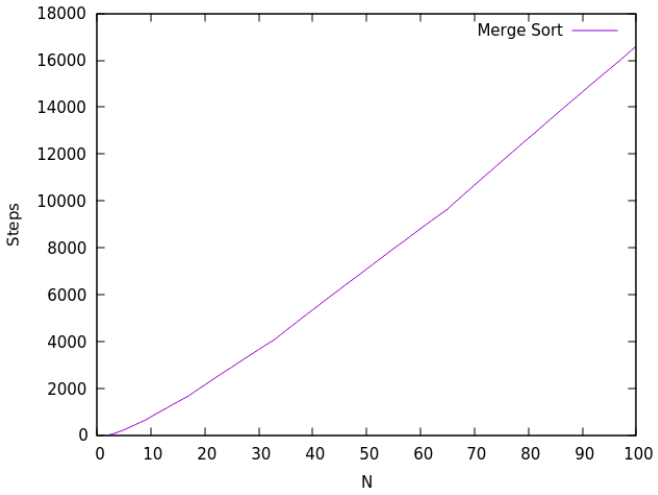

Fig. 1. Time Complexity for Merge Sort

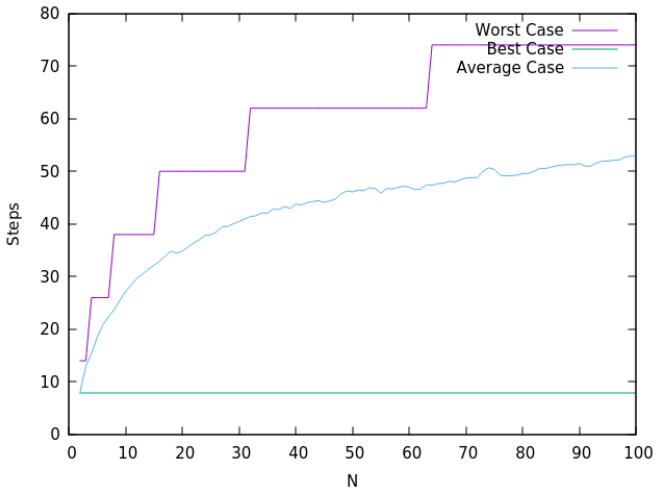

Fig. 2. Time Complexity for Binary Search

V. DISCUSSION

A. Generation of random numbers

We have used the $rand()$, $srand()$ functions and the $< time.h >$ header file to fill the array with random numbers. The $rand()$ function generates a pseudo random number, that is, the same random number is generated every time we compile and run the program. So, we used the $srand()$ function to set the seed for the $rand()$ function.

In this question, we have used $time(0)$ as the seed. The call to $time()$ returns a $time\_t$ value which varies every time and hence the pseudo random number varies for every program call. The $time\_t$ is a variable type defined in the $< time.h >$ header which is suitable for

storing the calendar time.

B. Divide and conquer strategy

Merge sort is an example of divide and conquer algorithm, which solves the problem by dividing it into several subproblems. When the solution to each of the subproblem is ready, we combine them to get the solution to the main problem.

Likewise in merge sort, we repeatedly divide the array until we try to perform merge sort on sub-arrays of size 1. Then, the merge function combines the sorted arrays into larger ones until the complete array is merged.

VI. Conclusion

We have implemented the merge sort algorithm, which has $\Theta(n * log(n))$ time complexity in all the cases, to sort the given set of unsorted numbers. For finding a key in the sorted array, binary search has been used, which has a time complexity of $\Omega(1)$ in the best case and $O(logn)$ in the worst case.