HW0

Ronan Wallace

February 14, 2019

1 Algorithm Design

```
ALGORITHM    SecondLargest(A[0,...,n-1])
    //Finds the second largest number in the array when $n \geq 2$
    $//Input : AnarrayA[0, ..., n-1]ofdistinctintegers$
    $//Output : Secondlargestintegerofgivenarray$
    $A \leftarrow [0, ..., n-1]$ (array of distinct numbers given by the user)
    if $n \leftarrow 2$ do (if array contains only two integers)
        $initValue \leftarrow A[0]$
        if $initValue > A[1]$ do
            return $A[1]$
        else
            return $initValue$
    $firstLargest \leftarrow A[0]$
    for $i \leftarrow 0$ to $n-1$ do (arrays contains n+2 integers)
        if $firstLargest < A[i]$
            $firstLargest \leftarrow A[i]$
    $secondLargest \leftarrow A[0]$
    for $i \leftarrow 0$ to $n-1$ do
        if $secondLargest \neq firstLargest$ && $secondLargest < A[i]$
            $secondLargest \leftarrow A[i]$
    return $secondLargest$
```

EXAMPLE

//**Sample: [1,0,5,2]**

//n is greater than 2, therefore, surpasses initial **if** statement

//When passing through the first **for** loop, the largest integer from the list is assigned to *firstLargest* through comparisons. In this case, it will be 5

//In the next **for** loop, the magnitude of each integer will be compared like in the first loop. Using the *firstLargest* variable, the algorithm avoids assigning the *secondLargest* variable with the largest integer. After going through the entire array, *secondLargest* will be returned. In this case, 2 will be returned

//**Sample: [9,4]**

//n is equal to 2, therefore, is caught by the first **if** statement

//First integer is assigned to *initValue*
//The first value (*initValue*) is compared to the second value at index position 1. **If** *initValue* is greater than compared integer, *initValue* is returned. **If** it is less, then the compared integer at index position 1 will be returned and the algorithm will terminate

2 Weighted, Directed Graph

Adjacency Matrix

$$\begin{array}{cccccc} & a & b & c & d & e \\ a & \infty & \infty & \infty & 9 & \infty \\ b & 39 & \infty & 11 & \infty & \infty \\ c & 12 & \infty & \infty & \infty & 5 \\ d & \infty & \infty & \infty & \infty & \infty \\ e & \infty & 4 & 2 & \infty & \infty \end{array}$$

Or

$$\begin{bmatrix} \infty & \infty & \infty & 9 & \infty \\ 39 & \infty & 11 & \infty & \infty \\ 12 & \infty & \infty & \infty & 5 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 4 & 2 & \infty & \infty \end{bmatrix}$$

Tried very hard to get the brackets and the column/row labels to cooperate, but could only get either one or the other for the adjacency matrix.

Adjacency List

$a \to d, 9$
$b \to a, 39 \to c, 11$
$c \to a, 12 \to e, 5$
$d \to null$
$e \to b, 4 \to c, 2$

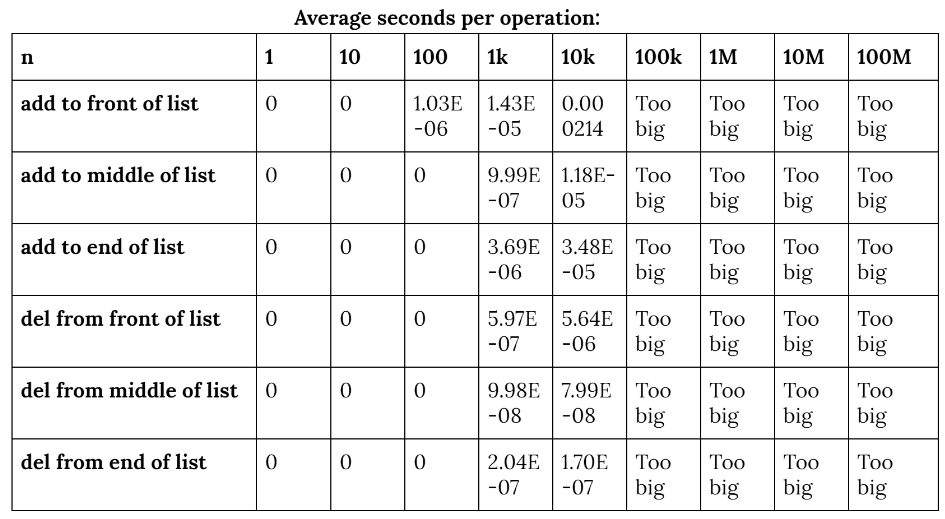
3 Applied Data Structures

a.) For this situation, a queue would be used. It operates in a "first-in-first-out" fashion. This gives priority to the first phone call at the service center. The queue will give the first caller priority and those after will be placed on hold until the customers ahead of them have been helped.

b.) In a backlog situation, a queue would also be used (or a stack, will be explained further) again because of the specification of how it's received will be how it's sent (FIFO: first received, first to be sent back to the customers). It is possible that I don't fully understand a backlog order. If it is, for example, a stack of orders on someone's desk, the stack data structure will be used because someone wouldn't pull a document from the bottom of the stack (like the stack of plates example).

c.) At first, I was thinking of using a stack data structure for the calculator, given that the user would input numbers from left to right and be placed into the stack and then operated on by popping them from the stack. However, I think a priority queue would be more useful in the sense that priority would be given to multiplication and division because of order of operation.

4 Program Analysis



Average seconds per operation:

n	1	10	100	1k	10k	100k	1M	10M	100M
add to front of list	0	0	1.03E-06	1.43E-05	0.000214	Too big	Too big	Too big	Too big
add to middle of list	0	0	0	9.99E-07	1.18E-05	Too big	Too big	Too big	Too big
add to end of list	0	0	0	3.69E-06	3.48E-05	Too big	Too big	Too big	Too big
del from front of list	0	0	0	5.97E-07	5.64E-06	Too big	Too big	Too big	Too big
del from middle of list	0	0	0	9.98E-08	7.99E-08	Too big	Too big	Too big	Too big
del from end of list	0	0	0	2.04E-07	1.70E-07	Too big	Too big	Too big	Too big

Figure 1: Average Seconds per Operation Table

ANALYSIS

When n is equal to 1 or 10, the operations run instantly (0 seconds). When n is 100, operations run instantly except for adding to the front of the list, which runs in 1.03E-06 seconds. When n reaches 1k, seconds per operation is more than zero, meaning that the processing speed is increasing. However, when n reaches 100k, the operations take too long to process (making them "Too big"). The slowest function is manipulating the list by adding to the front of the list. The second most efficient function is deleting from the middle of the list. I expected adding to the front of the list would take the longest because in order to add to the front, the function must shift all of the elements of the list + 1 index position. I expected adding to the end of the list would be faster than adding to the middle, which is initially the case. However, when n is equal to 10k, adding to the middle becomes on average slightly faster by 2.3E-05 seconds. I am not entirely sure why this would somehow become faster (one possibility is that an operation gave an outlier average operation speed during one of the repetitions that was a lot faster than the others, decreasing the overall average seconds per operation). If the average seconds per operation could be calculated quickly when n greater than or equal to 100k, we would be able to see the trend of the average seconds per operation more clearly.