CS 161 W19: Recitation 1 Solutions

January 2019

Exercise 0

For each of the following functions, prove whether $f = O(g)$, $f = \Omega(g)$, or $f = \Theta(g)$. For example, by specifying some explicit constants n_0 and $c > 0$ such that the definition of Big-Oh, Big-Omega, or Big-Theta is satisfied.

(a)	$f(n) = n\log(n^3)$	$g(n) = n\log n$
(b)	$f(n) = 2^{2n}$	$g(n) = 3^n$
(c)	$f(n) = \sum_{i=1}^{n} \log i$	$g(n) = n\log n$

Solution 0

1. $f(n) \in \Theta(g(n))$. Observe that $f(n) = n\log(n^3) = 3n\log n$. To prove big-Oh, chose any c above 3 (for example $c = 4$), then $f(n) = 3n\log n \leq 4n\log n = cg(n) \quad \forall n \geq n_0$ for any n_0 of your choice. To prove big-Omega, chose any c below 3 (for example $c = 2$), then $f(n) = 3n\log n \geq 2n\log n = cg(n) \quad \forall n \geq n_0$ of any n_0 of your choice.

2. $f(n) \in \Omega(g(n))$. Observe that $f(n) = 2^{2n} = 4^n$. Chose $c = 1, n_0 = 1$; then $f(n) = 4^n \geq 1 \times 3^n = cg(n) \quad \forall n \geq n_0$. To disprove Big-O, assume to obtain a contradiction that there exists some c such that for sufficiently large n,

$$4^n \leq c3^n$$
$$n\log 4 \leq \log c + n\log 3$$
$$n \leq \frac{\log c}{\log 4 - \log 3}$$

Thus for any c and n_0, if we take $n \geq n_0$ such that $n > \frac{\log c}{\log 4 - \log 3}$, we have a contradiction.

3. We proceed by manipulating the summation directly.

$$\sum_{i=1}^{n} \log i = \log 1 + \log 2 + \log 3 + ... + \log n$$
$$\sum_{i=1}^{n} \log i \leq \log n + \log n + \log n + ... + \log n$$
$$\sum_{i=1}^{n} \log i \leq n\log n$$

Thus we have proven Big-Oh (with an implicit constant $c = 1$).

In order to prove Big-Omega, we again unravel summation:

$$\sum_{i=1}^{n} \log i = \log 1 + \log 2 + \log 3 + ... + \log(\frac{n}{2}) + ... + \log n$$

Because log is strictly increasing, we know that that each one of the terms in the second half of the series is $> \log\left(\frac{n}{2}\right)$. Therefore:

$$\sum_{i=1}^{n} \log i \geq \frac{n}{2}\log(\frac{n}{2}) = \frac{n}{2}(\log(n) - \log(2))$$

So

$$\frac{n}{2}(\log(n) - \log(2)) \geq cn\log n$$
$$(1-2c)\log n \geq 1.$$

Now we can pick $c_0 = \frac{1}{4}, n_0 = 4$.

Exercise 1

Recall the Master theorem from lecture:
Theorem *Given a recurrence* $T(n) = aT(\frac{n}{b}) + O(n^d)$ *with* $a \geq 1$, *and* $b > 1$ *and* $T(1) = \Theta(1)$, *then*

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Give a Big-Oh expression for each of the following recurrence relations:

1. $T(n) = 3T(\frac{n}{2}) + \Theta(n^2)$
2. $T(n) = 4T(\frac{n}{2}) + \Theta(n)$
3. $T(n) = 2T(\sqrt{n}) + O(\log n)$

Solution 1

1. Using the Master Theorem, $a = 3, b = 2,$ and $d = 2$. Since $a = 3 < b^d = 4$, we fall into the second case. So, the runtime is $O(n^d) = O(n^2)$.
2. Using the Master Theorem, $a = 4, b = 2,$ and $d = 1$. Since $a = 4 > b^d = 2$, we fall into the third case. So the runtime is $O(n^{\log_b a}) = O(n^{log_2 4}) = O(n^2)$.
3. In this case the master theorem does not apply directly! In order to solve this question, we can either guess-and-check (using the substitution method from class) or use the following trick. Define $k = \log n$, so $n = 2^k$, and $\sqrt{n} = 2^{\frac{k}{2}}$. Then the recurrence relation becomes:

$$T(2^k) = 2T(2^{\frac{k}{2}}) + O(k)$$

Next, let $S(k) = T(2^k)$ so $S(\frac{k}{2}) = T(2^{\frac{k}{2}})$:

$$S(k) = 2S(\frac{k}{2}) + O(k)$$

Using the master theorem, we get

$$S(k) = O(k^d \log k) = O(k \log k) = O(\log n \log(\log n)).$$

Exercise 2

Given an array of integers $A[1 \dots n]$, compute a contiguous subarray $A[i \dots j]$ with the maximum possible sum. The entries of the array might be positive or negative.

1. What is the runtime of a brute force solution ?
2. The maximum sum subarray may lie entirely in the first half of the array or entirely in the second half. What is the third and only other possible case ?
3. Using the above observation, apply divide-and-conquer to arrive at a more efficient algorithm.
4. Give a rigorous proof by induction that your algorithm is correct. Make sure to explicitly state your inductive hypothesis, base case, inductive step, and conclusion.
5. What is the runtime of your solution ?
6. Advanced (Take Home) - Can you do even better using other non-recursive methods? ($O(n)$ is possible.)

Solution 2

1. The brute force approach involves summing up all possible $O(n^2)$ subarrays and finding the max amongst them for a total run time of $O(n^3)$. We can optimize this by pre-computing the running sums for the array so that we can find the sum of each subarray in $O(1)$ giving us a total run time of $O(n^2)$.
2. The maximum sum subarray can also overlap both halves; in other words, it passes through the middle element.
3. We divide the array into two and recurse to find the maximum sub array in the two segments. The best subarray of the third type consists of the best subarray that ends at $n/2$ and the best subarray that starts at $n/2$. We can compute these in O(n) time. To arrive at the final answer we return the max amongst these three types. This gives us a recurrence relation of the form $T(n) = 2T(n/2) + O(n)$.
4. First, to formalize the above algorithm:
   If you are given a 0 length array:

   return 0

   If you are given a 1 length array:

   if the single element is negative return 0, otherwise return the value of the element.

   If you are given an array of length $n \geq 2$:

   (a) Find the maximum subarray sum in the first half of the array by recursively passing the first $\frac{n}{2}$ items in the array to this algorithm

   (b) Find the maximum subarray sum in the second half of the array by recursively passing the last $\frac{n}{2}$ items in the array into this algorithm

(c) Find the maximum sum for arrays which cross the middle element. One example of an O(n) algorithm: Initialize the sub-array as starting and ending at the middle of the array. Progress the start of the sub-array left, one element at a time and keep track of the start that yields the largest sum. Progress the end of the sub-array right, one element at a time and keep track of the end that yields the largest sum. Return the sum of elements from the selected start to the selected end.

(d) return the maximum out of the best sum contained entirely in the first half of the array, the best sum contained in the last half of the array and the best sum spanning the middle of the array.

Now to prove the algorithm above is correct:
Base case: 0 length array and 1 length array are handled correctly in the algorithm.
Inductive hypothesis: For all j, $0 \leq j \leq n$ the algorithm returns the correct sum.
Inductive step: There are only 3 options for when where the maximum subarray could lie: either it lies entirely in the first half of the array, entirely in the second half of the array, or it spans the middle of the array.
Because these three options are exhaustive and mutually exclusive, if we were to arrive at the correct results for these three sub-problems and take their max that would be maximum sum for a subarray in the array.
We can arrive at the correct solution for both halves of the array by our inductive hypothesis, these arrays will have length $n/2 < n$
We know that our O(n) algorithm proposed above is correct as it checks over all possible subarrays spanning the middle and compares them all.
Therefore, our algorithm is correct.

5. Using the master theorem, we see that the run time of $T(n) = 2T(n/2) + O(n)$ solves to $T(n) = O(n \log n)$.

Exercise 3

You have seen how integer multiplication can be improved upon with divide-and-conquer. Let us see a more generalized example of matrix multiplication. Assume that we have matrices A and B and we'd like to multiply them.

1. What is the naive solution and what is its runtime? Think about how you multiply matrices.

2. Now, let us divide up the problem into smaller chunks:

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

   We now have a divide-and-conquer strategy! Find a recurrence relation that captures the running time of this strategy, and then solve the recurrence relation to find the big-Oh runtime of the algorithm.

3. Can we do better? It turns out we can, by calculating only 7 of the sub-problems:

$$\begin{aligned} P_1 &= A(F - H) & P_5 &= (A + D)(E + H) \\ P_2 &= (A + B)H & P_6 &= (B - D)(G + H) \\ P_3 &= (C + D)E & P_7 &= (A - C)(E + F) \\ P_4 &= D(G - E) \end{aligned}$$

   Then we can solve XY as

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

We now have a more efficient divide-and-conquer strategy! Find a recurrence relation that captures the running time of this strategy, and then solve the recurrence relation to find the big-Oh runtime of the algorithm.

Solution 3

1. The naive solution is that we will multiply row by column to get each element of the new matrix. Each new element of the new matrix is a sum of a row multiplied by a column, which takes n time, and there are n^2 new elements to compute, resulting in a runtime of $O(n^3)$.

2. The recurrence relation of this approach is $T(n) = 8T(\frac{n}{2}) + O(n^2)$ because you have 8 subproblems, and you are cutting subproblem size by 2, while doing n^2 additions to combine the subproblems. By the master theorem, $a = 8, b = 2, d = 2$, and this falls in the third condition, which results in a runtime of $O(n^3)$. Alas, this is no better than the naive strategy!

3. The recurrence relation of this algorithm is $T(n) = 7T(\frac{n}{2}) + O(n^2)$ because you have 7 subproblems, and you are cutting subproblem size by 2, while doing n^2 additions to combine the subproblems. By the master theorem, $a = 7, b = 2, d = 2$, and this falls in the third condition, which results in a runtime of $O(n^{\log_2 7}) = O(n^{2.81})$.