Elissa Ito September 22, 2022

6.1210 Problem Set 1

Problem 1 *(Collaborators: None)*

Part 1(a)

line 2 costs $O(1)$
for i in range(n) $\Rightarrow i = 0, 1, ...n-1$
$\Rightarrow$ for j in range(i+1) $\Rightarrow j = 0; j = 0, 1; j = 0, 1, 2; ...j = 0, 1, 2...n-1$; Since $\sum_{i=0}^{n-1} O(i+1) = O(\frac{n}{2}(n+1)) = O(n^2)$
Thus the worst case upper bound would be $O(n^2) + O(1) = O(n^2)$

Assuming that the treasure is in the last clue, both for loops will run until the end.
$\Omega(1) + \sum_{i=0}^{n-1} \Omega(i+1) = \Omega(\frac{n}{2}(n+1)) = \Omega(n^2)$
Thus the worst case lower bound would be $\Omega(n^2)$

Part 1(b)

1. each insert_at(0,"a") is $O(n)$ $\Rightarrow$ executed n times is $O(n^2)$
   each delete_last() is $O(1_{am})$
   for k operations, worst case runtime is $\leq$ k*amortized time $\Rightarrow$ executed n times is $O(n)$
   the cost for all operations is $O(n^2) + O(n) = O(n^2)$
   the amortized cost for all operations is $O(n^2)/2n = O(n)$

2. append("a") is equivalent to insert_last("a") which is $O(1_{am})$, executed n times $\Rightarrow O(n)$
   delete_last() is $O(1_{am})$, executed n times $\Rightarrow O(n)$
   the cost for all operations is $O(n) + O(n) = O(n)$
   the amortized cost for all operations is $O(n)/2n = O(1_{am})$

3. for each iteration in the loop, append("a") is $O(1)$ and delete_at(0) is $O(1)$ since the length of the array is always 1.
   the cost for each loop iteration is $O(1) + O(1) = O(1)$
   the cost for all operations is $n * O(1) = O(n)$
   the amortized cost for all operations is $O(n)/2n = O(1)$

Problem 2 *(Collaborators: None)*

Part 2(a)

1. Find a number N where N is the upper bound for n:
   a. Start with $x = 1$
   b. Ask "is $n \leq x$?"
   c.i. If yes, $N = x$; go to step 2.
   c.ii. If no, double the value of x and repeat step 1b.

2. Upon finding N, complete binary search for n such that
   search if in array of all integers between exclusive upper bound N and inclusive lower bound N/2:
   a. Given element x midway between N/2 and N, ask "is $n \leq x$?"
   b.i. If yes, continue search with left half of current array.
   b.ii. If no, continue search with right half of current array.
   c. Check if current array is length 1
   d.i. If yes, return the item in array.
   d.ii. If no, get middle element x of current array and ask "is $n \leq x$?" and continue from step 2b.

how many steps is that? $O(log(n))$ lower bound is previous x, upper bound is x what is the distance between the lower and upper bound? how long would binary search between the two take?

if know n¡=N we can do binary search w/ N

Right now I'm thinking start with like a really big number $x = 10^{10}$
not sure how to find $n \leq N \leq 2 * n$
if $n > x$, try $new_x = 2 * x + 1$ (closest integer)
and if $n \leq x$, try $new_x = x/2$ (closest integer)
actually this creates an infinite loop - added + 1
since the time is $O(log(n))$ must be binary search right?

Part 2(b)

An algorithm is correct iff:
1. Every solution picked is a valid solution.
2. Every valid solution is picked.

We know that every solution picked is a valid solution because the initial array contains n. We know this since in step 1 we looked for the first $x \nleq n$, which means that $N > n$ and since in the previous step $\frac{N}{2} \leq n$, $\frac{N}{2} \leq n < N$ and since all integers from $\frac{N}{2}$ to N is in the initial array, so is n.

We know that every valid solution is picked because there is only one possible answer, n, and the algorithm runs until it is found.

Part 2(c)

The worst case upper bound runtime for each step is:

1. Find a number N where N is the upper bound for n:
   Let $N = 2^i$. i is the number of times step 1b and 1c are repeated, since at first $x = 1$ and x is doubled until $x = N$. Since $N \geq n \Rightarrow 2^i \geq n$. Taking the log of both sides yields $log(2^i) \geq log(n) \Rightarrow i * log(2) \geq log(n) \Rightarrow i \geq \frac{log(n)}{log(2)}$. Step 1a is $O(1)$, and step 1b and 1c are also $O(1)$. Step 1b and 1c repeated i times has the runtime of $O(2 * \frac{log(n)}{log(2)}) \Rightarrow O(log(n))$.

2. Binary search between $\frac{N}{2}$ and N:
   Since N is the first value in step 1 at which the answer to the question "is $n \leq x$?" was no, this means $N > n$ but also $N \leq 2n$. Since $n < N$, by the nature of integers $\frac{n}{2} < \frac{N}{2}$. This means the lower bound is greater than $\frac{n}{2}$ and the upper bound is smaller than $2n$. Thus the range of the array upon which binary search is conducted is $< (2n - \frac{n}{2}$, or $< \frac{3n}{2}$. Binary search on an array of size n is $O(log(n))$, thus a binary search on this array of size $< \frac{3n}{2}$ is $O(log(\frac{3n}{2})) = O(log(\frac{3}{2}) + O(log(n)) = O(log(n))$

3. Thus the total runtime of the algorithm is $O(log(n)) + O(log(n)) = O(log(n))$

Problem 3 *(Collaborators: None)*

Part 3(a)

First, conduct a binary search for s in A.

1. Base Case: If array is empty, return False since by definition s is not in array.
2. Second Base Case: If array is of size 1, check if s is the element in the array. If it is, return True, if not, return False.
3. Recursive Step: If the middle element of the array is equal to s, return True. If it is greater than s, recurse on the left half of array. If it is smaller than s, recurse on the right half of the array.

If s is not in A, return s. If not, conduct a binary search for the smallest element greater than s in A.

1. Base Case: If array is of size 1, return that element.
2. Recursive step:
   a. If the middle element is less than or equal to s, recurse on the right half of the array.
   b. If the first element of the array is greater than s, check if all elements to the left are consecutive by finding the difference between the middle index by the first index, and checking to see if that difference is the same as the value between the middle element and first element. If they are consecutive, recurse on the right half of the array. If they are not, recurse on the left half of the array.
   c. If the first element is not greater than s, find difference between the values of the middle element and s. If the element at the index that is the middle index minus the difference is s, that means all values between s in the array and the middle element are consecutive. If so, recurse on the right half of the array. If not, recurse on the left half of the array.

Part 3(b)

Proof of correctness for the first binary search:

1. Base Case: If the length of the array is zero by definition the return False is correct since s is not in the array.

If the length of the array is 1 and we check if that element is s or not, the result is correct by logic.

2. Recursive Step:
   Assuming the algorithm works for an array of size n-1, we add an element to the middle of the array making the array size n. (Recitation 2 notes states that we can assume all recursive calls are correct.)
   a. If the element is greater than s, the algorithm would recurse on the left half, yielding the correct answer.
   b. If the element is smaller than s, the algorithm would recurse on the right half, yielding the correct answer.
   c. If the element is s, the algorithm would return s, which is also the correct answer.

3. Thus this binary search algorithm is correct by induction.

Proof of correctness for the second binary search:
Since the algorithm only conducts this search if s is in the array, we can assume that s is in the array since by the above proof the first search is correct.

1. Base Case:
   If the array has size 1, return that element+1.

2. Recursive Step:
   Assuming the algorithm works for an array of size n-1, and we add an element to the middle of the array making the array size n.
   a. If the element is less than or equal to s, the algorithm would recurse on right half, yielding the correct answer.
   b. If the element is greater than s and the left half is consecutive, the algorithm would recurse on the right half, yielding the correct answer.
   If the element is greater than s and the left half is not, the algorithm would recurse on the left half, yielding the correct answer.

3. Thus this binary search algorithm is correct by induction.

Assuming the first search correctly determines whether s is in the array and the second search correctly finds the smallest element in the array greater than s (given s is not in the array), then every solution picked would be a valid solution since if s is not in the array the algorithm returns s, and if it is it would return the smallest element greater than s. Every valid solution is picked since there is only one valid solution possible.

Part 3(c)

The first binary search is $O(log(n))$, the second binary search is also $O(log(n))$ by nature of binary searches. Checking the answer of the first binary search is $O(1)$. The total algorithm would be $O(log(n))$.