CMPSCI 311: Introduction to Algorithms **Fall 2018**

Homework Solutions

Released 10/4/2018 Due 11:59pm 10/18/2018 in Gradescope

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

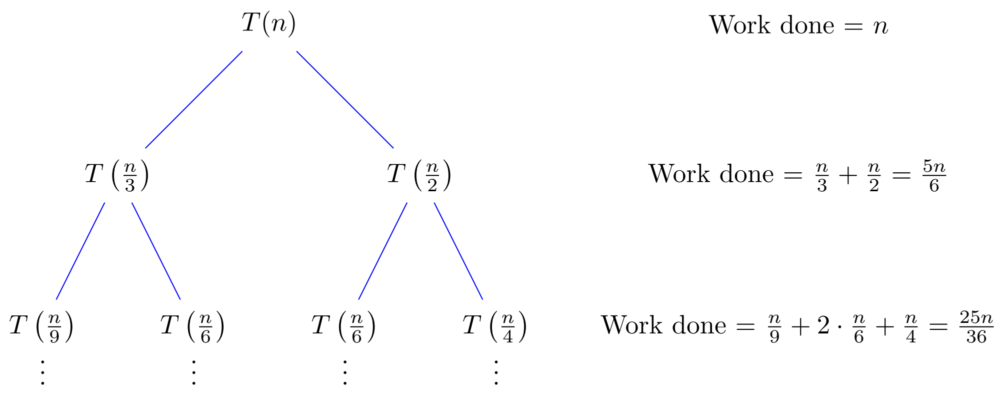
You may work in groups, but you must individually write your solutions yourself. List your collaborators on your submission.

If you are asked to design an algorithm as part of a homework problem, please provide: (a) the pseudocode for the algorithm, (b) an explanation of the intuition for the algorithm, (c) a proof of correctness, (d) the running time of your algorithm and (e) justification for your running time analysis.

Submissions. Please submit a PDF file. You may submit a scanned handwritten document, but a typed submission is preferred. It will be very helpful if in your submission each question starts on a new page.

1. **(20 points) Recurrences.**

   Solution. (a) The first few levels of the recursion tree look as follows.


$T(n)$
Work done $= n$
$T\left(\frac{n}{3}\right)$
$T\left(\frac{n}{2}\right)$
Work done $= \frac{n}{3} + \frac{n}{2} = \frac{5n}{6}$
$T\left(\frac{n}{9}\right)$
$T\left(\frac{n}{6}\right)$
$T\left(\frac{n}{6}\right)$
$T\left(\frac{n}{4}\right)$
Work done $= \frac{n}{9} + 2 \cdot \frac{n}{6} + \frac{n}{4} = \frac{25n}{36}$

At each level, the total work done is multiplied by 5/6 (**Think:** why? Formally speaking, this would still require a proof.). We stop at the leaves of the tree (the base cases), which have constant values. The number of leaves is at most $\Theta(n)$ (**Think:** why?), where the value of the function is constant. Therefore, the leaves contribute $O(n)$ work, and together with the rest of the work done on the levels, this adds up to

$$T(n) = n(1 + \frac{5}{6} + \frac{25}{36} + \ldots) + O(n) \leq n \cdot \frac{1}{1 - \frac{5}{6}} + O(n) = 6n + O(n) = O(n).$$

Here we have used the important identity, for any positive integer k and positive real number $r < 1$, we have

$$1 + r + r^2 + \ldots + r^k \leq \sum_{j=0}^{\infty} r^j = \frac{1}{1 - r}.$$

The lower bound holds trivially, since we need n work just at the first level of the recursion. Therefore, $T(n) = \Theta(n)$.

By the way, this is very easy to verify, using the upper bound found from the recursion tree. Simply take $T(n) = 6n$ (which we obtained as a bound for the work done on the levels), and check whether the recurrence is satisfied.

(b) We could use recursion trees for this as well, but instead we use the "unrolling" technique shown in discussions. We have, for any $1 \le k \le n$,

$$T(n) = T(n-1) + \log n = T(n-2) + \log(n-1) + \log n = \ldots = T(n-k) + \sum_{i=1}^{k} \log(n+1-i).$$

Therefore, using the base case $T(0) = \Theta(1)$, we get, setting $k = n$,

$$T(n) = T(0) + (\log 1 + \log 2 + \ldots + \log n) = \Theta(1) + \log(1 \cdot 2 \cdots n) = \Theta(\log(n!)).$$

To get a more explicit approximation, note that

$$\sum_{i=1}^{n} \log i \le \sum_{i=1}^{n} \log n = n \log n,$$

while

$$\sum_{i=1}^{n} \log i \ge \sum_{i=n/2}^{n} \log i \ge \sum_{i=n/2}^{n} \log(n/2) = \frac{n}{2} \cdot \log(n/2) = \frac{n}{2}(\log n - 1),$$

where we have used the fact that $\log n$ is a positive, monotone increasing function. It follows that $\log(n!) = \Theta(n \log n)$, showing that $T(n) = \Theta(n \log n)$ as well.

(c) We can use the "unrolling" technique again to get

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n} = 2\left(2T\left(\frac{n}{4}\right) + \frac{n/2}{\lg(n/2)}\right) + \frac{n}{\lg n} \\ &= 4T(n/4) + \frac{n}{\lg n - 1} + \frac{n}{\lg n} \\ &\vdots \\ &= 2^k T(n/2^k) + \sum_{i=1}^{k} \frac{n}{\lg n - (i-1)}. \end{aligned}$$

This process stops when $2^k = n$, or when $k = \log n$, with

$$T(n) = n \cdot T(1) + n \cdot \sum_{i=1}^{\log n} \frac{1}{\log n - (i-1)}.$$

This sum is equal to, by a change of variables (**Think:** why?),

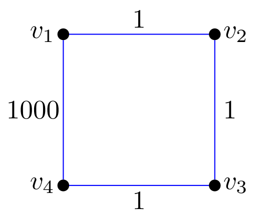
$$\sum_{i=1}^{\log n} \frac{1}{\log n - (i-1)} = \sum_{j=1}^{\log n} \frac{1}{j} = \Theta(\log \log n),$$

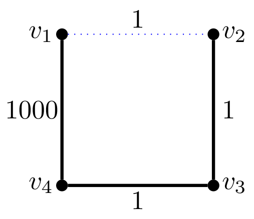
using the Fact given in the problem. It follows that

$$T(n) = n \cdot T(1) + \Theta(n \log \log n) = \Theta(n \log \log n).$$

2. **(15 points) Divide and Conquer the Tree.**

Solution. Consider the following graph with nodes v_1, v_2, v_3, and v_4. Suppose we denote the edges as e_{12}, e_{23}, e_{34} and e_{41} respectively, and suppose they have weights 1, 1, 1, and 1000 respectively. Using the given algorithm, it is possible (and acceptable) to partition V into $\{v_1, v_4\}$ and $\{v_2, v_3\}$ (their sizes are the same). Then, clearly the recursive MSTs would just be the single edges $\{e_{41}\}$ and $\{e_{23}\}$. We can then pick either of two weight-1 edges across the cut, and then we are left with an MST of weight $1 + 1000 + 1 = 1002$. However, note that the correct (unique) MST for G should be the path (v_1, v_2, v_3, v_4), of weight 3.





Incidentally, it was pointed out on Piazza that we could also get a counterexample by forcing one of the sides of the cut to be **disconnected**. Then, we can add a lowest-cost edge across the cut, but that would leave us with a forest with more than two trees, rather than a single tree. While this was not the intended structure of the counterexample, it should still get full credit, since we did not specify the constraint in the problem that we require the two sides of the cut to induce connected subgraphs.

For instance, if the graph under consideration is the two-edge path (a, b, c) with the only edges $\{a, b\}$ and $\{b, c\}$, then you could just put b in one part of the partition and $\{a, c\}$ in the other part. You can't find any nontrivial MST in either part, and so when you try and join them, you can only add at most one edge, which will not be enough to connect the tree. In this case, the counterexample goes through independent of whatever weights you put on the edges: the contradiction is on the "spanning" rather than the "minimum."

The (incorrect) algorithm clearly follows the recurrence $T(n) = 2T(n/2)+O(m_{\text{cut}})$, since we recursively call the algorithm on graphs of half the size, and then scan the edges across the cut for the least-weight one. Note that we can use the recursion tree approach to show that the work done in each level equals the edges across the corresponding cuts, but we never ever check the same edge during two recursive calls. Therefore, the total work during all levels of the recursion tree is at most all the edges of the graph, which is $O(m)$, and the leaves of the recursion tree correspond precisely to single nodes of the graph, which are n constant time operations, for a total of $O(n)$ operations. It follows that the total complexity is $O(n + m)$.

3. **(20 points) Decimal to Binary.**

Solution. (a) The intended missing line of code is

Return: `fastmultiply(pwr2bin(`$\lfloor n/2 \rfloor$`), pwr2bin(`$\lceil n/2 \rceil$`))`

Here if the two inputs in `fastmultiply` have different lengths, then we zero-pad the smaller one so that they have the same length.

This works because `pwr2bin(`$\lfloor n/2 \rfloor$`)` returns the binary representation of $10^{\lfloor n/2 \rfloor}$, and similarly `pwr2bin(`$\lceil n/2 \rceil$`)` returns the binary representation of $10^{\lceil n/2 \rceil}$. Then multiplying them using `fastmultiply` yields precisely the binary representation of $10^{\lfloor n/2 \rfloor} \cdot 10^{\lceil n/2 \rceil} = 10^n$, which is what we were looking for. (**Think:** We needed the floor and ceiling functions to account for whether n is even or odd: do you see why this is critical?) For the complexity, if $T(n)$ denotes the time taken to compute the binary representation of 10^n, then we need to first compute $\lfloor n/2 \rfloor$. This takes at most $O(n)$ time. Then, we need to solve a subproblem by calling `pwr2bin` recursively on $\lfloor n/2 \rfloor$, which takes $T(\lfloor n/2 \rfloor)$ time. Note that this gives us the binary representation of $10^{\lfloor n/2 \rfloor}$, which has $\log(10^{\lfloor n/2 \rfloor}) = \Theta(n)$ bits. Now, we obtain `pwr2bin(`$\lceil n/2 \rceil$`)` from this, by keeping it as is, or possibly adding one (if n were odd). This takes at most $O(n)$ time (the complexity of adding 1 to a $\Theta(n)$-bit binary number). Finally, we need to multiply them together using `fastmultiply`. Note that because the arguments differ by at most 1, zero-padding by an additional prefixed 0 will not change the complexity of `fastmultiply` (**Think:** why?). Applying `fastmultiply` on two $\Theta(n)$-bit numbers will require $\Theta(n^{\log_2(3)})$ additional work. Therefore, the total work done is expressed as the recurrence

$$T(n) = T(n/2) + \Theta(n^{\log_2 3}),$$

where the smaller $O(n)$ and $O(1)$ work done in dividing, (possibly) adding one, and so on have been subsumed into the $O(n^{\log_2 3})$ term, and we have replaced $\lfloor n/2 \rfloor$ with $n/2$, because asymptotically the two recurrences will behave in the same way (**Think:** why?). Using the Master Theorem, note that here $a = 1$, $b = 2$, and $f(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.584})$, while $n^{\log_b a} = n^0 = 1 = o(f(n))$. It follows that $T(n) = \Theta(f(n)) = \Theta(n^{\log_2 3})$. As stated in the clarification on Piazza, using big-Oh, the same proof would go through in terms of only the upper bound.

(b) The intended missing line of code is

Return: `fastmultiply(dec2bin(`x_L`), pwr2bin(`$\lceil n/2 \rceil$`)) + dec2bin(`x_R`)`

To see that this is correct, note that we are told that x_L is precisely the first $\lfloor n/2 \rfloor$ digits of x, and x_R the last $\lceil n/2 \rceil$ digits, and therefore, $x = x_L \cdot 10^{\lceil n/2 \rceil} + x_R$. Furthermore, $10^{\lceil n/2 \rceil}$ has $\Theta(n)$ bits, as does x_L (**Think:** why?), and so we can use `fastmultiply` on the two of them after zero-padding. To compute the complexity, note once again that we start by splitting x into x_L and x_R and computing $\lceil n/2 \rceil$, which both take at most $O(n)$ time. We then call `pwr2bin` on it, which by the previous part takes $\Theta(n^{\log_2 3})$ time. We recursively call `dec2bin` on x_L, which is $T(\lfloor n/2 \rfloor)$ work, and then use `fastmultiply` on two $\Theta(n)$-bit numbers, which we know takes $\Theta(n^{\log_2 3})$ time. Finally, we compute `dec2bin`(x_R), which is another $T(\lceil n/2 \rceil)$ work, and finally, add the two together, which takes $O(n)$ time (for bitwise addition). Adding these up and noting that $T(\lfloor n/2 \rfloor) = T(\lceil n/2 \rceil) = T(n/2)$ asymptotically without changing the complexity, we get

$$T(n) = 2T(n/2) + \Theta(n^{\log_2 3}),$$

where the $n^{\log_2 3}$ has absorbed lower order terms. Once again, using the Master Theorem, we obtain a running time of $T(n) = \Theta(n^{\log_2 3})$ (**Think:** how did the Master Theorem give you the same asymptotics in this case as well, despite the different coefficient on the recursion?).

4. **(15 points) Almost Balanced Trees.**

Solution. The following algorithm determines whether a binary tree G is almost balanced.

```
isAlmostBalanced($x$)      % $x$ is the root of $G$
    if BalanceChecker($x$) = -1 return ‘‘NO’’
    else return ‘‘YES’’
    end if

BalanceChecker($x$)      % $x$ is the root of $G$
    if $x$ has no children return 1
    else
        hleft = BalanceChecker($x.left$), hright = BalanceChecker($x.right$)
        if hleft or hright = -1 return -1
        else if |hleft - hright| > 1 return -1
        else return 1 + max(hleft, hright)
        end if
    end if
```

The helper function `BalanceChecker` starts at the root, and recursively checks that the left and right subtree are both almost balanced; if either is not, then the whole tree G is also not almost balanced, and so `BalanceChecker` returns -1. Otherwise it checks the difference in heights of the left and right subtrees; if this difference is more than 1, then G is not almost balanced, and `BalanceChecker` returns -1. Otherwise it returns the height of the subtree rooted at the current node.

Importantly, we are not storing any value in any of the nodes; during computation at that node, we are recursively computing and immediately returning a value.

The `BalanceChecker` function could also just return the height and throw an exception for unbalanced trees (caught in the main function). This eliminates checking the subtree calls for error values (-1).

We don't know that the tree is balanced, so it's hard to set up a recurrence, but we don't need this. The algorithm is an ordinary tree traversal that reaches each node at most once, and the computations and tests take constant time at each node. The runnong time is thus $O(n)$.

5. **(10 points) Discrete Optimization.**

Solution. The following algorithm finds a local minimum of G.

```
FindLocalMinimum($x$)      % $x$ is the root of $G$
    if $x$ is an internal node
        set the left and right children of $x$ as $\ell$ and $r$
        compute query($x$), query($\ell$), query($r$)
        if query($x$) > query($\ell$) return FindLocalMinimum($\ell$)
        else if query($x$) > query($r$) return FindLocalMinimum($r$)
        else return $x$
        end if
    else return $x$      % $x$ is a leaf
    end if
```

The algorithm certainly finds a local minimum of G. Note that in order to decide whether a vertex is a local minimum, it suffices to check at most three neighbors in a complete binary tree – its parent, and its two children. To see that the algorithm works, observe that it starts at the root, and compares its value with that of its children. The algorithm maintains the following invariant: at any point, the node x under consideration has a value strictly less than that of its parent (if any) in the tree G. This can be seen by noting that the only way to move down the tree is for one of the conditions $x > \ell$ or $x > r$ to be true (**Think:** why does this imply the invariant?). Now note that the condition for the algorithm to return a node x is that both its children have value greater than its own value, in which case it is clearly a local minimum; or when it is not an internal vertex, meaning it can only be a leaf, at which point it has no children, and is therefore clearly a local minimum (by the invariant).

For the running time, G has $n = 2^d - 1$ nodes, so it must have 2^{d-1} leaves and $2^{d-1} - 1$ internal nodes. In particular, it has height d. The longest run of the algorithm is when it goes all the way down to a leaf before returning its value (otherwise, it terminates at some internal vertex, and it does not traverse as much of the tree). However, there is a unique path from any leaf to the root of a tree (**Think:** why?), of length d. At each node along this path, the algorithm queries at most three values, and therefore, the total number of queries made is $O(1) \cdot d = O(d) = O(\log n)$, and we are done.

6. **(20 points) Honest or Liar?**

Solution. (a) We start by pairing up the n people, and querying each pair, as stated. This amounts to $\lfloor n/2 \rfloor$ queries. At the end of this process, we throw out any pair where at least one person is accused of being a liar. As the problem states, in each such pair there must be at least one liar, so overall we throw out at least as many liars as honest people. Therefore the invariant that in the remaining set we have more honest people than liars is preserved.

Among the pairs claiming to be honest, let's reason about the numbers H and L of actual honest–honest vs. liar–liar pairs. We can't have $H < L$, since there would be more liars, even we had an extra odd person who is honest. Therefore, $H \geq L$, and if we arbitrarily select one person from each pair, we will have *at least* as many honest people as liars. But we need strictly more.

If $H + L$ is odd, $H \neq L$, so we must have $H > L$. So we will have more honest people just by choosing one from each pair, and we'll ignore any remaining odd person.

If $H + L$ is even, we include any remaining odd person together with the choices from each pair. We reason by cases: If $H > L$, the difference must be at least 2, since their sum is even. Then, choosing one from each pair we still have more honest people, even if we add an odd person who is a liar. If $H = L$, we must have an odd person who is honest (otherwise we wouldn't have more honest people), so by including them with the choice from each pair we get more honest people.

We have kept at most one person from each pair (since some pairs may have been thrown out), thus at most $n/2$, and that is all for n even. If n is odd, we may have kept the extra person, thus at most $(n-1)/2 + 1 = \lceil n/2 \rceil$. We have also shown the set of remaining people has precisely the same property as the original set, namely that strictly more than half the people are honest.

Therefore we are left with a problem that is about half the size (at most $\lceil n/2 \rceil$) of the original problem after just $\lfloor n/2 \rfloor$ tests.

(b) Let us first find one honest person among n people satisfying the given rule (strictly more than half are honest). We can follow the given procedure in the previous part recursively, halving the problem size every time, maintaining the invariant that there are strictly more honest people than liars. At some step of the recursion, we will reach a problem size of at most 2. Observe that for $n \leq 2$, everyone is honest, so we can just pick anyone. This therefore provides our recursive algorithm.

To analyze it, let $T(n)$ be the time required to find one honest person. Then, from the analysis in the previous part, we have $T(n) = T(\lceil n/2 \rceil) + n/2$, which is the same asymptotically as $T(n) = T(n/2) + n/2$. This solves (using any of the techniques we have seen in class – **Think:** how?) to

$$T(n) \leq \frac{n}{2} + \frac{n}{4} + \ldots = n \cdot \left(\frac{1}{2} + \frac{1}{4} + \ldots\right) = n.$$

So in $\Theta(n)$ steps (in fact, just n), we can identify one honest person. After that we can just pair her up with each of the other $n-1$ people and only listen to her assessment of their honesty to completely determine everyone else's identity. This takes another $n-1$ iterations (at most), and therefore the total number of tests taken is $\Theta(n)$.

It is interesting to note that this can be optimized a bit, by observing for instance that every pair that this final honest person was a part of had the other member honest as well, so we can save a few tests at the end, though asymptotically this is very small compared to the total number of people.