Assignment 2 - Hady Ibrahim
SFWRENG 2CO3: Data Structures and Algorithms–Winter 2023

Deadline: February 12, 2023

Department of Computing and Software
McMaster University

Please read the *Course Outline* for the general policies related to assignments.

Plagiarism is a *serious academic offense* and will be handled accordingly.
All suspicions will be reported to the *Office of Academic Integrity*
(in accordance with the Academic Integrity Policy).

This assignment is an *individual* assignment: do not submit work of others. All parts of your submission *must* be your own work and be based on your own ideas and conclusions. Only *discuss or share* any parts of your submissions with your TA or instructor. You are *responsible for protecting* your work: you are strongly advised to password-protect and lock your electronic devices (e.g., laptop) and to not share your logins with partners or friends! If you *submit* work, then you are certifying that you have completed the work for this assignment by yourself. By submitting work, you agree to automated and manual plagiarism checking of all submitted work.

Late submission policy. Late submissions will receive a late penalty of 20% on the score per day late (with a five hour grace period on the first day, e.g., to deal with technical issues) and submissions five days (or more) past the due date are not accepted. In case of technical issues while submitting, contact the instructor *before* the deadline.

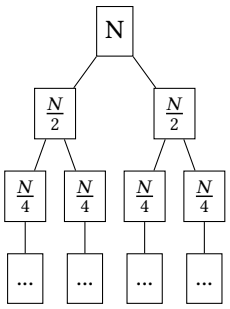
Problem 1. Assume we have a computer with infinite processor cores that can all operate at the same time. In an attempt to sort *as fast as possible*, we make a *parallelized* version of MergeSort that uses multiple processor cores at the same time. To achieve this, we change the top-down merge-sort algorithm as follows:

Algorithm ParallelMergeSort($L[start \ldots end]$)**:**

1: **if** $start + 1 < end$ **then**
2: $mid := (start + end)$ div 2.
3: Start ParallelMergeSort($L[start, mid]$) on a fresh processor core C.
4: $L_2 :=$ ParallelMergeSort($L[mid \ldots end]$).
5: Wait until C finished ParallelMergeSort($L[start, mid]$), resulting in L_1.
6: **return** Merge(L_1, L_2).
7: **else**
8: **return** L.
9: **end if**

Let $n = |L|$ be the length of the list being sorted by ParallelMergeSort.

P1.1. Give a recurrence $T(n)$ for the runtime complexity of ParallelMergeSort and solve the recurrence $T(n)$ by proving that $T(n) \sim e(n)$ for some expression e that uses n.
Answer:



Number	Cost	Total
1	N	$1 * N$
1	$\frac{N}{2}$	$1 * \frac{N}{2}$
1	$\frac{N}{4}$	$1 * \frac{N}{4}$
...	...	...
1	$\frac{N}{2^i}$	$1 * \frac{N}{2^i}$
—	—	—
		$2N$

Note: the number, cost and total is meant to line up side by side with the tree. So cost N is the root of the tree, cost $\frac{N}{2}$ is the second level and so on.

As you can see, the tree shows that each step has $\frac{N}{2^i}$ cost. There is only 1 counted as they all occur in parallel, so unlike the cost complexity, the time complexity is as if the step is just run once. When you go to sum up all the steps, you can see that the steps will converge to 2N which is ~ N. Note: $N + \frac{N}{2} + \frac{N}{4} + \frac{N}{8} + ... = 2N$. Given N is not always a power of 2, we know that $O(N)$ is upper bounded by 2N and it must be lower bounded by N since the first step will always have a cost of N. This is enough to prove that the algorithm is $\sim N$.

P1.2. Provide a strict bound on the number of processor cores that ParallelMergeSort uses (hence, how many different processor cores are used by ParallelMergeSort at the same time).
Answer:
Assumption: When a core calls two parallel MergeSorts, the core will handle one and delegate the other call to a new core.

Assume N is a power of 2. The number of processors is $2^{\log_2 N} = N$. This is because the recurrence tree proves to be $\log_2 N$ height and each step has double the processors of the last step. On the last recurrence level, it will terminate as there is only 1 element in the List, so it will hit the else statement (rather than the if condition) and return the List. Given N is not always a power of 2, we know the number of processors is in the range of "the nearest power of 2 that's smaller than N" <= $processors$ <= N, which contains the min and max number of processors needed per level on the recurrence tree.

Problem 2. Consider the following recursive sorting algorithm.

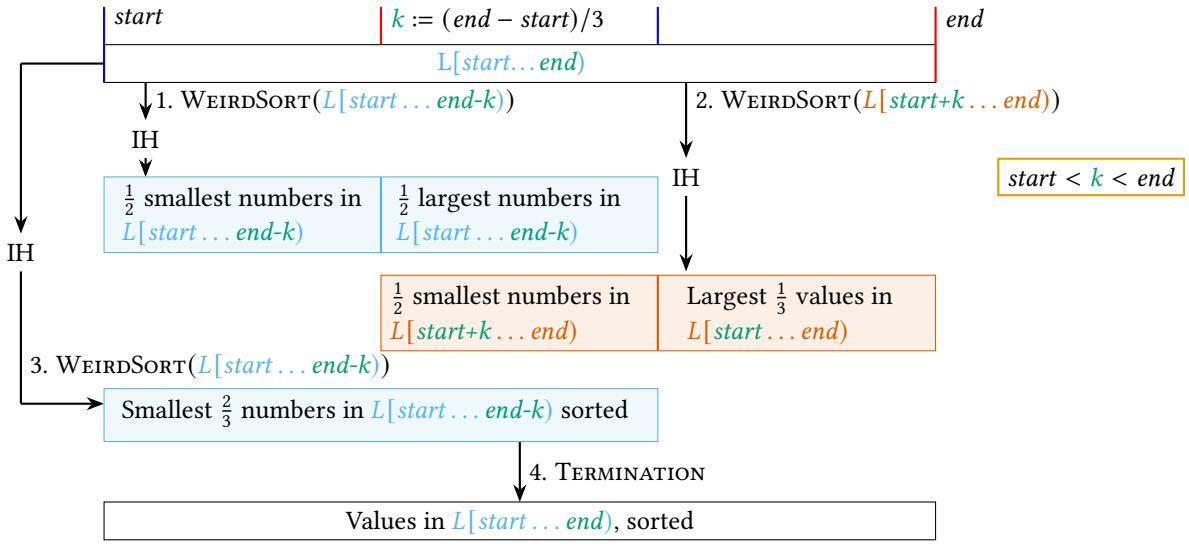
Algorithm WeirdSort(L, $start$, end)**:**

```
1:  if end − start = 2 then
2:     if L[start] ≥ L[start + 1] then
3:        Exchange L[start] and L[start + 1].
4:     end if
5:  else if end − start > 2 then
6:     k := (end − start) div 3.
7:     WeirdSort(L, start, end − k).
8:     WeirdSort(L, start + k, end).
9:     WeirdSort(L, start, end − k).
10: end if
```

Let $n = |L|$ be the length of the list being sorted by WeirdSort.

P2.1. Is WeirdSort a *stable* sort algorithm? If yes, explain why. If no, show why not and indicate whether the algorithm can be made stable.
Answer:
This is not a stable sort algorithm as in line 2, if the earlier element is equal to the next element, they get swapped. To make this a stable algorithm, change ≥ to >.

P2.2. Prove via induction that WeirdSort will sort list L.
Answer:



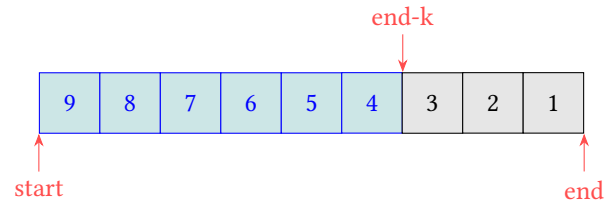
Base case 1: $0 <= end - start = 2$ values
In this case, it will hit the first if statement and sort the 2 values based on which is largest, then return. There is no recursion in this case.

Base case 2: $0 <= end - start = 3$ values
This is the lowest amount of values where there will be recursion. This will result in $k = (3-0)/3 = 1$
Induction Hypothesis: WeirdSort sorts $0 <= end - start < n$ values correct.
Induction Step: Prove WeirdSort sorts $end - start = n$ values

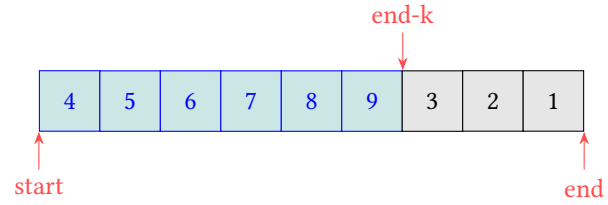
To start, assume $|L| = 3n, n \in \mathbb{N}, n > 0$ ($|L|$ is a multiple of 3). k will always give us $\frac{1}{3}|L|$, so each WeirdSort call will be $\frac{2}{3}$ of L. Since our base case 2 shows that $k >= 1$ always, this means the lists drawn for the 3 WeirdSort calls will be at least 1 smaller than the input array. Given this, we can see that, assuming WeirdSort can sort $< n$ values via IH, WeirdSort call 1 will sort the largest $\frac{1}{3}$ values of L into the middle third of L (second half of L in WeirdSort call 1). WeirdSort call 2 will then take those largest numbers sorted in the first call and sort them with the remaining $\frac{1}{3}$ values. By the end of this call, the largest $\frac{1}{3}$ values in L will be sorted at the end of the list. WeirdSort call 3 will then sort the last (and smallest) $\frac{2}{3}$ values.

The following is looking at the case where the largest values are at the start of the list and the smallest at the end (assume reverse ordered list). You can see that there are enough of the largest values moved to the end of the first WeirdSort call, so that they can be sorted to the end of the list. This is because as seen by the final WeirdSort call, you only need to move the $\frac{1}{3}$ largest values to the last $\frac{2}{3}$ of the list, in order for them to be sorted to the end of the list. The final WeirdSort call will handle the rest.

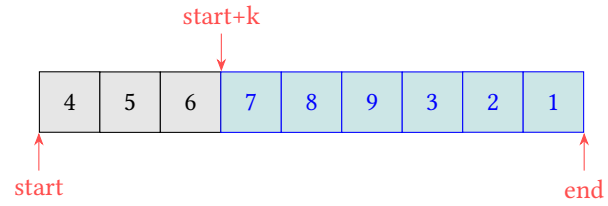
Start of WeirdSort call 1



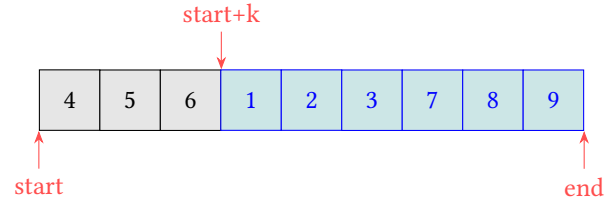
End of WeirdSort call 1



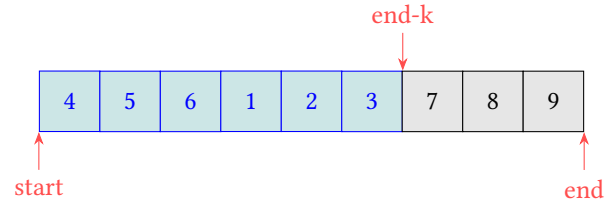
Start of WeirdSort call 2



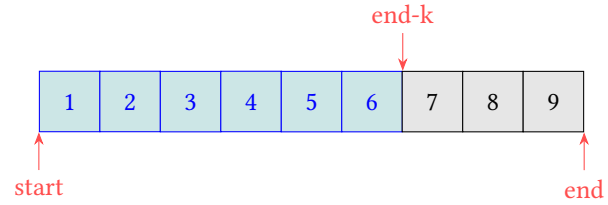
End of WeirdSort call 2



Start of WeirdSort call 3



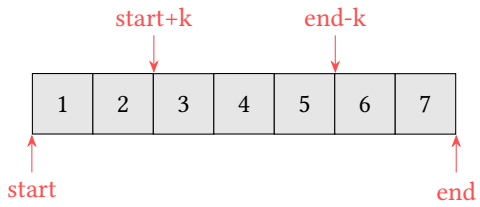
End of WeirdSort call 3



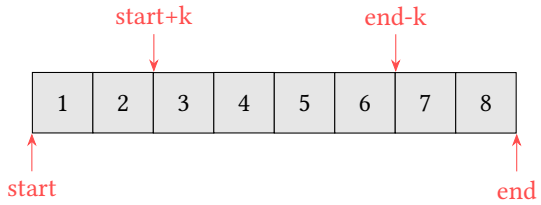
Now given that $|L|$ is not a multiple of 3, there are two cases: $|L| = 3n + 1$ and $|L| = 3n + 2$. For both of these cases, there will be more values pushed to the end of the list in WeirdSort call 1, for WerirdSort call 2 to move to the end of the list. This is because k always rounds down, so the subsequent lists from

WeirdSort calls will be bigger. More formally, for this to occur $|L[start+k, end-k)| >= L[end-k, end)$ An example can be seen below given

$|L| = 3n + 1$



$|L| = 3n + 2$



P2.3. Give a recurrence $T(n)$ for the runtime complexity of WeirdSort and solve the recurrence $T(n)$ by proving that $T(n) \sim e(n)$ for some expression e that uses n.
Answer:

$$T(N) = \left\{ \begin{array}{ll} 0, & \text{if } N = 1 \\ 1, & \text{if } N = 2 \\ T(\frac{2N}{3}) + T(\frac{2N}{3}) + T(\frac{2N}{3}) + 3, & \text{if } N > 2 \end{array} \right\} \quad (1)$$

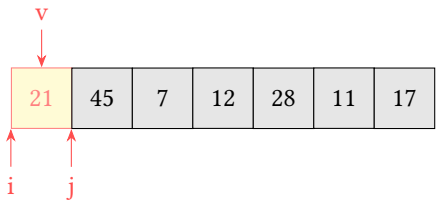
Using Master Theorem, we have $a = 3$, $b = 1.5$, $f(n) = 3 \sim 1 = N^{\log_{1.5}(1.5)} = O(N^{\log_{1.5}(3-\epsilon)})$ where $\epsilon = 1.5 > 0$, which is case 1 in Master Theorem. This means $T(N) \sim N^{\log_{1.5}(3)} \approx N^{2.71}$.

Problem 3. Consider the following Partition algorithm used by QuickSort (this version of Partition is based on the algorithm from the slides with the for-loop replaced by a while-loop).

Algorithm Partition(L, *start*, *end*)**:**

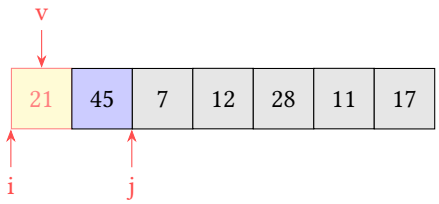
```
1:  v, i, j := L[start], start, start + 1.
2:  while j ≠ end do
3:    if L[j] ≤ v then
4:      i := i + 1.
5:      Exchange L[i] and L[j].
6:    end if
7:    j := j + 1.
8:  end while
9:  Exchange L[i] and L[start]
10: return i.
```

P3.1. Illustrate the operations performed by Partition on the array $A = [21, 45, 7, 12, 28, 11, 17]$. Show the content of A after each execution of the loop body.
Answer:
Start of function

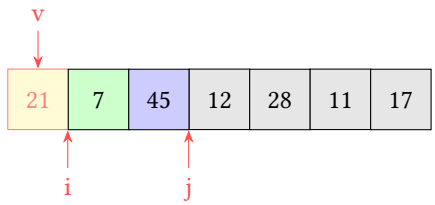


Start of loop

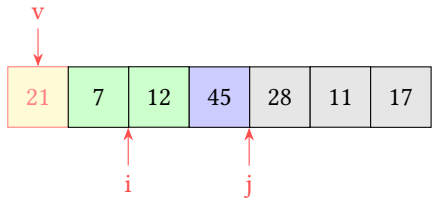
Did not enter if statement
End Loop 1



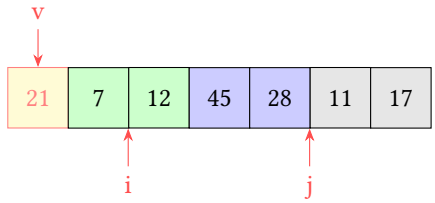
Entered if statement
End Loop 2



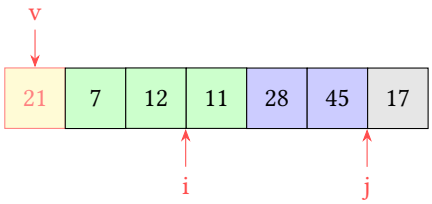
Entered if statement
End Loop 3



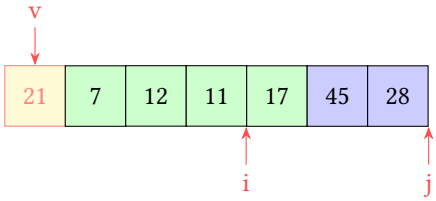
Did not enter if statement
End Loop 4



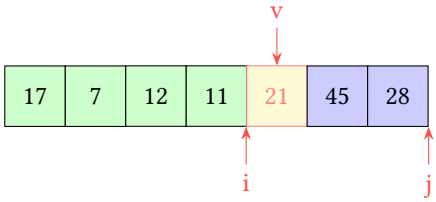
Entered if statement
End Loop 5



Entered if statement
End Loop 6



End of loop
Exchange L[i] and L[start]



P3.2. Provide pre-conditions and post-conditions for PARTITION and provide an invariant and bound function for the while-loop at Line 2. Prove the correctness of PARTITION.
Pre-condition:

- $|L| > 0$, else L[start] would error

Post-condition:

- All elements to the left of v are smaller than or equal to v, all elements to the right of v are larger than v, v is in the middle of the smaller and larger elements.
- The input array is the same length as the output array and contains the same values (not necessarily in the same order)

Loop Invariant:

- All entries in A[$start + 1 ... i$] are <= pivot.
- All entries in A[$i + 1 ... j$) are > pivot.
- A[start] = pivot.

Bound function:

- $end - j$, this is because when $end = j$, this means $end - j = 0$ and the loop will terminate

---

Algorithm PARTITION(L, $start$, end)**:**

1: $v, i, j := L[start], start, start + 1.$
Base case: v=L[start] (pivot) and i=0 and j=1 implies invariant (L1 and L2 are both empty because their range is out of bounds)

Given the invariant is:
All entries in L1 = L[$start + 1...i$] are <= pivot.
All entries in L2 = L[$i + 1...j$) are > pivot.
Induction hypothesis: The invariant holds at every step in the loop.
Induction step: Prove that the invariant holds at every step in the loop.

2: **while** $j \neq end$ **do**
3: Known: $j \neq end$
4: **if** $L[j] \leq v$ **then**
5: $i := i + 1$.
6: Known: $L[j] \leq pivot$, L[i_{new}] is ≥ pivot because i_{old} is always 1 cell behind the first element in L2
7: Exchange $L[i]$ and $L[j]$.
8: Known: j isn't at end of list yet. Exchanged the first element in L2 with the value at L[j], which we know is ≤ pivot. Due to this, L[$start + 1...i$] are all smaller than or equal to pivot and L[$i + 1...j - 1$) are all larger than pivot
9: **end if**
10: $j := j + 1$.
11: $j_{new} = j_{old} + 1$ meaning L[$i + 1...j_{new}$) are all larger than pivot, which is our invariant.
12: **end while**
13: Known: We know that L[$start + 1...i$] are all smaller or equal to pivot and L[$i + 1...j$) are all larger than pivot. We know that the i^{th} value is the last value included in L1 and everything past the i^{th} value is part of L2, so we can swap it with the start value (pivot), making the new list have all smaller or equal values to the left of the pivot and all larger to the right.
14: Exchange $L[i]$ and $L[start]$
Known: L is a list where L[start...i) values are smaller than or equal to the pivot, L[i] is the pivot value, L[i+1...end) are all values larger than pivot.
15: **return** i.

P3.3. Argue how PARTITION can be adjusted to run on singly linked lists L, while keeping a running time of $\sim|L|$.

Answer:
There are two methods to do this, both with the same thought process. (i) Exchange just the data between nodes. (ii) Exchange the actual nodes.

(i) Instead of holding the index i and j, we need to hold the nodes at position i and j. v can still be the data in the head node. To start, i would hold node 1, and j would hold node 2. As you are iterating through the loop, the comparison will need to be between v and j.data. If needing to exchange the nodes, you can just exchange the data, which is a constant time operation (3 ops to be specific). The while condition would now be j.next ≠ null as that specifies when you are at the last node. Whenever, there is an i++ or j++, that can be changed to node.next, which is also constant time. Finally, the final exchange at the end of the loop, is also a constant time function, as aforementioned. We would still need to hold a variable p that holds the index of the pivot (previous i), that will increment by 1 every time you call node.next on the ith node. This is because the index of the pivot needs to be returned at the end. This again is constant time as it is just incrementing an integer at max once per loop. Due to everything being constant time, while we loop through the whole list (N elements), this gives us $\sim L$ time complexity.

(ii) Instead of holding the index i and j, we need to hold the nodes right before position i and j. Holding the node before is since for a singly linked list, you can't go back, so you need to always have a reference (node.next) to the node you want to compare and swap. We also need a function to swap two nodes in the linked list. This function is constant time as it is just a few assignments. As exchanges

only happen between nodes i and j, it is sufficient to keep track of only these nodes, while updating them by calling node.next whenever we need to do i++ or j++. This way, we can traverse the list once while using constant time functions in each iteration, giving us a complexity of $\sim L$.

The while loop condition would now need to be when j.next.next = null as that is when we've reached the second last node in the list. This is when we want to terminate since we are always holding the node one before the one we want to compare (index j-1). Furthermore as mentioned in (i), we would need to hold a new variable p that holds the index of the ith node so we can return it at the end. As argued above, it is constant time, so it won't affect the time complexity.

Note: You don't need to hold the node at v as we only need the value for comparisons during the loop, and you can quickly access the head of a linked list when it comes to getting the value of the head at the start and exchanging it at the end of the function.

Problem 4. Consider pairs (x_i, y_i) such that x_i is the time at which person $i = 0, 1, 2, \ldots$ enters the museum and y_i is the time at which person i leaves the museum. You may assume that consecutive people enter the museum in order of increasing time ($x_0 \leq x_1 \leq \ldots$).

P4.1. Provide an algorithm MaxVisitors that takes as input $L = [(x_0, y_0), \ldots, (x_{N-1}, y_{N-1})]$ and computes in $\sim N \log N$ the maximum number of visitors in the museum at any time.
Answer:

Algorithm MaxVisitors(L)**:**

```
1:  timesList = new List
2:  for i ∈ L do
3:      timesList.add((x_i, enter))
4:      timesList.add((y_i, leave))
5:  end for
6:  sortedL := MergeSort(timesList) -> Assumption: MergeSort is correct and sorts based on time
    (element.first)
7:  currentVisitors, maxVisitors := 0, 0
8:  for i ∈ sortedL do
9:      ".second gets the second element of the tuple"
10:     if (i.second == enter) then
11:         currentVisitors = currentVisitors + 1
12:     else if (i.second == leave) then
13:         currentVisitors = currentVisitors − 1
14:     end if
15:     if (maxVisitors < currentVisitors) then
16:         maxVisitors := currentVisitors
17:     end if
18: end for
19: return maxVisitors.
```

P4.2. Argue why your algorithm MaxVisitors is correct and has a runtime complexity of $\sim N \log N$.
Answer (time):
To begin, on line 2, I start by looping through the given list and adding the times to a new list as separate elements. This is $O(N)$ as I iterate through the list once and use constant time operations per iteration.

Then, on line 6, I sort the new list of length 2N using MergeSort. As proven in class, this is $\sim (2N)log_2(2N) \sim O(Nlog_2N)$.

Following that, I iterate through the sorted list of length 2N using constant time operations per iteration, which gives $\sim 2N \sim O(N)$.

Aside from these loops, there are a few constant time operations on lines 1, 7 and 19, which will not affect the overall time complexity of the algorithm as they are $\sim O(1)$. As the largest time complexity of any loop in the algorithm is $\sim O(N)$, that is the runtime complexity of the algorithm.

Answer (correct):
To begin, on line 2, I loop through the input List and add the values to a new array as a tuple (time, flag). Then on line 6, I sort the list so that there is a new list of the times in ascending order with entering and leaving times as their own individual times. Because consecutive people must enter in $(x_0 \leq x_1 \leq \dots)$ order, we know that if we go from left to right of the sorted list, we can visualize the exact order in which people enter and leave the museum. If we add 1 to currentVisitors everytime there is an entering time and subtract 1 every time there is a leaving time, we can keep tract of the current people in the museum at one time (this is what the loop on line 8 does). Now holding the maximum value of all people in the museum at one time, we can always check if the currentVisitors is greater than the maxVisitors. If $currentVisitors > maxVisitors$, we know that is our new max. We also know that currentVisitors ≥ 0 as for each person, the time they leave is larger than the time they enter. This means for every leaving time (where we subtract 1 from currentVisitors), there was an entering time earlier (where we added 1 to currentVisitors). At the end of this loop, we can return the max knowing it holds the maximum people in the museum at once.

P4.3. Assume that the museum has a maximum capacity of M. Provide a datastructure with an operation PersonEnters(x_i, y_i) that computes in at-most $\sim \log M$ the number of visitors in the museum when person i enters the museum (for any number of persons).
Answer:
Hold two self-balancing binary search trees: one for enter times and one for exit times. The nodes for the tree will hold the number of nodes in its left subtree ($lcount$). In general the idea for PersonEnters is the following:
Assumption: People who enter at the same time count as inside the museum and people who exit at the same time someone enters counts as not in the museum.
1) In the enter times tree: Search for all nodes less than or equal to the time person i entered. This will get all people who have entered before them.
2) In the exit times tree: Search for all nodes less than or equal to the time person i entered. This will get all people who have exited before them.
3) Subtract all the people who exited before person i from the people who entered before person i. This will be our result that we can return.

P4.4. Argue why your algorithm PersonEnters is correct and has a runtime complexity of $\sim \log M$.
Answer (correct):
PersonEnters is correct as people can only enter and exit the museum in that order. We know any enter time before a given time is how many people entered the museum before that time. We also know any exit time before a given time is how many people exited the museum before a given time. To find the number of people in the museum at a given time, we need to know how many people have entered but not exited. Knowing how many entered and exited before a time, we can simply subtract all those who exited from those who entered, which will leave us with the people who have entered but not exited.

Note: Some of the correctness proof will make more sense when reading the time part below.

Answer (time):
For the algorithm to be logarithmic time, (i) the time it takes to insert the new person must be $\sim \log M$ and (ii) the time it takes to count the nodes less than the current time in both binary search trees must

be $\sim \log M$ individually. As these are all consecutive functions, if they are all $\sim \log M$ time, the total time complexity would be $\sim \log M$.

(i) As seen in class, the average cost of adding values to a binary search tree is $\sim \log M$. As we are using a self-balancing tree, we can even assure that the height of the BST will be $\log_2 N$, where N is the number of values rounded up to the nearest power of 2. The use of self-balancing BST is so that there can never be a skewed tree.

The algorithm for inserting follows a normal BST inserting algorithm however, every time a node is added, as the algorithm is passing through the tree, you add 1 to the lcount on each node if the value inserting is smaller than the current node being compared. This way each node can will correctly keep *lcount* (count of nodes in left subtree).

(ii) The general algorithm is the following: Hold a var *counter* for how many nodes are smaller than the given time. Traverse the tree starting at the root. If the current node is greater than the given time, go down the left subtree. Else if the current node is less than or equal to the given time, add *lcount* + 1 to *counter* (+1 is for the current node itself) of the current node and go down the right subtree. Recur through this algorithm and it will return the number of smaller values than a given time. Since at every node, you either go left or right (not both), you traverse the tree linearly downwards. The height of the tree is $\log M$, so the complexity of this algorithm is $\sim \log M$.

Given both (i) and (ii) have been proven to be $\sim \log M$ time, and these are going to be used independently of each other, the overall time complexity of this algorithm is $\sim \log M$.

Assignment Details

Write a report in which you solve each of the above problems. Your submission:

1. must be a PDF file;
2. must have clearly labeled solutions to each of the stated problems;
3. must be clearly presented;
4. must *not* be hand-written: prepare your report in LaTeX or in a word processor such as Microsoft Word (that can print or exported to PDF).

Submissions that do not follow the above requirements will get a grade of zero.

Grading

Each problem counts equally toward the final grade of this assignment.