

15-251: Great Theoretical Ideas In Computer Science

Recitation 10 Solutions

Big Oh!

1. Which is asymptotically greater: $\log(\log^* n)$ or $\log^*(\log n)$?

Consider the values of the function at points $n = 2^{\text{STACK}_k}$ for various k . $\log \log^* 2^{\text{STACK}_k} = \log k$ and $\log^* \log 2^{\text{STACK}_k} = \log^* 2^{\text{STACK}_{k-1}} = k - 1$. Because $\log k < k - 1$ for $k > 1$, $\log \log^* n < \log^* \log n$.

Consider what happens at other points n . Both functions are increasing. That is, if $n' > n$, $\log \log^* n' > \log \log^* n$ and $\log^* \log n' > \log^* \log n$.

For $k \geq 7$, $\log(k + 1) < k - 1$. Take any $n > 2^{\text{STACK}_7}$, and let k be the largest integer such that $n > 2^{\text{STACK}_k}$. Then, n falls in the interval between 2^{STACK_k} and $2^{\text{STACK}_{k+1}}$, that is $2^{\text{STACK}_k} < n < 2^{\text{STACK}_{k+1}}$. Because the two functions are increasing,

$$\log \log^* n < \log \log^* 2^{\text{STACK}_{k+1}} = \log(k + 1) < k - 1 = \log^* \log 2^{\text{STACK}_k} < \log^* \log n$$

so $\log^* \log n$ is asymptotically greater than $\log \log^* n$.

2. Prove or disprove:

- $f(n) + O(f(n)) = \Theta(f(n))$

This notation means that the sum of $f(n)$ and any function in the set $O(f(n))$ is in the set $\Theta(f(n))$.

Take any function $g(n)$ in $O(f(n))$. By definition, there must be two constants $c > 0$ and $n_g \geq 0$ such that $g(n) \leq c \cdot f(n)$ whenever $n \geq n_g$. Then, $f(n) + g(n) \leq f(n) + c \cdot f(n) = (c + 1) \cdot f(n)$ whenever $n \geq n_g$. This means that $f(n) + g(n)$ is in $O(f(n))$.

Because $g(n) \geq 0$ (when dealing with asymptotic behavior, we usually restrict ourselves to non-negative-valued functions only), $f(n) + g(n) \geq f(n) \geq 1 \cdot f(n)$ whenever $n \geq 0$. Therefore, $f(n) + g(n)$ is in $\Omega(f(n))$.

Because the sum $f(n) + g(n)$ is in both $O(f(n))$ and $\Omega(f(n))$, it is in $\Theta(f(n))$. Because the choice of g was arbitrary, this holds for all functions g from $O(f(n))$: $f(n) + O(f(n)) = \Theta(f(n))$.

- $f(n) = O(f(n/2))$

Let $f(n) = 4^n$. Then $f(n/2) = 4^{n/2} = (\sqrt{4})^n = 2^n$. 4^n is not in $O(2^n)$.

- $f(n) + g(n) = O(\min(f(n), g(n)))$

Let $f(n) = n^2$ and $g(n) = n$. Then, $\min(f(n), g(n)) = n$, but it isn't true that $n^2 + n$ is in $O(n)$ (it is in $\Theta(n^2)$). What is true, on the other hand, is that $f(n) + g(n) = O(\max(f(n), g(n)))$, because $f(n) + g(n) \leq 2 \max(f(n), g(n))$ for all n .

- $f(n) = O(f(n)^2)$

If $f(n) = n^{-1}$, $f(n)^2 = n^{-2}$, and n^{-1} is not in $O(f(n)^{-2})$.

3. If $f(n) = O(g(n))$, is $g(n) = O(f(n))$?

No, if $f(n) = O(g(n))$, then there exist constants $c > 0$, n_f such that $f(n) \leq c \cdot g(n)$ whenever $n \geq n_f$. That is, g is an upper bound on f in this loose sense. This does not imply that f is any kind of upper bound on g .

However, what is true is that $g(n) \geq \frac{1}{c} \cdot f(n)$ whenever $n \geq n_f$ (rearrange the inequality in the previous paragraph). Since $c > 0$, $\frac{1}{c} > 0$. This means that $g(n) = \Omega(f(n))$.

Algorithms

4. Sub-linear Algorithms:

- Given an array of data, when can we perform a sub-linear search of the data?

If the array is sorted, it is possible to use binary search, which runs in logarithmic time — asymptotically faster than linear time.

- Given an unsorted array of data, can we sort it in sub-linear time?

Suppose A is an algorithm for sorting arrays of data in sub-linear time. Suppose A is given an array of length n . A cannot consider every element in the array: doing this takes at time proportional to n , the total number of elements. That is linear time, but A runs in sub-linear time.

Therefore, there must be some element in the array, call it e , that A never considered. A could not have made any decision based on the value of e , because the value of e always remained unknown to A . Therefore, the value of e can be set arbitrarily without affecting the outcome of running A on this array.

After A sorted the array, e ended up in some position. Call that position i . Then, e 's value can be set so that it belongs in position $i + 1$ in the final sorted array (or $i - 1$ if i is the last position). However, since A does not look at the value of e , it would still place e in position i . Therefore, A is incorrect.

This means that the running time of any algorithm for sorting data, expressed as a function of the input size n , is in the set $\Omega(n)$. Sorting requires at least linear time.

5. *Selection Sort: Given an array of n elements, traverse it to identify the smallest, and then recursively sort the remaining array. What is the recurrence $T(n)$ for selection sort?*

Only the recursive case will be given. It takes linear time to traverse the array of n elements and identify the smallest (which element is the smallest isn't known until all the elements in the array have been considered). After this element is set aside, the remaining array has size $n - 1$. Therefore, the recurrence relation for the running time of selection sort is $T(n) = cn + T(n - 1)$.

6. *Merge Sort: Given an array of $n = 2^k$ numbers, merge sort splits the array into two halves, recursively sorts both halves, and merges them. If the time taken to merge is just the size of the halves, write down the recurrence for $T(n)$.*

The algorithm first makes two recursive calls, each on inputs of size $\frac{n}{2}$. It then takes time proportional to the size of the halves (and therefore the original array) to merge the results. The recurrence relation for the running time of merge sort is therefore $T(n) = 2T\left(\frac{n}{2}\right) + cn$.

General Recurrences

Consider the following type of recurrence:

$$\begin{aligned}T(n) &= aT(n/b) + cn^k \\T(1) &= c\end{aligned}$$

for positive constants a, b, c, k . This recurrence corresponds to the time spent by an algorithm that divides the problem into a pieces of size n/b , solving each one recursively, and then doing cn^k work stitching them together.

7. If you are told that $a < b^k$, then can you find a Theta bound for $T(n)$? What happens if $a = b^k$? For simplicity, assume n is a power of b .

Unroll the recurrence. The first few steps are shown below (the recursive case is used in the form $T(n) = cn^k + aT(n/b)$ for convenience):

$$\begin{aligned} T(n) &= cn^k + aT(n/b) \\ &= cn^k + ac(n/b)^k + a^2T(n/b^2) \\ &= cn^k + ac(n/b)^k + a^2c(n/b^2)^k + a^3T(n/b^3) \\ &= \dots \end{aligned}$$

The first term can be written as $a^0c(n/b^0)^k$, and the second as $a^1c(n/b^1)^k$. Then, the general form for the i th term is $a^ic(n/b^i)^k = cn^k \left(\frac{a}{b^k}\right)^i$. In the last term, $i = \log_b n$. Therefore,

$$T(n) = \sum_{i=0}^{\log_b n} cn^k \left(\frac{a}{b^k}\right)^i = cn^k \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i$$

The sum is simply a finite geometric series.

When $a < b^k$, $a/b^k < 1$, and it is convenient to note that the finite geometric series is bounded above by the infinite geometric series with the same base (which simply has more terms). In this case, the infinite geometric series converges. The finite geometric series is always bounded below by 1, so

$$cn^k \times 1 \leq cn^k \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i \leq cn^k \sum_{i=0}^{\infty} \left(\frac{a}{b^k}\right)^i = cn^k \frac{1}{1 - a/b^k}$$

i.e. $T(n)$ is asymptotically bounded above and below by n^k . In other words, $T(n) = \Theta(n^k)$.

When $a = b^k$, $a/b^k = 1$, and every term of the series is 1. Then,

$$T(n) = cn^k \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i = cn^k \sum_{i=0}^{\log_b n} 1 = cn^k(1 + \log_b n)$$

and $T(n) = \Theta(n^k \log n)$.

8. Give the values for a , b , and k if the algorithm under consideration is merge sort. What asymptotic bound can you find for merge sort?

It was shown above that the recurrence relation for the running time of merge sort is $T(n) = 2T\left(\frac{n}{2}\right) + cn^1$. For this recurrence, $a = 2$, $b = 2$, and $k = 1$. $2 = 2^1$, so the running time of merge sort is in $\Theta(n^1 \log n)$.

9. How about the binary search algorithm for identifying if an element is present in a sorted array?

This algorithm performs a constant amount of work to check whether the current element under consideration is the one being searched for, or precedes, or follows the one being searched for. Then, based on the result of the check, it performs at most one recursive call on an array of half the size of the original input. Therefore, the recurrence relation for the running time of binary search is $T(n) = T(n/2) + cn^0$. In this recurrence, $a = 1$, $b = 2$, and $k = 0$. $1 = 2^0$, so the running time of binary search is in $\Theta(n^0 \log n) = \Theta(\log n)$.

10. Prove that, if $T(n) = n^5 + T(n - 1)$, and $T(1) = 1$, then $T(n) = \Theta(n^6)$.

Unrolling the recurrence yields the sum $n^5 + (n - 1)^5 + (n - 2)^5 + \dots + 2^5 + 1^5$. There are n terms, each of which is at most n^5 . Therefore, $T(n)$ is bounded above by $n \cdot n^5 = n^6$.

Unroll the recurrence again, but this time consider only the first $\frac{n}{2}$ of the terms. This sum is $n^5 + (n - 1)^5 + \dots + (\frac{n}{2})^5$. This is a lower bound on the value of $T(n)$, because $T(n)$ has all the terms in this sum, and then $\frac{n}{2}$ more. The value of each term is at bounded below by $(\frac{n}{2})^5$. There are $\frac{n}{2}$ terms total in the sum. Therefore, $T(n) \geq \frac{n}{2} \cdot (\frac{n}{2})^5 = \frac{n^6}{64}$.

Since $T(n)$ is bounded above by n^6 and below by $\frac{1}{64}n^6$, $T(n)$ is in $\Theta(n^6)$.