CSE102-01: Intro to Algos[*] PSET-2 Solutions
19th January 2024, Deadline: 26th Jan.

Instructions: Deadline in a week, so submit by **Fr. Jan. 26 midnight PST**. Submit to Canvas a pdf of your solutions, with your name and ucsc id number. Feel free to discuss solutions with your colleagues (other students), explore the internet, read and be inspired from the textbooks etc. But what you type should be typed by you and you only in latex. In algorithms, we appreciate brevity and accuracy: your solutions should be to the point, often one or two sentences suffice to provide all necessary details and descriptions.

Ex.1 (More Recursions) As we saw in class, sometimes the Master Theorem cannot directly be applied. Solve the following recursive relations (assume all base cases $T(1)$=1 and/or that n is a power of 2 if it helps you.)
(i) $T(n) = 2T(\sqrt{n}) + \log n$ (Hint: Maybe replacing n with 2^m will help you.)
(ii) $T(n) = 2T(n/2) + n \log n$ (Hint: You may want to check again the proof of the Master Theorem.)
(iii) $T(n) = 2T(\sqrt{n}) + 1$
(iv) $T(n) = T(n/2) + T(n/3) + T(n/10) + T(n/20) + n$ (Hint: Check again the substitution method we saw in class.)

Solution We solve them one by one:

- (i) Answer is $T(n) = O(\log n \log\log n)$. As the hint says, we set $n = 2^m \iff \log n = m$. Then the recursive relation is:

  $$T(2^m) = 2T(2^{m/2}) + m$$

  Now we do a further replacement: we rename $T(2^m)$ by another function, say $Q(m)$, that is $Q(m) = T(2^m) = T(n)$ (this is a simple renaming, nothing too deep). But now the relation reads:

  $$T(2^m) = 2T(2^{m/2}) + m \iff Q(m) = 2Q(m/2) + m$$

  Now, this is a recursive relationship with respect to m that we have seen multiple times. The answer can be given with the Master Theorem and is $Q(m) = m \log m \iff T(n) = \log n \log\log n$ and we are done!

- (ii) Answer is $T(n) = O(n \log^2 n)$. For this relation, Master Theorem doesn't apply directly so we need to revisit the proof of Master Theorem

[*] PI: Vaggos Chatziafratis

and "unpack" the tree, as we did in class. Unpacking for one more step yields the following calculation:

$$T(n) = 2T(n/2)+n\log n = 2\left(2T(\frac{n}{4}) + \frac{n}{2}\log(\frac{n}{2})\right)+n\log n = 4T(\frac{n}{4})+n\log(\frac{n}{2})+n\log n.$$

Continuing the unpacking we observe that we have at level ℓ of the tree, the following terms: $2^\ell T(\frac{n}{2^\ell})$ and also $n\sum_{i=0}^{\ell-1}\log\frac{n}{2^i}$. As always, we ask ourselves "what is the height of the tree" or equivalently, what is the depth ℓ. Since we divide n by 2 at every level, we will have only $\log n$ levels, by the definition of the logarithm.

This means the final runtime is the sum of the following $\log n$ many terms:

$$n\log n + n\log(\frac{n}{2}) + n\log(\frac{n}{4}) + n\log(\frac{n}{8}) + n\log(\frac{n}{16}) + \ldots + n$$

Since there are $\log n$ many terms, using properties of the logarithm this is the same as:

$$n\log\left(\frac{n\cdot n\cdot n\cdot n\cdots n}{1\cdot 2\cdot 4\cdot 8\cdots n}\right) = n\log(n^{\log n}) - n\log(1\cdot 2\cdot 4\cdot 8\cdots 2^{\log n})$$

Now notice that $n\log(n^{\log n}) = n\log^2 n$ because by properties of logarithms we have $\log(n^{\log n}) = \log n\cdot\log n$. The other term is equal to:

$$n\log(1\cdot 2\cdot 4\cdot 8\cdots 2^{\log n}) = n\log(2^{1+2+3+4+\log n})$$

As we have seen many times, the sum of the first k numbers is $1+2+3+4+\ldots+k = \frac{k(k+1)}{2}$. Using this sum for $k=\log n$ we simply get $1+2+3+\ldots+\log n = \frac{\log n\cdot(\log n+1)}{2} = \frac{1}{2}\log^2 n + \frac{1}{2}\log n$.

Hence our final calculation gives runtime:

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

where we again used properties of the logarithms $\log a^b = b\log a$.

- (iii) Answer is $T(n) = O(\log n)$. To figure out why, see again (i) relation above. It resembles (i) in that we take square root, so the same hint can apply. We again replace $n = 2^m \iff \log n = m$ and hence:

$$T(n) = 2T(\sqrt{n}) + 1 \iff T(2^m) = 2T(2^{m/2}) + 1$$

We now rename $Q(m) = T(2^m) = T(n)$ so:

$$T(2^m) = 2T(2^{m/2}) + 1 \iff Q(m) = 2Q(m/2) + 1$$

Now the Master Theorem can be applied and we get directly that $Q(m) = \Theta(m)$. Replacing back we have $T(n) = Q(m) = \Theta(m) = \Theta(\log n)$.

- (iv) Answer is $T(n) = O(n)$ and in fact $T(n) = \Theta(n)$ since even in the first step we need to pay n and so we deduce that $T(n) = \Omega(n)$ already. We saw a very similar recursion in class and we shall go with the substitution method (Master Theorem doesn't apply here). For our guess, let's say $T(n) \leq 60n$ and let's prove our guess is correct using induction.

  Base case: As in all such exercises, for small values of n, say $n = 1$ (or $n = 2$), we have $T(1) = 1 \leq 60$ (or $T(2) \leq 100$) and the base case goes through easily.

  Inductive Hypothesis: We assume $T(k) \leq 60k$ for all values of k that are **strictly** less than n, and then need to prove the inductive step that is $T(n) \leq 60n$. To do so we use the recursive relation that creates **smaller** subproblems onto which we can apply the Inductive Hypothesis stated above for the terms $T(n/2)$ and $T(n/3)$ and $T(n/10)$ and $T(n/20)$ since all these correspond to smaller-than-n subproblems:

  $$T(n) = T(n/2)+T(n/3)+T(n/10)+T(n/20)+n \leq 60\cdot\frac{n}{2}+60\cdot\frac{n}{3}+60\cdot\frac{n}{10}+60\cdot\frac{n}{20}+n$$

  Notice that doing the calculations we get:

  $$60 \cdot \frac{n}{2} + 60 \cdot \frac{n}{3} + 60 \cdot \frac{n}{10} + 60 \cdot \frac{n}{20} + n = 30n + 20n + 6n + 3n + n = 60n$$

  hence we proved the inductive step, that is: $T(n) \leq 60n$. So we proved that $T(n) \leq 60n \forall n$ which concludes the upper bound analysis. As we mentioned $T(n) = \Omega(n)$ because we need to pay a linear in n term even in the first step, and so we conclude the tight bound that $T(n) = \Theta(n)$.

Ex.2 (Binary Search, Unlimited) You are given an infinite array $A[\cdot]$ in which the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. Importantly, you are not given the value of n. Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, in $O(\log n)$ time. (If you are disturbed by the fact that the array A has infinite length, assume instead that it is of length n, but that you don't know this length, and that the implementation of the array data type in your programming language returns the error message ∞ whenever elements $A[i]$ with $i > n$ are accessed.)

Solution Here the important thing to remember is that binary search can still be implemented to run in $O(\log n)$ time, even if the length of the array n is not known a priori. Below we describe a way to figure out what the length of the array is in time $O(\log n)$; once we do this, we can simply run binary search (in $O(\log n)$ time) on that prefix of the array up to n that only contains valid numbers, and no infinity values. The total time would be $O(\log n)$.

Main Idea: To find the length n, simply start with the first element and **keep doubling** the index of the element you check until you hit an ∞ value. In other words, we access element $A[1]$, then $A[2]$, then $A[4]$, then $A[8]$, then $A[16]$,

then $A[32]$ and so on. If we see a number, we keep doubling, and whenever we see an ∞ value we stop, and never check beyond this index.

The question we have to understand is "how many elements are we gonna check before hitting the "invalid" value of ∞?" The answer is at most $\log(2n) = 1 + \log n = O(\log n)$ (can you see why we never overshoot beyond $2n$?). After this, we use binary search to determine exactly the point where the ∞ values start (and hence what the last element of the array is). In total, we spent $O(\log n)$ time.

Ex.3 (Index-Value Match) Given a sorted array of distinct integers $A[1, \ldots, n]$, you want to find out whether there is an index i for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $O(\log n)$.

Solution This is a standard coding interview question that we can tackle with binary search.

Main idea: Check the middle element of the array. There are 3 cases:

- Case I: If $A[i] = i$: We are done in this case and we declare that yes there is such an index.
- Case II: If $A[i] > i \iff A[i] - i > 0$: In this case, since the array is sorted and elements are distinct, we are guaranteed that any element **to the right of** $A[i]$ is no good. In other words, checking indices larger than i is never gonna yield an index with the desired property. It is important that numbers are distinct, so we know the values inside the array increase at least by $+1$ so the index value can never catch up to the stored value in the array. Formally, if $i' > i$ then $A[i'] \geq A[i] + (i' - i) = (A[i] - i) + i' > i'$. Try to understand this also intuitively why this is the case, by considering and playing around with an example: maybe $i = 30$ and $A[30] = 50$. Then $A[31] \geq 51, A[32] \geq 52$ etc. so the index can never catch up with the stored value. Notice how important it is that we assume distinct numbers and sorted array.

  Our algorithm then needs to check the left part of the array and we do so recursively. Notice since at this step we got rid of half of the array, and we paid only a constant cost (just did one comparison), so our runtime relation is identical to binary search so this leads to $O(\log n)$ runtime.
- Case III: If $A[i] < i$: Similar to the previous case, but now there is no point in checking **to the left of** $A[i]$. So we recurse only on the right side of $A[i]$. Again notice how we get rid of half of the array and so this leads to a $O(\log n)$ runtime like in binary search.

Ex.4 (k-way merge operation). Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

(i) Here's one strategy: Using the merge procedure from class, merge the first

two arrays, then merge in the third, then merge in the fourth, and so on. What is the time complexity of this algorithm, in terms of k and n?
(ii) Give a more efficient solution to this problem, using divide-and-conquer.

Solution

- (i) This is a naive strategy. We know merging a sorted list of n numbers with a sorted list of m numbers takes time $n+m$ in general. So merging the first two arrays takes time $n+n=2n$, creating an array R of size $2n$, and then there are $k-2$ other arrays left-to-be-merged with R. Merging the resulting array R with the third array takes time $2n+n=3n$, and creates the array Q of size $3n$. We see that every time we make a merge, the size increases by an additive factor of n, and so finally the last merge will happen between an array of size $(k-1)n$ and the last sorted array of size n which takes time $(k-1)n+n=kn$. The total computation time was:

$$2n+3n+4n+5n+6n+\ldots+kn = n(2+3+4+\ldots+k) = \Theta(nk^2)$$

- (ii) We can do better by using divide-and-conquer, or by simply remembering the idea we saw in class about finding the minimum and maximum elements of an array simultaneously (or finding the min and second min element of an array): the idea is that we should build a tree.

  To do so, we merge in pairs: array 1 with array 2, array 3 with array 4, array 5 with array 6, array 7 with array 8 and so on. Generally, we create pairs of arrays that we merge together and since there are in total k arrays, there are $k/2$ such pairs. The time needed for each of those merges is $2n$ so in total we spent $2n \cdot \frac{k}{2} = nk$ time for this level. Once we are done with those merges, we repeat this merging-in-pairs idea for the resulting $k/2$ arrays.

  How much time are we spending? The computation can easily be represented by a tree with the bottom level having k leaves, then after pairing and merging we have $k/2$ nodes, then $k/4$, then $k/8$ etc. until the final merge which is between an array of size $nk/2$ and another array of size $nk/2$. The tree has depth $\log k$ (since originally there were k arrays at the leaves of the tree to be merged) and for the total time spent at each level is nk. Summing over all the levels yields a time $(nk)\cdot(\log k) = nk\log k = o(nk^2)$ so we conclude that this second merging strategy is strictly better than the naive first strategy.

Ex.5 You are given two sorted lists of size m and n. Give an $O(\log m + \log n)$ time algorithm for computing the kth smallest element in the union of the two lists.

Solution This is also a popular coding interview question where again we can make use of binary search and ideas from the QuickSelect algorithm we say in class for partitioning an array based on a pivot in linear time.

Main Idea: This is a standard interview question, with lots of materials online to prepare against it. Here we briefly go over the main idea to highlight some points. The idea is similar to Binary Search.

Suppose that the arrays are called A and B, and the k-th smallest element is greater than or equal to the first m values of A. This would mean that it is also greater than or equal to the first $k-m$ values in B. Here take specific examples to understand how we proceed. For example, think of $n = 100, m = 100, k = 80, m = 20$. Draw an example with these values.

The idea is now to use a binary search to find the number m in the range 0 to k. To use a binary search, we initialize a candidate value $q = k/2$. We then have to be able to determine if the number q we are currently trying is lower than the correct value m, or higher. But that is easy: check if $A[q+1]$ is less than $B[k-q]$, then $A[q+1]$ is part of the smallest k elements, so $q < m$. Again, try a specific example to see what this means. When $q = k/2$ we are initially comparing $A[k/2]$ against $B[k/2]$.

By similar reasoning, if $B[k-q+1]$ is less than $A[q]$, then we know that $q > m$. (The details of the algorithm, and the stopping conditions require careful attention. I haven't written down about these cases, but if you implement the algorithm you of course need to take care of these. Here I wanted for you to understand the basic logic behind the solution, not the edge cases.)

In this way, we can reduce the interval of the search by **half** each iteration, and find the value of m in $O(\log k)$ steps. This is less than the runtime required in the question, because we know that $k \leq n + m$, so $\log(k) \leq \log(n+m)$, so we take the log of the sum of n and m rather than the sum of the logs. Hence, this gives us total runtime $O(\log n + \log m)$.

Ex.6 Recall the factorial of n, denoted $n!$, which is equal to the product $1 \cdot 2 \cdot 3 \cdots n$. Show that $\log(n!) = \Theta(n \log n)$. (You can use Stirling's approximation for the factorial, but there is a way to do it without it too.)

Solution We use and prove two simple inequalities:

$$n! \geq (n/2)^{n/2} \texttt{ and also } n! \leq n^n$$

For the first one, observe that by definition $n!$ contains $n/2$ terms each of which is at least $n/2$ (the terms are specifically $n/2 \cdot (n/2+1) \cdot (n/2+2) \cdots (n-1) \cdot n$). The second one is even easier:

$$n! = 1 \cdot 2 \cdot 3 \cdots n \leq n \cdot n \cdot n \cdots n = n^n$$

To conclude the exercise, observe that $\log\left((n/2)^{n/2}\right) = \frac{n}{2}\log(\frac{n}{2}) = \Theta(n \log n)$ and also $\log(n^n) = n \log n$ where we used the properties of the logarithms ($\log(ab) = \log a + \log b$).

Ex.7 Sort the following functions in increasing order of magnitude, that is from the smaller to larger growth rate. If two of them are asymptotically of the same order, you should note it. Explain your answer by comparing every two consecutive functions in your sorted order.

2^{2^n}	$n!$	$n2^n$	$10n$
$\log(n!)$	$\log^{10} n$	$n^{\log\log n}$	$\left(\frac{\log n}{\log\log n}\right)^{\frac{\log n}{\log\log n}}$
$\log n^3$	$\frac{n}{\log n}$	$\frac{n^2}{\log^{10} n}$	$\frac{\log n}{n}$
$3n^6$	e^n	$\sqrt{n!}$	$\binom{n}{4}$

Solution The correct ordering is in the following figure, where we sort them in increasing order going from top-left to bottom-right (meaning that the smallest of all is $\frac{\log n}{n}$ and the largest of all is 2^{2^n}).

$\frac{\log n}{n}$	$\log n^3$	$\log^{10} n$	$\left(\frac{\log n}{\log\log n}\right)^{\frac{\log n}{\log\log n}}$
$\frac{n}{\log n}$	$10n$	$\log(n!)$	$\frac{n^2}{\log^{10} n}$
$\binom{n}{4}$	$3n^6$	$n^{\log\log n}$	$n2^n$
e^n	$\sqrt{n!}$	$n!$	2^{2^n}

For the proof we could use the definitions of the big-O and little-o notation and big-Ω and little-ω notation, but it often becomes easier when we use the limit definitions we saw in class.

For example:

$$If \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0, \ then \ f(n) = o(g(n))$$

whereas

$$If \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty, \ then \ f(n) = \omega(g(n))$$

But if it happens and the limit is equal to a constant c (independent of n):

$$If \lim_{n\to\infty} \frac{f(n)}{g(n)} = c, \ then \ f(n) = \Theta(g(n))$$

Other useful relations here that you should keep in mind are:

- $\log n! = \Theta(n \log n)$
- $\binom{n}{4} = \frac{n!}{4!(n-4!)} = \frac{n(n-1)(n-2)(n-3)}{4!} = \Theta(n^4)$
- $n! = \Theta(\sqrt{n}(\frac{n}{e})^n)$ (this is called Stirling's approximation, see Wiki)
- $n^{\log\log n} = \Theta(2^{\log n \log\log n})$ (by properties of logarithms)