IDAA432C

Assignment 2

By:

- Neil Syiemlieh
- Aditya Vallabh
- Kiran Velumuri
- Tauhid Alam

Abstract

A sorted array of numbers is a sequence of numbers arranged in either ascending or descending order. The purpose of this project is to:

- Sort a given array of numbers and keep track of the original location of its elements
- Find the location of a key in the sorted array

To generate random numbers:

- rand function used to populate the array

To keep track of elements' original location:

- n x 2 matrix as the input array, where

● 1st column - indices of elements
● 2nd column - the elements themselves

Algorithm Design

Merge Sort

Steps:

- If input array has at least two elements, proceed
- Find the middle element of the array
- Call merge sort on the first half of the array
- Call merge sort on the second half of the array
- Merge the two halves by calling the merge function

Algorithm 1 Merge Sort Algorithm

Input: arr, l, r

if $l < r$ **then**

$m \leftarrow (l + r)/2$

$mergeSort(arr, l, m)$

$mereSort(arr, m + 1, r)$

$merge(arr, l, m, r)$

Merge

Steps:

- Create two temporary arrays *L* and *R*
- Copy the left half of input array to *L* and right half to *R*
- Replace the input array by elements of *L* and *R* element-wise, in order (merge step)
- Copy all the remaining element of either *L* or *R* into the input array

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
    $R[j][0] \leftarrow arr[m + 1 + j][0]$
    $R[j][1] \leftarrow arr[m + 1 + j][1]$
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
    $arr[k][0] \leftarrow R[j][0]$
    $arr[k][1] \leftarrow R[j][1]$
    $j \leftarrow j + 1$
    $k \leftarrow k + 1$
```

Binary Search

Steps:

- If input array has at least one element, proceed. Else, return -1 to say that the key doesn’t exist in the array
- Find the middle element of the array
- If middle element equal to key, return middle element’s location
- If middle element less than key, call binary search on right half of array
- If middle element greater than key, call binary search on left half of 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

    $r \leftarrow m - 1$

$return - 1$

Complexity Analysis

A. Merge Sort

Computation time recurrence relation:

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

Deriving the time expression

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

Through the derivation on the left, we obtain the expression for the computation time below:

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

Time Complexity

Time complexity is the same for all cases because the algorithm never terminates prematurely.

$O(n \log n)$	$\Theta(n \log n)$	$\Omega(n \log n)$
Worst	Average	Best

B. Binary Search

Computation time recurrence relation:

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

$$T(1) = O(1)$$

Deriving the time expression

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

Through the derivation on the left, we obtain the expression for the computation time below:

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

Time Complexity

Average and worst case are the same. Best case happens when the key is equal to the middle element of the input array.

$$O(\log n)$$

Worst

$$\Theta(\log n)$$

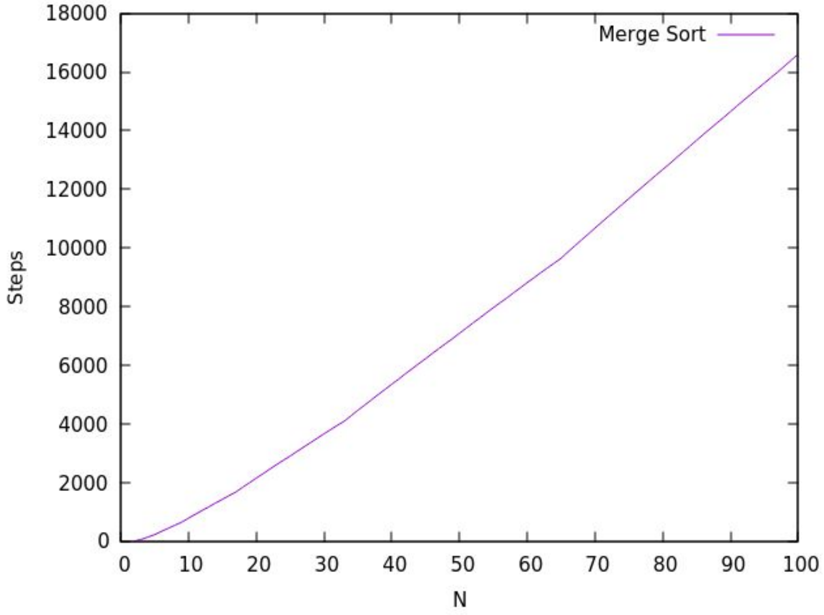
Average

$$\Omega(1)$$

Best

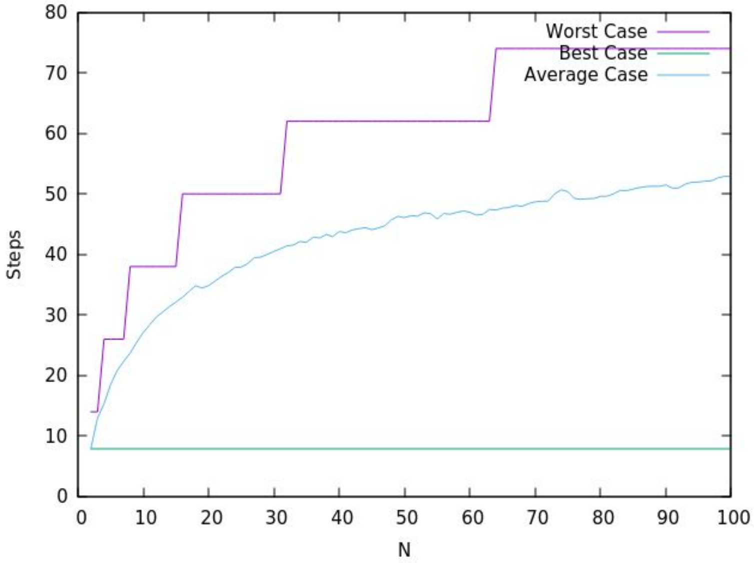
Experimental Study

Merge Sort Complexity



$$\Theta(n \log n)$$

Binary Search Complexity



Best: $\Omega(1)$

Average: $\Theta(\log n)$

Worst: $O(\log n)$

Discussions

Comparison

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)
Bubble Sort	O(n)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)

Generation of random numbers

- Dynamic allocation of memory for creating and storing matrices
- Used rand() and srand() from stdlib
- srand() sets the seed which is used by rand to generate “random” numbers
- Setting the seed as current time to produce different pseudo-random numbers on each run

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 $O(\log n)$ in the worst case.


Thank you