IDAA432C

Assignment 3

By:

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

Question

Given a set of numbers, find the smallest and largest numbers in the list. Swap the smallest number with the first and the largest number with the last. Then do the same with the 2nd smallest and largest and so on until all elements are sorted. Track the movements of all the elements in the list while sorting.

Algorithm Design

Sorting

Basically selection sort, except we find two elements at every iteration - smallest and largest.

And then we swap the smallest with the first, and the largest with the last.

Steps:

- For every subarray from index *l* to *r*:
- Find the smallest element
- Swap with the first element
- Find the largest element
- Swap with the last element
- Increment *l* and decrement *r*

Algorithm 1 Sort Algorithm

Input: $arr, heads, n$

for $i \leftarrow 0\ to\ \frac{n}{2} - 1$ **do**

  $mini = getMinIdx(arr, n, i)$

  if $mini \neq i$ **then**

    $swap(arr, heads, i + 1, mini, i)$

  $maxi = getMaxIdx(arr, n, i)$

  if $maxi \neq n - i - 1$ **then**

    $swap(arr, heads, i + 1, maxi, n - i - 1)$

Finding smallest and largest elements

Assume the first element is the smallest/largest, and store it. Then traverse the array.

During traversal, if the current element is smaller/larger than the stored one, update that stored one to the current element.

Algorithm 3 getMinIdx Algorithm

Input: arr, n, i

$idx \leftarrow i$

$min \leftarrow arr[i]$

for $j \leftarrow i + 1$ to $n - i - 1$ **do**

  if $arr[j] < min$ **then**

    $min \leftarrow arr[j]$

    $idx \leftarrow j$

$return\ idx$

Algorithm 4 getMaxIdx Algorithm

Input: arr, n, i

$idx \leftarrow i$

$max \leftarrow arr[n - i - 1]$

for $j \leftarrow i$ to $n - i - 2$ **do**

  if $arr[j] \geq max$ **then**

    $max \leftarrow arr[j]$

    $idx \leftarrow j$

$return\ idx$

Tracking Movement

- Associate a linked list with every array element.
- Head of linked list sits in another array parallel to the input array.
- When elements are swapped, swap their heads too.
- Update both elements' linked lists with:
  - New location
  - Iteration at which swap happened

Algorithm 2 Swap Algorithm

Input: $arr, heads, it, a, b$

$heads[a] \leftarrow insertNode(heads[a], it, b)$

$heads[b] \leftarrow insertNode(heads[a], it, a)$

$tmp \leftarrow arr[a]$

$arr[a] \leftarrow arr[b]$

$arr[b] \leftarrow tmp$

$temp \leftarrow heads[a]$

$heads[a] \leftarrow heads[b]$

$heads[b] \leftarrow temp$

Complexity Analysis

Computation time:

$$
\begin{aligned}
T(n) &= 2(O(n) + O(n-2) + \ldots + O(2)) \\
&= 2\sum_{k=0}^{\frac{n}{2}-1} O(n-2k) \\
&= \frac{n^2+2n}{2}
\end{aligned}
$$

Time Complexity

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

$O(n^2)$	$\Theta(n^2)$	$\Omega(n^2)$
Worst	Average	Best

Space consumed

Since a new node is created for location tracking only when elements are swapped, the space complexity is directly proportional to the number of swaps.

Best Case:

All elements are already sorted. No swapping needed.

Space consumed = n

Worst Case:

Maximum number of swaps that can occur is *(n-1)*

Space consumed = n + 2*(n - 1)

Average Case:

Close to worst case, since random elements are likely unsorted

Space Complexity

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

Experimental Study

Time Complexity

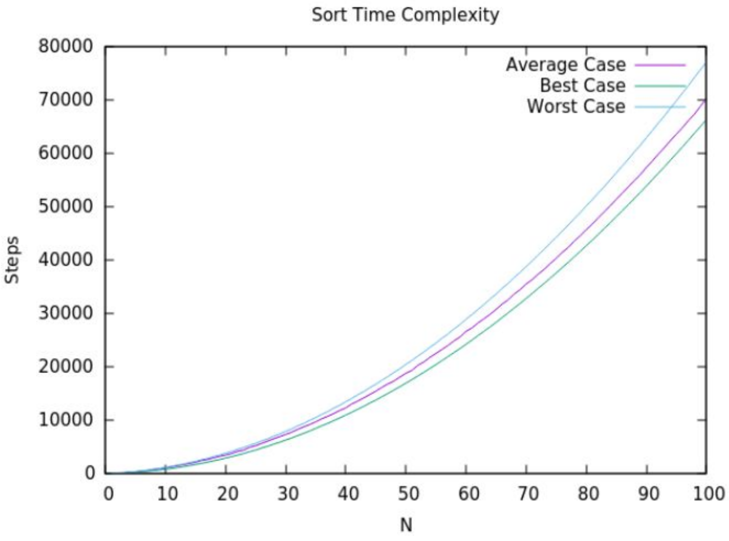


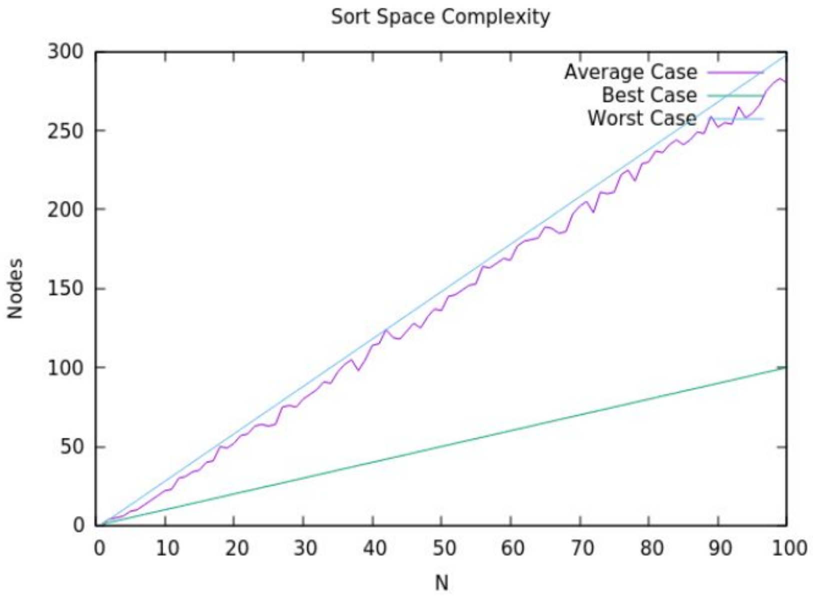
Fig. 1. Time Complexity for the Sorting Algorithm

Best: $\Omega(n^2)$

Average: $\Theta(n^2)$

Worst: $O(n^2)$

Space Complexity



Best: $\Omega(n)$

Average: $\Theta(n)$

Worst: $O(n)$

Discussions

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

Sort Stability

- *getMinIdx* and *getMaxIdx* determine sort stability
- *(arr[j] < min)* in *getMinIdx* returns 1st occurrence of smallest element
- *(arr[j] >= max)* in *getMaxIdx* returns last occurrence of largest element
- Makes the sort stable

Conclusion

We have implemented the sort algorithm, which has $\Theta(n^2)$ time complexity in all the cases, to sort the given set of unsorted numbers.

For tracking the movement of the elements, linked lists have been used, which has a the space complexity of $\Theta(n)$ in all cases.


Thank you