MIT 6.1210 Problem Set 1

Joshua Pereira

February 22, 2024

1 Problem 1

Suppose that Slowpoke's algorithm encounters a list of clues where none of the indices correspond to a `here`. We first note that the algorithm is doing a redundant scan up to each successive index in the list. In other words, first the index 1 is checked, then 1 and 2, and then 1,2, and 3, and so on. Therefore, if there are no clues in the list that match `here`, the algorithm will reach the end of the list after:

$$\sum_{i=0}^{n-1} i = \frac{n(n+1)}{2}$$

steps. Therefore, the worst-case upper-bound of the algorithm is $O(n^2)$.

Likewise, suppose that Slowpoke's algorithm encounters a list of clues where the last index, $n-1$, corresponds to a `here`. The algorithm will therefore perform its redundant scan where the number of scans without a match is countable as:

$$\text{num\_scans} = 1 + 2 + 3 + ... + (n-1)$$

As Slowpoke's algorithm encounters the last index, it will proceed to return the index $n-1$. Therefore, we have that the number of total steps is:

$$\text{num\_steps} = 1 + 2 + 3 + ... + (n-1) + 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

steps. Therefore, the worst-case lower-bound of the algorithm is $\Omega(n^2)$. Thus, Slowpoke's algorithm is $\Theta(n^2)$

2 Problem 2

Given the array:

$$A = [3, 7, 5, 9, 10, 12, 4, 6, 8, 10, 13, 15, 17, 9, 5, 6, 7, 10, 15, 16, 13, 12, 7, 14, 12, 10, 8, 5, 7, 10, 12, 11]$$

The algorithm will first calculates $n = 32$ and $j = 16$. The index $A[16] = 7$ is not a peak as $A[15] = 6 < A[16] = 7$ but, $A[17] = 10 > A[16] = 7$. The algorithm will thus choose the else branch, as $A[j+1] > A[j]$ and return $17 + \text{find\_peak}(A[17, ..., 31])$.

Given the array $A' = A[17, ..., 31]$:

$$A' = A[17, ..., 31] = [10, 15, 16, 13, 12, 7, 14, 12, 10, 8, 5, 7, 10, 12, 11]$$

The algorithm then calculates $n' = 15$ and $j' = 7$ (by the floor). The index $A'[7] = 12$ is not a peak as $A'[7] = 12 > A'[8] = 10$ but, $A'[6] = 14 > A'[7] = 12$. Therefore, the algorithm will choose the if branch as $A'[j'-1] > A'[j']$ and return $\text{find\_peak}(A'[0, ..., 6])$.

Given the array $A'' = A'[0, ..., 6]$:

$$A'' = A'[0, ..., 6] = [10, 15, 16, 13, 12, 7, 14]$$

The algorithm then calculates $n'' = 7$ and $j'' = 3$ (by the floor). The index $A''[3] = 13$ is not a peak as $A''[3] = 13 > A''[4] = 12$ but, $A''[2] = 16 > A''[3] = 13$. Therefore the algorithm will choose the if branch as $A''[j''-1] > A''[j'']$ and return $\text{find\_peak}(A''[0, 1, 2])$.

Given the array $A''' = A''[0, 1, 2]$:

$$A''' = A''[0, 1, 2] = [10, 15, 16]$$

The algorithm will first calculates $n''' = 3$ and $j''' = 1$. The index $A'''[1] = 15$ is not a peak as $A'''[1] = 15 > A'''[0] = 10$ but, $A'''[1] = 15 < A'''[2] = 16$. The algorithm will thus choose the else branch, as $A'''[j'''+1] > A'''[j''']$ and return $2 + \text{find\_peak}(A'''[2])$.

Given the array $A'''' = A'''[2]$:

$$A'''' = A'''[2] = [16]$$

the algorithm will detect a singular index and return 0.

Closing the third recursive call, the algorithm will return $2 + 0 = 2$.

Closing the second recursive call, the algorithm will return 2.

Closing the first recursive call, the algorithm will return 2.

Closing the top-level call, the algorithm will return $17 + 2 = 19$.

The algorithm will choose the index $i = 19$, $A[19] = 16$, as the identified peak. In fact, this is a peak, a local-maxima. This is because $A[19] = 16 > A[18] = 15$ and $A[19] = 16 > A[20] = 13$.

3 Problem 3

3.A Part 1

Beginning with an empty array, the first insert_first(0, "a") will be an $O(1)$ operation to bind the value. However, with the next iteration, we run out

of space for two elements. So, we double the size of the array. This will be done repeatedly throughout the process as we eventually get to at least n total spots of memory. However, insert_first(0,"a") will require that all indices in the existing array be slid down to the right before the first index is rebound. This is an amortized $O(n)$ operation, accounting for the reallocation. So, we will have an $O(n^2)$ worst-case amortized run-time for the first part.

The second for-loop is n different calls of delete_last(). Any one call of delete_last() is an $O(1)$ amortized operation. This is because when less than $\frac{n'}{2}$ spaces are filled, then the dynamic array is resized. Therefore, this is an $O(n)$ amortized call.

Therefore, the total worst-case runtime is quantifiable as $O(n^2)$.

3.B Part 2

Beginning with an empty array, the first insert_last("a") operation will be an $O(1)$ operation to bind the value.

However, with the next iteration, we run out of space and need to reallocate memory. This will happen periodically, again, throughout the process. However, the unlike the previous question, the successive insert_last("a") operations will be $O(1)$ amortized operations. This is because we are appending to the end of a list without affecting other elements, unless we need to resize. So, we will have an $O(n)$ worst-case amortized run-time for the first part.

For the second for-loop, the we have the same case as in part 1. The deletions will be an amortized $O(1)$ operation each, resulting in an $O(n)$ operation overall.

Therefore, the total worst-case runtime is quantifiable as $O(n)$.

3.C Part 3

Beginning with an empty array, all calls of insert_last and delete_at will be the same because they undo each other. In other words, the first call of insert_last will result in an $O(1)$ placement of the first element. However, the next line delete_at will remove this lone element in $O(1)$ time and return the array to an empty array. Therefore, these n $O(1)$ different calls will result in an $O(n)$ run-time. This will not be amortized, as we will choose to resize the array only when we request an addition and it is full.

4 Problem 4

4.A Part a

Suppose that $f' \in O(f)$ and $g' \in O(g)$. Therefore, we have the relations:

$$f'(n) \leq c_1 \cdot f(n) \ \forall n > n_1, c_1 > 0$$
$$g'(n) \leq c_2 \cdot g(n) \ \forall n > n_2, c_2 > 0$$

We can add these two relations together:

$$f'(n) + g'(n) \leq c_1 \cdot f(n) + c_2 \cdot g(n)$$

However, we must carefully choose our constants. We know that since $c_1 > 0$ and $c_2 > 0$, then we can choose $c_1 \geq c_2$ without loss of generality. This would be the same as choosing the largest of the two constants. We note that by doing so, $g(n) \leq c_1 \cdot g'(n)$ is still true since:

$$g'(n) \leq c_2 \cdot g(n) \leq c_1 \cdot g'(n)$$

Moreover, suppose we choose $n > n_1$, where again, $n_1 > n_2$. In other words, if we choose $c_3 = \max(c_1, c_2)$ and $n_3 = \max(n_1, n_2)$, then we have found constants such that:

$$f' + g'(n) \leq c_3 \cdot f + g(n), \ \forall n > n_3$$

where $c_3 = \max(c_1, c_2)$ and $n_3 = \max(n_1, n_2)$. $f' + g' \in O(f + g)$. `TRUE`.

4.B Part b

Suppose we have the case that $f' = 2\sqrt{n}$ and $f = \sqrt{n}$ and $g' = 2e^n$ and $g = e^n$. In these examples, it is true that $f' \in O(f)$ and $g' \in O(g)$. However, when we go to test the composition: $g' \circ f'$

$$2e^{2\sqrt{n}} \leq c_3 \cdot e^{\sqrt{n}} \ \forall n \geq n_3$$

there exists no constant. So, we cannot have the case that $g' \circ f' \in O(g \circ f)$. `FALSE`.

4.C Part c

We can solve the recurrence:

$$\sum_{i=0}^{\log n} 2^i \cdot \frac{n}{2^i} \cdot \log(\frac{n}{2^i}) = n \sum_{i=0}^{\log n} \log(n) - \log(2^i)$$

$$n \cdot \log(n) - n \sum_{i=0}^{\log n} \log(2^i) = n \cdot \log(n) - n \log(2) \sum_{i=0}^{\log n} i$$

Simplifying:

$$n \cdot \log(n) - n \log(2) \frac{\log(n)(\log(n) + 1)}{2} =$$

$$n \log(n) - \frac{n \log^2(n) \log(2) + n \log(n)}{2} =$$

$$\frac{n \log(n)}{2} - \frac{n \log^2(n) \log(2)}{2}$$

Therefore, it must be the case that: $T(n) \in O(n \log(n))$.

5 Problem 5

Let us keep all existing infrastructure for a doubly-linked list but, declare a second pointer, along with the head and tail pointers. We will name the pointer `mid`. Upon initialization of an empty doubly linked list, we set the mid pointer to `null`.

After the addition of an element by either insert_last or insert_first, before ending either function call, we will additionally determine if we need to update the mid pointer. If the size of the array is odd **after addition of the new node**, do nothing. The index determined by the floored-half does not change. Otherwise, if the size of the array is even after addition of an element, perform one of the following:

- If the mid pointer is set to null, we an empty list. Set the mid pointer to the node indicated by the pointer `head`.
- Otherwise, the mid pointer is set to some node, A. Set the "mid" pointer to the node located at A.next to increment the middle index.

After the removal of an element by either delete_last or delete_first, before ending either function call, we will again determine if we need to update the mid pointer. If the size of the array is even **after removal of a node**, do nothing. Otherwise, perform one of the following:

- If the new size of the list is 0, we have created an empty list. Set the mid pointer to `null`.
- Otherwise, the mid pointer must be set to the parent node of its current assignment, A. Set the mid pointer to the node located at A.prev.

With this updated infrastructure, we can implement insert_mid(x) by first checking the size counter of the array. If the array is empty, then execute insert_first(x). Otherwise, follow the pointer `mid` to the current middle node A. Upon reaching the node, we can begin by rebinding pointers.

$$A.\text{prev.next} = x$$

$$x.\text{prev} = A.\text{prev}$$

$$A.\text{prev} = x$$

$$x.\text{next} = A$$

Then, we can increment the size counter for the entire list by one. If the size counter is even after, do nothing. This is because we have added the node x to the left of the original mid node A. Therefore, new middle node would be A in the updated list, as we had left it. However, if the size counter is

odd after updating, set the mid pointer to the node located at A.prev because the intended middle index does not change but, x is located at this location (A.prev).

Similarly, we can implement delete_mid() by first checking the size counter of the array. If the array has one element, then execute delete_last(). Otherwise, follow the pointer `mid` to the current middle node A. Upon reaching the node, we can begin by rebinding pointers.

$$A.\text{prev.next} = A.\text{next}$$

$$A.\text{next.prev} = A.\text{prev}$$

Then, we can decrement the size counter for the entire list by one. If the size counter is odd after decrementing, set `mid` to A.prev. This is because we have removed a node and the floor decreases by one unit. Therefore, the new middle node would be to the previous node of A. However, if the size counter is even after updating, set the mid pointer to the node located at A.next because the floor has gone down by one index, located at the location (A.next).

Then, we clean-up.

$$A.\text{prev} = \texttt{null}$$

$$A.\text{next} = \texttt{null}$$

Both of these algorithms only update $O(1)$ pointers each execution. As well, we only perform $O(1)$ operations in reading and writing pointers. Furthermore, these algorithms work in-place as we keep track of the middle node. So, we will have an $O(1)$ and $\Omega(1)$ run-time: $\Theta(1)$.

6 Problem 6

Given an array A of length n with strictly increasing integers, we can solve the problem by recursively checking the middle element of the array and moving into the left or right sub-arrays as needed. The algorithm is as follows:

6.A Part i

Given an array A of length n, if the array is empty, return s. Otherwise, calculate the middle index as $i = \lfloor \frac{n}{2} \rfloor$:

If the middle element $A[i] < s$, then the smallest integer greater than or equal to s not in A must be greater than $A[i]$. In other words, because the array is strictly increasing, we recurse on $A[i+1:]$ to locate the missing integer on the right side.

Otherwise, the middle element $A[i] \geq s$. If this is the case, we need to check if there are unbroken elements up to $A[i]$ from s. In other words, if there are

no missing integers when counting from s to $A[i]$ in the array, then the missing integer must be located in the right sub-array and is greater than or equal to $A[i] + 1$: it is the same missing integer when the algorithm is applied to the right sub-array with $s = A[i] + 1$. This can be done by checking if $A[i - k] = s$, where $k = A[i] - s$. If $A[i - k] \neq s$, then there is a missing integer in the left sub-array that is greater than or equal to s. Suppose that $i - k < 0$. Then, it is imperative that we check if $A[0] = s - i$ to check for consecutive values. This will prevent an index out of bounds error if s is disproportionate to the size of the array and its values.

6.B Part ii

We will prove the correctness of the algorithm by induction. For an empty array, we will return s as a base-case.

For the purposes of induction, suppose the algorithm works for arrays of length k. Now, suppose we are given an array of length $k + 1$.

Suppose the middle index, $i = \lfloor \frac{k+1}{2} \rfloor = A[i] \leq s$. Therefore, the smallest integer greater than or equal to s not in A is the same integer greater than or equal to s that is also not in $A[i + 1 :]$ because all other indices are not eligible. $A[: i]$ and $A[i]$ are all less than or equal to s. Therefore, by the inductive assumption, the algorithm will locate the correct smallest integer greater than or equal to s not in $A[i + 1 :]$.

Suppose the middle index, $i = \lfloor \frac{k+1}{2} \rfloor = A[i] > s$. We have the two built-in cases: one with consecutive values up to $A[i]$ and one without. If there are consecutive values, then we recurse on the latter-half of the array with a larger value of s that matches the new sub-problem with updated bounds. If there are no consecutive values, then we recurse on the former-half of the array with the same value of s searching for the missing piece in the broken sequence. Therefore, by the inductive assumption, the algorithm will locate the correct smallest integer greater than or equal to s not in either sub-array.

6.C Part iii

In the worst-case lower-bound, suppose the algorithm is asked to find a value s that is equal to the last index. This will result in $\log(n)$ recursive divisions. In the worst-case upper-bound, suppose the algorithm is asked to find a value s that is greater than the last index. This will result in $\log(n)$ recursive divisions. The important part is that both function calls will trigger essentially identical recursive trees but, end on different base-cases. In other words, the lower-bound and upper-bound coincide and the bound is tight for:

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

Solving:

$$T(n) = \sum_{i=0}^{\log(n)} 1 = \log(n)$$

Therefore, the algorithm is $\Theta(\log(n))$: $O(\log(n))$.