Assignment2

Isaac Ng

March 14, 2020

Question 1a Use unwinding to find a closed form for T(n) when $n \geq k$

Intuitively, this is a merge-sort function where at a certain level k , the program will run constantly rather than as a function of n.
When $n > k$, the run-time at each depth is n so, all I need to find is the depth of n that is greater than k. This is $log_2(n/k)$.
The number of nodes at each level is 2^q where q is the depth/height of the node. This value is $2^{log_2(n/k)} = n/k$.
For values between two powers of two, we will want to round the value either up or down. In this case, we want to round up. For example, $n = 8, k = 5$ we want $n/k = 2$ and $1 < 8/5 < 2$.
Combining these values, the Run-time of the function is $nlog_2(\lceil n/k \rceil) + c(\lceil n/k \rceil)$.

Question 1b What is the big-Θ comnplexity of T(n)? Does it depend on k? Briefly justify your answer (no proof required). You may not assume $n \geq k$ for this part. Do not use the master theorem.

$\Theta(T(n))$ is $nlog(\lceil n/k \rceil)$.
The function depends on k because if k is larger than n, the run-time becomes constant. Otherwise, it is a function of n.

Question 1c Rather than assigning a fixed cost to the $n \leq k$ case, replace c with n^2. Find a closed form for T'(n) for $n \geq k$, and show how you got there.

Looking back at part 1a), I notice the only place I use c is for counting the cost of the leaves of the tree.
This means I only need to replace c with n^2 to get my desired outcome for run-time T'(n).
Run-time $T'(n) = nlog_2(\lceil n/k \rceil) + n^2(\lceil n/k \rceil)$.

Question 1d Is $T'(n) \in \Theta(T(n))$? Why or why not? Briefly justify your answer.

When looking at $\Theta(T'(n))$, the latter term (containing n^2) is much larger than the first term (with the logarithm. This is because n is a larger term than a logarithm.
This means that $\Theta(T'(n)) = n^3$ which is definitely not the same as $\Theta(T(n)) = nlog(\lceil n/k \rceil)$ and since it is a larger function (as explained above), $T'(n) \notin \Theta(T(n))$.

Question 2a Based on the informal specification above, write precise pre- and post-conditions for umax. Your postcondition should use symbolic notation rather than restating the English description above.

Pre-conditions:
1) non-empty list A : $A \neq \emptyset$
2) positive integer elements in A : $A \subset \mathbb{Z}^+$

Post-conditions:
I want to use maximum's definition as stated in the question.
max(A) = x where $(\exists j \in \mathbb{N}, A[j] = x) \bigwedge (\forall i \in \mathbb{N}, i < len(A) \implies A[i] \leq x)$
If a unique maximum exists:
umax(A) = x where $(x = max(A)) \bigwedge [\forall i, j \in \mathbb{N}, (i, j < len(A)) \bigwedge (i \neq j) \bigwedge (A[i] = x) \implies A[j] \neq x]$
If a unique maximum does not exist:
Some negative number is returned.

Question 2b The given Python code above has a bug. Demonstrate the bug by finding a value of A which meets the precondition, where the umax mis-behaves. For the value of A that you find, you should state the expected behaviour and how it differs from the function's actual behaviour on that input.

I want to choose $A = [2, 3, 3]$. A negative value should be returned because a unique maximum does not exist.
When umax on the tail is performed, we get -1 so, tmax = -1.
When comparing A[0], which is 2, to the absolute value of -1 which is 1, $2 > 1$ so 2 is returned when a negative value should be returned.
Therefore, on $A = [2, 3, 3]$, the expected behaviour is for a negative value to be returned. However, 2 is returned.

Question 2c Consider our second draft of the function umax. Prove that this function is correct with respect to the specifications you devised in part (a).

Assume A follows the pre-conditions, it is a non-empty subset of $\mathbb{Z}^+$
I want to show that the post-condition is held in this program: a negative value is returned when a unique maximum does not exist or the unique maximum x is returned if it does exist.
Looking at the code, I want the negative value to be negative of the maximum in the list, if it is not unique.
I want to prove this with induction.
Predicate P(n): The function's post-conditions, that either the unimax or the negative of the maximum is returned, are met when a list A of length n, following the aforementioned preconditions, is used as the input.
Base Case P(1) (A is of length 1, $A = [k]$ for some $k \in \mathbb{Z}^=$)

According to lines 2 and 3, A[0], its only element, is returned, which is the unique maximum of the list.
My post-condition is held in the Base Case.

Inductive Step
Assume P(n): on a list of size n, A's unimax is returned by the function when it exists. If it does not, the negative maximum is returned.
WTS P(n+1)
Analysis of the function:
tmax holds the value of umax(tail), which is umax(A[1:]).
Since A is of length n + 1, the tail is of length n.
Let the max of A[1:] be at index j.
By the inductive hypothesis, since tail is of length n, the post-condition is met.
tmax = +/-A[j] depending on if the tail has a unique maximum.

I want to break the rest of my proof into cases.

Case 1: $A[0] > A[j]$
Line 7 would be false since they are not equal.
Line 9 would be true, so A[0] is returned.
This is correct since $A[0]$ is the unique maximum.
My post-condition is held in this case.

Case 2: $A[0] = A[j]$ and A[j] is a unique maximum.
Line 7 would be true since they are equal so $-A[0]$ is returned. This is correct since $A[0] = A[j]$ which is the maximum, meaning its negative value should be returned.
My post-condition is held in this case
Case 3: $A[0] = A[j]$ and A[j] is not a unique maximum.
$A[0] \neq -A[0] = -A[j]$ = tmax so line 7 would be false.
$A[0] \not> A[j]$ in this case so line 9 would be false.
Else is reached and tmax = (-A[j]) is returned.
My post-condition is held in this case
Case 4: $A[0] < A[j]$
Lines 7 and 9 are both false so else statement is run.
The else statement is reached so, tmax is returned.
My post-condition is held in this case.

Therefore, I have proven with four cases, that P(n+1) holds given P(n).
I have shown using simple induction that the second draft of the function umax is correct.

Question 3 Prove that maj is correct.

Part 1: Loop Invariants

Lemma 1: x is majority in $Prev_j$.

Predicate P(j): x is majority in $Prev_j$ assuming j iterations occur.
Base Case: P(0): no iterations.
From the Pre-condition that x is a majority in A, $Prev_0 = A$ so x is majority in $Prev_0$.

Inductive Step:
Assume P(j): x is majority in $Prev_j$.
Let x be the majority element in $Prev_j$.
Let k be any non-majority element in $Prev_j$.
Let n be the length of $Prev_j$.
Let x_n be the number of elements x in $Prev_j$.
Let k_n be the number of elements k in $Prev_j$.
Let x_p be the number of pairs x in $Prev_j$.
Let k_p be the number of pairs k in $Prev_j$.
Let k_g be the number of groups of k in $Prev_j$

From the inductive hypothesis, we know that $x_n > k_n$.
This means that $x_n > n/2$ or $\lfloor n/2 \rfloor + 1 \leq x_n \leq n$.
Since $x_n + k_n = n$, $0 \leq k_n \leq \lceil n/2 \rceil - 1$.
Let me take $k_n = \lceil n/2 \rceil - 1$. If I can prove for this number of k that x is majority in $Prev_{j+1}$ then I can prove for other values of k_n since less k would mean more x and more x pairs.
For this value of k, it can be grouped in many ways, ranging from 1 group of $\lceil n/2 \rceil - 2$ pairs to $\lceil n/2 \rceil - 1$ groups of 0 pairs. We can say that $k_g + k_p = \lceil n/2 \rceil - 1$ because $k_g + k_p = k_n$.
The intuition behind this is simple. Let's say we have a group of 2. This means this group has a pair, or one group has one pair. The number of objects is 2, placed in 1 group of 1 pair.
I will use $k_{g1}, k_{g2}, ...k_{gn}$ to denote each grouping of k in $Prev_j$.
I can surround each group so that we have $[x, k_{g1}, x, g_{g2}, ..., x, kg_n]$ without placing the other x into the list. This is because, by definition of group, groups of k will have at least 1 x between them otherwise they would simply be a larger group.
For instance, $[x, k_{g1}, k_{g2}]$ could be simple $[x, k_{g1}]$.
Notice how at this point I have placed the same number of x as k_g.
When I place the rest of x, since they cannot go within a group of k, they will always for a pair with another x. The remaining x that need to be placed is the number of pairs of x, x_p.
So, we have $x_p + 2 \times k_g + k_p = n$.
From before, we also have $k_g + k_p = \lceil n/2 \rceil - 1$.
Substituting k_g, we have $x_p = n - 2[\lceil n/2 \rceil - 1 - k_p] - k_p = n - 2(\lceil n/2 \rceil) + 2 + k_p$.
Case n is even:
$2 \times \lceil n/2 \rceil = n$ so $x_p = 2 + k_p$.
Case n is odd:
$2 \times \lceil n/2 \rceil - 1 = n$ so $x_p = 1 + k_p$.
This shows that $x_p > k_p$ for all values of n. When we reduce k_n and do not assume k_n to be the maximum, we will have even more pairs of x.

Line 16 of the code makes $Prev_{j+1} = Curr_j$ and $Curr_j$ is the list of pairs in $Prev_j$ so, since $x_p > k_p$ as shown above, x will be majority in $Curr_j$ which means that x is majority in

$Prev_{j+1}$.
I have shown through induction that x is always majority of $Prev_j$ on each iteration j, assuming iteration j exists.

Lemma 2: $x \in Curr_j$
P(j): x is in $Curr_j$ after j iterations.
Base Case: P(0)
By Lemma 1, x is a majority in $Prev_1$ which means $x \in Prev_1$.
Also, $Curr_0 = Prev_1$ so $x \in Curr_0$ as I wanted.

Inductive Step:
Assume P(j) that x is in $Curr_j$ after j iterations.
I want to show that $x \in Curr_{j+1}$.
Case 1: iteration j+2 exists.
Then, From Lemma 1, x has a majority in $Prev_{j+2}$ which means that $x \in Prev_{j+2}$.
Also, $Prev_{j+2} = Curr_{j+1}$.
Case 2: iteration j+2 does not exist.
Then this means $\text{len}(Curr_{j+1}) = \text{len}(Prev_{j+1})$.
This only occurs when there are n pairs in a list of n, meaning all elements are the same in $Prev_{j+1}$.
Since x is a majority in $Prev_{j+1}$, that element must be x, so the list $Curr_{j+1}$ is entirely made of x and $x \in Curr_{j+1}$.

Lemma 3: $Curr_j \subset Prev_j$
Looking at the code, we see that $Curr_j = R(Prev_j)$.
From Theorem 1.1, we know that R is correct.
The number of times each element appears in the returned list is equivalent to the number of pairs of that element in the input list.
Therefore, $Curr_j \subset Prev_j$ according to Theorem 1.1, which is proven in the document.

Part 2: Partial Correctness
Assume that the while loop terminates after k iterations.
Therefore, we know the while loop condition is false, which means that $\text{len}(Curr_k) = \text{len}(Prev_k)$.
This only happens when all elements in both prev and curr are the same.
This means that for curr and prev to be the same, there will have to be the same number of pairs as there are elements.
This only occurs when all elements are the same since there are n possible pairs in a list of length n (including pair with indices n-1 and 0).
By Lemma 1, the majority element x is always a majority in Prev.
This means $x \in Prev_j$ for each iteration j and $x \in Prev_k$.
By Lemma 2, $x \in Curr_j$ for each iteration j and $x \in Curr_k$.
When all elements are the same in both $Prev_k$ and $Curr_k$, this means the only element remaining is x, which is the majority element, which is exactly what we wanted.

Part 3: Termination

Let $m_j = len(Prev_j) + len(Curr_j)$.
Decreasing: Show that $len(Prev_j) + len(Curr_j) > len(Prev_{j+1}) + len(Curr_{j+1})$
We know that $len(Curr_j) = len(Prev_{j+1})$ because $Curr_j = Prev_{j+1}$.
Now I only need to show that this is strictly decreasing; $len(Prev_j) > len(Curr_{j+1}$.
We know the while loop terminates when $len(Prev_j) = len(Curr_j)$.
If there is a j+1 iteration, that means that $len(Prev_j) \neq len(Curr_j)$.
From Lemma 3, we know $Curr_j \subset Prev_j$ so, since they are not equal, $len(Prev_j) > len(Curr_j)$.
Combining these facts, we have $len(Prev_j) > len(Curr_j) = len(Prev_{j+1}) \geq len(Curr_{j+1})$.
I have shown that $len(Prev_j) + len(Curr_j) > len(Prev_{j+1}) + len(Curr_{j+1})$.

$m_j \in \mathbb{N}$.
We know that $len(Prev_j), len(Curr_j) \in \mathbb{N}$ so, since we are adding them, the value can absolutely not go below 0 and will always be an integer.
I have proven that the function terminates and is Partially Correct. Therefore, I have shown the iterative correctness of maj.