Homework 1

Joni Vrapi

09/10/2022

Statement of Integrity: I, Joni Vrapi, attempted to answer each question honestly and to the best of my abilities. I cited any, and all, help that I received in completing this assignment.

Problem 1. For this problem we are asked to describe the time complexity of the stated linear search algorithm, showing the tightest asymptotic representation from Θ, O, or Ω. My first inkling on this was to heed the name of the algorithm and assume that it is, in fact, linear in time. This algorithm accepts a parameter x, which is the thing that is being searched for, and another parameter A, which is the array that is to be searched. If $x \in A$, the index of x is returned.

It begins by iterating through the array, checking if the indexing variable $i \leq A.length$ and that $x \neq A[i]$. This, to me, implies that in the *worst case*, where the item that you are searching for is not in input the array, this algorithm will loop through every item in the array for a total of n times for a time complexity of $O(n)$. Finally, it checks if $i \leq A.length$ and assigns to the *location* variable either the indexing variable i or 0 if $i > A.length$. This implies that the *best case* scenario for this algorithm, where the item that you are looking for is in the first position of the array, is $\Omega(1)$. For the average case, we have two cases:

1. The item can be in any of n possible positions in the array, from index 1 to index $n + 1$.
2. The item is not in the array.

So, there are n possibilities in the first case, and 1 possibility in the second case, for a total of $n+1$ possibilities. If the item you are looking for is at index k, linear search will do $k + 1$ comparisons. Summing those up, you get the known sum of [1]:

$$\frac{n(n+1)}{2} \tag{1}$$

In the second case, there are n possible positions in the array, and all of them are searched, with no result being found. Therefore, the total amount of cases for both situations are:

$$\frac{n(n+1)}{2} + n \tag{2}$$
$$= n(\frac{n+1}{2} + 1) \tag{3}$$

For the average number of comparisons we have

$$\frac{n(\frac{n+1}{2} + 1)}{n+1} \tag{4}$$
$$= \frac{n}{2} + \frac{n}{n+1} \tag{5}$$

The dominant term here is the $\frac{n}{2}$, so the average case complexity here is also $O(n)$. What is the tightest asymptotic representation? Because of the best case complexity here being $\Omega(1)$ and the worst case being $O(n)$ as well as:

$$\Theta(f(n)) \Leftrightarrow (\Omega(f(n)) \wedge O(f(n))) \tag{6}$$

Which was pulled from [2], we can not have an asymptotically tight bound. I would therefore argue that the tightest bound here is $O(n)$.

Problem 2a. For this problem we are asked to describe the time complexity of the stated binary search algorithm. This is a wicked interesting searching algorithm because it begins with an item to be found, x, and a *sorted* input array A. It works by comparing x to the value in the middle of A, and then narrowing the search to only the left/right half of A (depending on if the value of $x > A[i] \vee x < A[i]$), and then repeating itself. This continues until either the value is found, or the interval of the array that you are looking at is empty, at which point it returns either the index of the found item, or zero if it's not found. This is an excellent strategy for finding things, because whenever you are able to reduce your problem size by half on each iteration, you are talking about logarithmic time (which is faster than linear time!). The question, however, begets a precise asymptotic bound, but knowing that the problem size is being reduced gives me a hint that my answer should have a $log(n)$ in it.

So, let's think about this. The *best case* scenario would be if $x = A[\frac{A.length}{2}]$, making the algorithm exit after only 1 comparison, leading to an $\Omega(1)$. For the *worst case*, letting $B =$ the interval of the array we are searching, we have

- On the first iteration: $B.length = n$.
- On the second iteration: $B.length = \frac{n}{2}$.
- On the third iteration: $B.length = \frac{n}{2^2}$.
- On the k^{th} iteration: $B.length = \frac{n}{2^k}$.

We also know that after k iterations, $B.length = 1$,

$$\therefore \frac{n}{2^k} = 1 \tag{7}$$

Applying the log function to both sides yields

$$log(n) = log(2^k) \implies log(n) = klog(2) \tag{8}$$

$$\therefore k = log(n) \tag{9}$$

This will therefore yield a worst case time complexity of $O(log(n))$ for binary search. Due to (6), and considering the best and worst case complexities being different, I would argue the tightest bound is $O(log(n))$.

Problem 2b. This algorithm makes use of the $\lfloor floor \rfloor$ function to determine the middle index of the array interval that it is currently looking at. While the floor function runs in constant time, its use can be eliminated by a clever manipulation of the middle index. From [3] we see that we can

```
BINARY-SEARCH(x, A)
 1  i = 1
 2  j = A.length
 3  while i < j
 4        m = (i + j)/2
 5        if x > A[m]
 6              i = m + 1
 7        else j = m − 1
 8        if x = A[i]
 9              location = i
10        else location = 0
11  return location
```

...change line 7 in the algorithm to subtract 1 from m instead of setting $j = m$, resulting in the same algorithm as before, but without use of the $\lfloor floor \rfloor$ function. This does not change the overall time complexity of the algorithm, keeping it at $O(log(n))$, but it does make it run slightly faster.

Problem 3. For this problem we are asked to solve the recurrence relation

$$T(n) = 4T(\frac{n}{4}) + n^2 \tag{10}$$

The first thing that pops into my mind looking at this problem is that the Master Theorem applies, and will likely be the easiest way of proving this. From the master theorem

$$T(n) = aT(\frac{n}{b}) + f(n) \tag{11}$$

We can see that $a = 4$, and $b = 4$. Since $a > 1$, $b > 1$, and f is asymptotically positive, we can see that $n^{\log_a b} \implies n^{\log_4 4} \implies n$ and $f(n) = n^2$. The master theorem tells us to then check whether $4f(\frac{n}{4}) \leq cf(n)$ for some $c < 1$ and $\forall n$ sufficiently large which is indeed the case here.

$$\therefore T(n) \implies \Theta(f(n)) = \Theta(n^2) \tag{12}$$

Problem 4. For this problem we are asked to solve the recurrence relation

$$T(n) = 3T(\frac{n}{3} + 1) + n \tag{13}$$

The first thing that pops into my mind looking at this problem is that the Master Theorem applies, and will likely be the easiest way of proving this. From the master theorem

$$T(n) = aT(\frac{n}{b}) + f(n) \tag{14}$$

We can see that $a = 3$, and $b = 3$. Since $a > 1$, $b > 1$, and f is asymptotically positive, we can see that $n^{\log_a b} \implies n^{\log_3 3} \implies n$ and $f(n) = n$. At this point, I was confused about what to do with the remaining $+1$ in this relation, but proceeded to drop it as an insignificant term. Hopefully this was correct.

The master theorem then tells us that $T(n) \implies \Theta(nlog(n))$.

$$\therefore T(n) = \Theta(nlog(n)) \tag{15}$$

Problem 5. For this problem we are asked to solve the recurrence relation

$$T(n) = T(\sqrt{n}) + 1 \tag{16}$$

The first thing that pops into my mind looking at this problem is that I think we need to use substitution with the Master Theorem in order to solve this. If we substitute $n = 2^m$, we obtain

$$T(2^m) = T(\sqrt{2^m}) + 1 = T(2^{\frac{m}{2}}) + 1 \tag{17}$$

We can then define

$$S(m) = T(2^m) \tag{18}$$

$$\therefore S(m) = S(\frac{m}{2}) + 1 \tag{19}$$

At this point I am unsure, however by the Master Theorem I believe that $S(m) \in \Theta(2^m)$ which $\implies T(n) \in \Theta(n)$

Problem 6a. For this problem we are asked to prove that insertion sort can sort the $\frac{n}{k}$ sublists, each of length k, in $\Theta(nk)$ worst case time.

If we can say that, for an input of size n, insertion sort runs on $\Theta(n^2)$ worst case time, then we can also say that for an input of size k, insertion sort runs on $\Theta(k^2)$ worst case time. So, the worst case time to sort $\frac{n}{k}$ sublists of length k will be:

$$\frac{n}{k} * \Theta(k^2) \implies \Theta(nk) \tag{20}$$

Proving this by induction we can say:

$$n = 1 \tag{21}$$

$$\frac{1}{k} * k^2 = 1k \tag{22}$$

Assuming $n = k$ we can say:

$$\frac{k}{k} * k^2 = k * k \tag{23}$$

$$k^2 = k^2 \tag{24}$$

From $n = k$ we show that the result holds for $n = k + 1$

$$\frac{k+1}{k} * k^2 = (k+1)k \tag{25}$$

$$= (k+1)k = (k+1)k \tag{26}$$

Problem 6b. For this problem we are asked to prove how to merge the sublists in $\Theta(nlog(\frac{n}{k}))$ worst case time.

For this problem, we have n elements divided into $\frac{n}{k}$ sublists of length k. In order to merge these into one list of length n, we have to take two sublists at a time and recursively merge them. From Figure 2.5 (my page numbers are off, so I did not mention them here because they will be different from yours) [4] we see that this will take $log(\frac{n}{k})$ steps, and at every step, we will do n comparisons. $\therefore$ the whole process will run in $\Theta(nlog(\frac{n}{k}))$

I was not sure how to develop a formal proof for this, however, so after scouring the internet I was able to find [5] which I will reference here.

The recurrence relation for merging the sublists is

$$T(a) = \begin{cases} 0 & \text{if } a = 1, \\ 2T(\frac{a}{2}) + ak & \text{if } a = 2^p, \text{if } p > 0 \end{cases} \tag{27}$$

Proving this by induction we can say:

$$T(1) = 1klog(1) = k * 0 = 0 \tag{28}$$

Assuming $T(a) = aklog(a)$ we can say:

$$T(2a) = 2T(a) + 2ak = 2(T(a) + ak) = 2(aklog(a) + ak) \tag{29}$$
$$= 2ak(log(a) + 1) = 2ak(log(a) + log(2)) \tag{30}$$
$$= 2aklog(2a) \tag{31}$$

Substituting in $\frac{n}{k}$ for a:

$$T(\frac{n}{k}) = \frac{n}{k}klog(\frac{n}{k}) \tag{32}$$
$$\implies nlog(\frac{n}{k}) \tag{33}$$

Problem 6c. For this problem we are asked what the largest value of k, as a function of n, is that will make the modified algorithm (which runs in $\Theta(nk + nlog(\frac{n}{k}))$) have the same running time as the standard merge sort ($\Theta(nlog(n))$).

For this problem, I believe we can do some algebra as well. For example, for the modified algorithm to have the same running time as the standard one, k must not grow faster than $log(n)$, because if it does, then because of the nk term, the algorithm will run slower than $\Theta(nlog(n))$. Therefore, if we assume that $k = log(n)$ we can:

$$k = log(n) \tag{34}$$
$$\Theta(nk + nlog(\frac{n}{k})) \tag{35}$$
$$\begin{aligned} &= \Theta(nk + nlog(n) - nlog(k)) \\ &= \Theta(n(log(n)) + nlog(n) - nlog(log(n)))) \\ &= \Theta(2nlog(n) - nlog(log(n)) \\ &= \Theta(nlog(n)) \end{aligned} \tag{36}$$

Problem 6d. For this problem we are asked how we should choose the optimal value of k | insertion sort $>$ merge sort. To do this, I think we need to calculate the exact running time expression based on the method in [4] in Exercise 1.2.2. In Exercise 1.2.2 we are told that, for inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64nlog(n)$ steps. So, basically, I believe we have to make the following inequality *false*:

$$8n^2 < 64nlog(n) \tag{37}$$

$$= n < 8log(n) \tag{38}$$

$$= \frac{n}{8} < log(n) \tag{39}$$

$$= 2^{\frac{n}{8}} < n \tag{40}$$

Which is to say that, for whatever values of n we can use to make this inequality false, e.g. $2^{\frac{n}{8}} > n$, that will be the value at which merge sort becomes faster than insertion sort again. I say "again" here because we know insertion sort runs in $\Theta(n^2)$ which is a lot slower as $n \to \infty$ than merge sort's $\Theta(nlog(n))$. At $n = 1$, merge sort beats insertion sort, but for values greater than 1 and less than k, insertion sort beats merge sort. For what value that is exactly, I just brute forced it starting with $n = 8$ and going up by powers of 2 until it was true, then walking it back down binary-search (by hand) style. Through this I was able to find that $n = 44$ is the pivot value where merge sort starts to beat insertion sort again. $\therefore 1 > n < 44$ describes the space where insertion sort beats merge sort.

References

[1] "List of mathematical series." https://en.wikipedia.org/wiki/List_of_mathematical_series. Accessed on 2022-09-11.

[2] G. D. Luca, "The difference between lower bound and tight bound." https://www.baeldung.com/cs/lower-bound-vs-tight-bound. Accessed on 2022-09-11.

[3] https://stackoverflow.com/questions/27655955/why-does-binary-search-algorithm-use-floor-and-not-ceiling-not-in-an-half-open. Accessed on 2022-09-11.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. The MIT Press, 4 ed., 2022.

[5] https://clrs.skanev.com/02/problems/01.html. Accessed on 2022-09-12.