Homework 1 Solutions

CS 161 Summer 2019

Friday July 5th

Exercise 1 *Set-up (2 pts)*

(a) Please do the following, if you haven't already, then answer "I did the things" for part (a).

- Read the syllabus (at least up to the "Resources" section).
- Join the course Piazza and Gradescope (hint: links are in the syllabus).

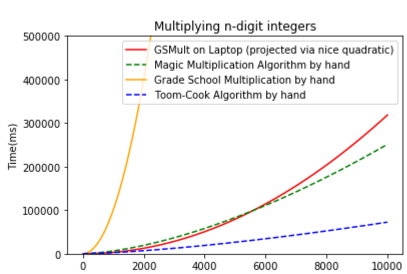
(b) Set up Jupyter notebooks and make sure you can get `lecture1_karatsuba.ipynb` up and running:

- Go to jupyter.org and choose either "Try it in your browser" or "Install the Notebook". Follow the instructions.
- Get the files `multHelpers.py` and `lecture1_karatsuba.ipynb` from Canvas ("Files" tab, in Lecture Materials>Lecture 01) and run `lecture1_karatsuba.ipynb` in the Jupyter Notebook.
  - If you are using the Notebook in your browser, you can upload the files by using File>Open... and then clicking Upload.
  - If you are installing the Notebook, you should install Python 3.3 or higher, and you may need to install matplotlib separately if you do not go through Anaconda to install Python.
- In lecture, we briefly mentioned that the Toom-Cook multiplication algorithm runs in time $O(n^{1.465})$. Find the cell where we compare the running time of grade-school multiplication to "Magic Multiplication" (Karatsuba). Add the following line, then re-run the cell to generate the new plot comparing Toom-Cook to the multiplication algorithms from class, and include a screenshot of it as your answer for part (b)!

```
plt.plot(nValsTmp, [ n**(1.465)/10 + 100 for n in nValsTmp], "--", color="blue",
                                         label="Toom-Cook Algorithm by hand")
```

Solution 1

(a) I did the things.



(b)

Exercise 2 *Basic Big-O (4 pts)*

Using the definitions of $O(\cdot), \Omega(\cdot)$, and $\Theta(\cdot)$, formally prove the following statements:

(a) $5\sqrt{n} + 3 = O(\sqrt{n})$.

(b) $n^{100} = \Omega(n)$.

(c) $2^{100} = \Theta(1)$.

(d) 4^n is **not** $O(2^n)$.

Solution 2

(a) Let $c = 8$ and $n_0 = 1$. We need to show that for all $n \geq n_0$, $0 \leq 5\sqrt{n} + 3 \leq c\sqrt{n}$. For the first inequality, observe that

$$0 < 8 \leq 5\sqrt{n} + 3$$

for all $n \geq 1 = n_0$. For the second, we have

$$5\sqrt{n} + 3 \leq 8\sqrt{n} = c\sqrt{n}$$

for $n \geq 1 = n_0$, as desired.

(b) Let $c = n_0 = 1$. We need to show that for all $n \geq n_0$, $0 \leq n \leq n^{100}$. The first inequality clearly holds since $n \geq 1 > 0$. For the second, observe that

$$\begin{aligned} 1 &\leq n \\ 1^{99} &\leq n^{99} \\ n &\leq n \times n^{99} = n^{100} \end{aligned}$$

as desired.

(c) Let $c = 2^{100}$ and $n_0 = 0$. We see that $2^{100} = O(1)$ because for all $n \geq n_0$,

$$0 \leq 2^{100} \leq 2^{100} \times 1$$

and $2^{100} = \Omega(1)$ because for all $n \geq n_0$,

$$0 \leq 2^{100} \times 1 \leq 2^{100}.$$

Therefore $2^{100} = \Theta(1)$.

(d) Assume to the contrary that 4^n is $O(2^n)$. Then there exists a constant $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$,

$$0 \leq 4^n \leq c2^n.$$

However, taking the logarithm of both sides of the right-hand inequality, we obtain

$$\begin{aligned} \log(4^n) &\leq \log(c2^n) \\ n\log(4) &\leq \log(c) + n\log(2) \\ 2n &\leq \log(c) + n \\ n &\leq \log(c). \end{aligned}$$

However, this is a contradiction because $n = n_0 + |\log(c)| + 1$ is larger than n_0 but violates this inequality. Thus 4^n is not $O(2^n)$.

Exercise 3 *More Big-O: True or False? (8 pts)*

In the following, suppose that $f(n)$ and $g(n)$ are strictly positive, strictly increasing functions. Formally prove or disprove the following statements:

(a) If $f(n) = O(g(n))$ then $100f^2(n) = O(g^2(n))$.

(b) There exists a constant $\alpha > 0$ such that $\log n = \Omega(n^\alpha)$. (You may assume, if it helps, that $\log n = O(n)$ but n is not $O(\log n)$.)

(c) If $f(n) = O(g(n))$ then $\log(f(n)) = O(\log(g(n)))$. (You may assume $\log(f(n)), \log(g(n)) > 0$.)

(d) If $f(n) = O(g(n))$ then $2^{f(n)} = O(2^{g(n)})$.

Solution 3

(a) The statement is true. Suppose that $f(n) = O(g(n))$. Then there exists $c > 0, n_0 \geq 0$ such that for all $n \geq n_0$, $0 \leq f(n) \leq cg(n)$. Choose $c' = 100c^2$. Clearly for all $n \geq n_0$, $0 \leq 100f^2(n)$ since $f(n)$ is strictly positive. Additionally,

$$\begin{aligned} f(n) &\leq cg(n) \\ f^2(n) &\leq c^2 g^2(n) \\ 100f^2(n) &\leq 100c^2 g(n) = c'g(n). \end{aligned}$$

Thus $100f^2(n) = O(g^2(n))$ as desired.

(b) The statement is false. Assume to the contrary that for some constant $\alpha > 0$, $\log n = \Omega(n^\alpha)$. Then there exists $c > 0, n_0 \geq 0$ such that for all $n \geq n_0$

$$0 \leq cn^\alpha \leq \log n.$$

Let's simplify this expression by defining $x = n^\alpha$. We obtain

$$\begin{aligned} cx &\leq \log x^{1/\alpha} \\ c\alpha x &\leq \log x. \end{aligned}$$

However, observe that this contradicts that x is not $O(\log x)$. In particular, if we let $c' = \frac{1}{c\alpha}$ and $x_0 = n_0^\alpha$, then the above implies $0 \leq x \leq c' \log x$ for all $x \geq x_0$, which would imply $x = O(\log x)$. Thus we have obtained a contradiction, so $\log n$ is not $\Omega(n^\alpha)$ for any constant $\alpha > 0$.

(c) The statement is true. Suppose that $f(n) = O(g(n))$. Then there exists $c > 0, n_0 \geq 0$ such that for all $n \geq n_0$, $0 \leq f(n) \leq cg(n)$.

By assumption $0 \leq \log(f(n))$. To complete the definition of big-O, we will find c' such that for all $n \geq n_0$, $\log(f(n)) \leq c' \log(g(n))$. Taking the logarithm of the equation in the previous paragraph, we see that

$$\log(f(n)) \leq \log(cg(n)) = \log c + \log(g(n)).$$

Define $c' = 1 + \frac{\log c}{\log(g(n_0))}$. Because $g(n)$ is increasing, $\log(g(n))$ is also increasing, so for all $n \geq n_0$ we have

$$\begin{aligned} \log(f(n)) &\leq \log c + \log(g(n)) \\ &\leq \log c \times \frac{\log(g(n))}{\log(g(n_0))} + \log(g(n)) = c' \log(g(n)). \end{aligned}$$

Therefore $\log(f(n)) = O(\log(g(n)))$ as desired.

(d) The statement is false. Consider $f(n) = 2n$ and $g(n) = n$. Clearly $f(n) = O(g(n))$ but we showed in exercise 2 that $2^{f(n)} = 2^{2n} = 4^n$ is not $O(2^n)$

Exercise 4 *Recurrence Relations (6 pts)*

Using either the Master Theorem or the "tree" method (show your work), give the best big-O bound possible on the following recurrences:

(a) $T(n) = 5T(\frac{n}{3}) + n$ with $T(n) = 1$ for $n < 3$.

(b) $T(n) = 2T(\frac{n}{2}) + n^2$ with $T(n) = 1$ for $n < 2$.

(c) $T(n) = T(n-2) + n$ with $T(n) = 1$ for $n < 2$.

(d) $T(n) = 4T(\sqrt[4]{n}) + \log n$ with $T(n) = 1$ for $n < 2$.

Solution 4

(a) We apply the Master Theorem with $a = 5, b = 3$, and $d = 1$. Since $a > b^d$ we obtain $T(n) = O(n^{\log_3 5})$.

(b) We apply the Master Theorem with $a = 2, b = 2$, and $d = 2$. Since $a < b^d$ we obtain $T(n) = O(n^2)$.

(c) The Master Theorem does not apply, because our problem size is not being cut down by a constant factor. However, we can consider the recursion tree. It has one sub-problem per level, where the 0th level has size n, the 1st level has size $n-2$, and in general the t'th level has size $n - 2t$. If n is even, the total work is

$$T(0) + \sum_{t=0}^{\frac{n}{2}-1} (n - 2t) = 1 + \sum_{i=1}^{\frac{n}{2}} 2i = 1 + \frac{n}{2}\left(\frac{n}{2} + 1\right) = O(n^2)$$

and similarly if n is odd, the total work is

$$\sum_{t=0}^{\frac{n-1}{2}} (n - 2t) = \sum_{i=0}^{\frac{n-1}{2}} (2i + 1) = \frac{n-1}{2} + 1 + \frac{n+1}{2}\left(\frac{n-1}{2}\right) = O(n^2).$$

Thus $T(n) = O(n^2)$.

(d) Again, the Master Theorem does not apply. However, the recursion tree has 1 problem of size n at the 0th level, 4 problems of size $n^{1/4}$ at the 1st level, 16 problems of size $n^{1/16}$ at the 2nd level, and in general 4^t problems of size $n^{1/4^t}$ at the t'th level. Thus the total amount of work per level is $4^t \log n^{1/4^t} = \log n$. Finally, we need to know how many levels there are. At the bottom level, we know we should have $n^{1/4^t} < 2$. Taking the logarithm of both sides, rearranging, and then taking the logarithm again, this is equivalent to

$$\begin{aligned} \frac{1}{4^t} \log n &< 1 \\ \log n &< 4^t \\ \log \log n &< 2t \\ t &> \frac{1}{2} \log \log n. \end{aligned}$$

Thus the total number of levels is $O(\log \log n)$ so overall $T(n) = O(\log n \log \log n)$.

Exercise 5 *Generalized Karatsuba (6 pts)*

In class, we saw that Karatsuba's Algorithm breaks up the problem of integer multiplication of two n-digit numbers into 3 sub-problems of size $\frac{n}{2}$. We also mentioned that the Toom-Cook Algorithm breaks up integer multiplication into 5 sub-problems of size $\frac{n}{3}$.

Suppose that, for any integer $k \geq 2$, we can break up one integer multiplication xy into $2k-1$ sub-problems of size $\frac{n}{k}$ in time $O(n)$. Furthermore, suppose that by simple addition and subtraction of the correct sub-problems, we can calculate $\frac{2n}{k}$-digit numbers $q_0, \ldots, q_{2k-2}$ such that $xy = q_{2k-2}10^{(2k-2)n/k} + \cdots + q_2 10^{2n/k} + q_1 10^{n/k} + q_0 10^0$.

(a) Argue that, for any constant k, the number of one-digit operations required to reassemble the sub-problems into the overall product xy is $O(n)$.

(b) Give a recurrence relation representing the run-time of this generalized-Karatsuba algorithm for arbitrary k, and solve the recurrence relation to give the tightest-possible big-O bound.

(c) Your friend, Ms. Take, is very excited about this problem and tells you she's come up with an $O(n \log n)$ algorithm for integer multiplication—the fastest ever discovered! Here is her reasoning:

   i. We know that for any k, we can break up an n-digit multiplication into $2k-1$ sub-problems of size $\frac{n}{k}$.

   ii. You showed in part (a) that recombining these sub-problems takes $O(n)$ time.

   iii. Therefore, just let $k = \sqrt{n}$. This gives us the recurrence relation $T(n) = (2\sqrt{n}-1)T(\sqrt{n}) + O(n)$.

   iv. This recurrence solves to $O(n \log n)$. First, notice that there are $\log \log n$ levels of the recursion tree, because that's how many times you have to take the square root of n to get down to $O(1)$. At the 0th level, there is 1 sub-problem of size n. At the 1st level, there are fewer than $2\sqrt{n}$ sub-problems of size $\sqrt{n}$. At the 2nd level, there are fewer than $2\sqrt{n} \times 2n^{1/4} = 4n^{3/4}$ sub-problems of size $n^{1/4}$. In general, at the t'th level, there are fewer than $2^t n^{1-\frac{1}{2^t}}$ sub-problems of size $n^{1/2^t}$, for a total of $2^t O(n)$ work at level t. Finally, observe that $\sum_{t=0}^{\log \log n} 2^t O(n) = O(2^{\log \log n} n) = O(n \log n)$, as claimed!

   What big conceptual error has Ms. Take made?

Solution 5

(a) We essentially have to perform three operations to recombine the sub-problemts: (1) calculate the q_i's, (2) multiply the q_i's by powers of 10, and (3) add up the resulting numbers. Step (1) is performed by adding and subtracting results from the appropriate sub-problems. There are only $2k-1$ sub-problems, each having about $\frac{2n}{k}$ digits in the result, so it should only take $O(n)$ one-digit operations to assemble a given q_i, and there are only a constant number of these to assemble.

   Similarly, for step (2), we only need to stick a linear number of zeros onto a constant number of q_i's, and for step (3) we need to add a constant number of $\frac{2n}{k}$-digit numbers. Thus overall we require $O(n)$ operations.

(b) Since we create $2k-1$ sub-problems of size $\frac{n}{k}$, and do linear work to reassemble the result, we have $T(n) = (2k-1)T(\frac{n}{k}) + O(n)$. Using the Master Theorem, we have $a = 2k-1, b = k$, and $d = 1$ so $a > b^d$ and we get $T(n) = O(n^{\log_k(2k-1)})$.

(c) Our result for part (a) assumed that k was a constant, which is not true if we set $k = \sqrt{n}$. In particular, it could conceivably take $O(n)$ work to construct a single q_i from the $2k-1$, $\frac{2n}{k}$-digit sub-problem results. Thus if the number of q_i's, $2k-1$, is not a constant, our $O(n)$ bound no longer holds.

Exercise 6 *Preview of Selection (6 pts)*

In lecture 03 we will discuss the selection problem: given an unsorted list of n elements and an integer k, return the k'th smallest element—i.e., when $k = 1$ return the min, when $k = n$ return the max, etc.

In this problem, we are instead given two **sorted** lists, each of length n; assume that all the elements are distinct. Give a divide-and-conquer algorithm that returns the k'th smallest element in the union of the two lists, in time $O(\log n)$. Provide an informal argument that your algorithm has the correct runtime, and a concise proof by induction that your algorithm returns the correct answer.

Solution 6

Our algorithm proceeds as follows: Consider the middle elements of the two sorted lists, e_1 and e_2. Suppose k is small—we're looking for something in the smaller half of the remaining elements. Then we *can't* be looking for the larger of the two middle elements e_i, because the other middle element, and everything before the middle elements, is smaller. Thus we can safely "throw out" element e_i *and* everything larger than it in the same list, and recurse. Similarly, if k is large—we're looking for something in the larger half of the remaining elements—then we can safely "throw out" the *smaller* middle element e_j and everything smaller than it in the same list. We recurse on the remaining elements, but also have to decrease k because we threw out a bunch of smaller elements. (Note that when we "throw out" part of the list, we don't need to make a copy of the list; we can just keep track of the first and last indices in each list that we're still considering.) In pseudocode:

Algorithm 1: SELECT(list1, list2, k)

```
n1 = len(list1)
n2 = len(list2)
if n1 == 0 or n2 == 0 then
    return the k'th element of the non-empty list
e1 = list1[floor(n1/2)]
e2 = list2[floor(n2/2)]
if k ≤ floor(n1/2) + floor(n2/2) + 1 then
    if e1>e2 then
        return SELECT(list1[:floor(n1/2)], list2, k)
    else
        return SELECT(list1, list2[:floor(n2/2)], k)
else
    if e1<e2 then
        return SELECT(list1[floor(n1/2)+1:], list2, k - floor(n1/2) - 1)
    else
        return SELECT(list1, list2[floor(n2/2)+1:], k - floor(n2/2) - 1)
```

This algorithm has the correct runtime because each sub-problem only takes constant time to calculate some indices and compare two elements, then makes a single recursive call. I.e., there is only one sub-problem per level and constant work per level in the recursion tree. Finally, the number of levels is $O(\log n)$ because each sub-problem cuts one of the two lists at least in half, so after at most $2 \log n$ levels, one of the list sizes must be reduced to 0.

To show the algorithm is correct, we will prove the following (strong) inductive hypothesis.

Inductive hypothesis: SELECT(list1, list2, k) returns the k'th smallest element when the total number of elements in the two lists is at most i.

Base case: When there is only one element between the two lists, one of the lists must be empty, so the algorithm correctly returns the k'th smallest element in the other list.

Inductive step: Assume the algorithm is correct when the total number of elements is at most i, and consider an input with $i + 1$ elements. If one of the lists is empty, then the algorithm correctly returns the k'th smallest element in the non-empty list. Otherwise, if $k \leq \lfloor n_1/2 \rfloor + \lfloor n_2/2 \rfloor + 1$, then the larger middle element e_i is at least the $k + 1$'th smallest. This holds because the lists are sorted, so both the other middle element, and everything preceding the middle elements must be smaller. Therefore the algorithm correctly returns the k'th smallest element of {list1, list2} which is the same as the k'th smallest element after removing e_i and everything after it.

Conversely, if $k > \lfloor n_1/2 \rfloor + \lfloor n_2/2 \rfloor + 1$, then the smaller middle element e_j is at most the $k - 1$'th smallest, because everything smaller than it must come before one of the middle elements. Therefore the algorithm correctly returns the k'th smallest element of {list1, list2} which is the same as the (k − # elements removed)'th smallest element after removing e_i and everything after it.

Conclusion: SELECT(list1, list2, k) correctly returns the k'th smallest element in the two lists.