Design and Analysis of Algorithms CS3230

Divesh Aggarwal

August 2020

1 Preliminaries

All logarithms are base 2. When we write $T(n/a)$ for any integer $a > 1$, we mean $T(\lfloor n/a \rfloor)$.

2 Lecture 1

Here we solve some of the exercises in Lecture 1.

Exercise 1. *Solve the recurrence relation:*

$$T(i) \le C, \quad \textit{for } 1 \le i \le 5 \ ,$$

and

$$T(n) \le T(n/3) + T(2n/3) + Cn \ .$$

Solution.
As discussed in the lecture, our guess for the recurrence is $T(n) = O(n \log n)$. So lets try to prove this.

Claim 1. *$T(n) \le 2Cn \log n$ for all $n \ge 2$.*

Proof. We prove this by induction on n. The statement is true for $k = 2, 3, 4, 5$.
Induction hypothesis: We assume that the statement $2 \le k \le n-1$ for $n \ge 6$.
Using the induction hypothesis, we have that

$$\begin{aligned}
T(n) &\le T(n/3) + T(2n/3) + Cn \\
&\le 2C\frac{n}{3}\log\frac{n}{3} + 2C\frac{2n}{3}\log\frac{2n}{3} + Cn \\
&\le 2Cn\log n - Cn\left(\frac{2\log 3}{3} + \frac{4\log(3/2)}{3} - 1\right) \\
&\le 2Cn\log n \ .
\end{aligned}$$

□

Remark 1: The above statement does not hold for $n = 1$ since $\log 1 = 0$.
Remark 2: The first condition in the recurrence relation can always be satisfied by choosing a large enough constant C since $T(i)$ is bounded by a constant for $1 \le i \le 5$.

Exercise 2. *Solve the recurrence relation. Let a, b be constants.*

$$T(i) = \Theta(1) \ \ \textit{for} \ \ 1 \le i \le b \ ,$$

$$f(i) = \Theta(1) \ \ \textit{for} \ \ 1 \le i \le b \ ,$$

and

$$T(n) = aT(n/b) + f(n) \ .$$

Solution

Let us assume that $b^{L+1} \leq n < b^{L+2}$.

Claim 2. *For $1 \leq i \leq L+1$. $T(n) = a^i T(n/b^i) + \sum_{j=0}^{i-1} a^j f(n/b^j)$.*

Proof. We prove this by induction on i. For $i = 0$, we get $T(n) = T(n)$, which is trivially true.

Let us assume it is true for $i = k$. Thus,

$$T(n) = a^k T(n/b^k) + \sum_{j=0}^{k-1} a^j f(n/b^j) \ .$$

Writing $T(n/b^k)$ as $aT(n/b^{k+1}) + f(n/b^k)$ using the recurrence relation $T(n) = aT(n/b) + f(n)$ with n replaced by n/b^k, we get that

$$T(n) = a^k(aT(n/b^{k+1}) + f(n/b^k)) + \sum_{j=0}^{k-1} a^j f(n/b^j) = a^{k+1}T(n/b^{k+1} + \sum_{j=0}^{k} a^j f(n/b^j) \ .$$

□

Corollary 1.

$$T(n) = \Theta\left(\sum_{j=0}^{L} a^j f(n/b^j)\right) \ .$$

Proof. Substituting $i = L+1$ in the above claim gives us

$$T(n) = a^{L+1}T(n/b^{L+1}) + \sum_{j=0}^{L} a^j f(n/b^j) \ .$$

Notice that

$$a^{L+1}T(n/b^{L+1}) \leq \Theta(a^L) \ .$$

and the last term of the summation

$$a^L f(n/b^L) = \Theta(a^L) \ ,$$

since $b \leq n/b^L < b^2$, which is a constant. The given statement follows immediately from this observation. □

Exercise 3. *Masters theorem: Solve the above recurrence for the following three special cases:*

1. *There exists $\alpha < 1$ such that $\forall n \geq 1$, $af(n/b) \leq \alpha f(n)$.*
2. *There exists $\beta > 1$ such that $\forall n \geq 1$, $af(n/b) \geq \beta f(n)$.*
3. *$\forall n \geq 1$, $af(n/b) = f(n)$.*

Solution.

1. In this case, notice that for any j such that $0 \leq j \leq L$

$$\begin{aligned} a^j f(n/b^j) &= a^{j-1} \cdot af(n/b^j) \\ &\leq \alpha a^{j-1} f(n/b^{j-1} \\ &\leq \alpha^2 a^{j-2} f(n/b^{j-2} \\ &\leq \cdots \\ &\leq \alpha^i f(n) \ . \end{aligned}$$

Thus,

$$\sum_{j=0}^{L} a^j f(n/b^j) \leq \sum_{i=0}^{L} \alpha^i f(n) \leq \frac{f(n)}{1-\alpha} \ .$$

Moreover, since

$$\sum_{j=0}^{L} a^j f(n/b^j) \geq f(n) \ ,$$

we get that $T(n) = \Theta(f(n))$.

2. Similar to the above, we have that for any j such that $0 \leq j \leq L$

$$\begin{aligned} a^L f(n/b^L) &\geq \beta a^{L-1} f(n/b^{L-1}) \\ &\geq \beta^2 a^{L-2} f(n/b^{L-2}) \\ &\geq \cdots \\ &\geq \beta^j a^{L-j} f(n/b^{L-j}) \ . \end{aligned}$$

Thus,

$$\sum_{j=0}^{L} a^j f(n/b^j) \leq \sum_{i=0}^{L} a^L f(n/b^L) \frac{1}{\beta^i} = \Theta(a^L) = \Theta(a^{\log_b n}) \ .$$

Moreover, since

$$\sum_{j=0}^{L} a^j f(n/b^j) \geq a^L f(n/b^L) \ ,$$

we get that

$$T(n) = \Theta(a^{\log_b n}) = \Theta(2^{\log_b a \log_b n}) = \Theta(n^{\log_b a}) \ .$$

In the above, use the expression that is most suitable. Usually, the last one is the most useful.

3. Similar to the above, we see that for $0 \leq i \leq L$

$$a^i f(n/b^i) = f(n) \ .$$

Thus,

$$\sum_{j=0}^{L} a^j f(n/b^j) = (L+1) \cdot f(n) = \Theta(f(n) \log_b n) \ .$$

Note that you are welcome to use any variant of the Master's theorem proved in any of the textbooks. If you notice carefully, and check the proofs, there is not much difference between different versions.

3 Proving Correctness and Analysing Complexity of Recursive Algorithms

In this section, I list some general tips that might be helpful in proving correctness of recursive algorithms. This is particularly relevant for the algorithms we saw in Lectures 2, 3, 4, but maybe helpful for other recursive algorithms as well. Don't use these tips for Greedy Algorithms studied in Lecture 5. For those algorithms, the proof strategies will be covered in the corresponding lecture slides.

3.1 The general structure of a recursive algorithm

The recursive algorithm for solving an instance I of a problem P has the following general strategy.

Base Case: If the instance is small enough, then we can obtain the solution directly. Otherwise,

Reduce/Divide the given problem instance into solving a finite number of "smaller" problem instances $I_1, I_2, \ldots, I_k$ of the same problem P.

Solve each problem instance $I_1, \ldots, I_k$ recursively.

Combine/Use the solutions of problem instances $I_1, \ldots, I_k$ to get a solution to problem instance I.

Every recursive algorithm we saw in Lectures 2, 3, 4 followed this general strategy

Remark 1. There might be a problem for which it might not be possible to get a recursive algorithm directly, but it might be possible to convert this into a more general problem for which you can find a recursive algorithm. For example, if we are given a problem of finding a median of an n-element array, we don't know of a recursive algorithm that makes calls only to an algorithm for finding median on a smaller array.

But, what we do know is a recursive algorithm for finding the k-th smallest element (i.e., the input is the array and the number k) that makes recursive calls to an algorithm for finding a k'-th smallest element.

So, to solve the problem of finding a median, we instead solve a more general problem of finding a k-th smallest element for any input k and then replace k by $n/2$.

Remark 2. The harder step in any of the algorithms we have seen in Lectures 2, 3, and 4 is to come up with the correct recursive algorithm following the above structure. The correctness proof and complexity analysis are usually straightforward.

Examples. The following are some examples to illustrate the general strategy mentioned above.

1. In the MERGESORT algorithm for sorting $A[1 \ldots n]$, we "divide" the array into two halves, made two recursive calls to sort on $A[1 \cdots n/2]$ and $A[n/2+1, \ldots, n]$. Then we combined the two sorted arrays using the MERGE procedure.

2. In the k-th smallest element finding algorithm, we find an appropriate set of $n/5$ elements, make a recursive call to find the $n/10$-th smallest element. Then we find another set of at most $7n/10$ elements, make another recursive call to k-th smallest or $(k-r)$-th smallest element (where r is the rank of the pivot)

3. In the Subset Sum problem for solving SubSum(i, T) which is a boolean function deciding whether $X[1 \ldots i]$ has a subset that adds to T we reduce the problem to two recursive calls to SubSum$(i-1, T)$ and SubSum$(i-1, T-X[i])$. Then we use the result of the recursive calls to obtain the value of SubSum(i, T).

4. In the LCS problem, for solving LCS$[i, j]$ which finds the length of the LCS of $X[1 \ldots i]$ and $Y[1 \ldots j]$, we check whether $X[i] = Y[j]$ and based on that, either make one recursive call with LCS$[i-1, j-1]$ or make two recursive calls to LCS$[i-1, j]$ and LCS$[i, j-1]$. Then we use the result of the recursive calls to obtain the value of LCS$[i, j]$.

3.2 How to write a proof of correctness for a recursive algorithm

Here I will write how a formal proof of correctness for a general recursive algorithm will go, and highlight the steps that are required when you write a solution to a homework problem or an exam problem.

Remark 3. In lectures 3, 4 we saw dynamic programming/memoization which is a way to make these recursive algorithms faster by not solving any subproblem more than once. This speed-up has nothing to do with whether our recursive algorithm is correct or not, and so we are not talking about in this subsection.

The formal proof is by induction. We want to prove that for any instance I of a problem P, your algorithm produces the correct solution.

Base case. We need to show that our algorithm outputs the correct solution for the base case. This is usually obvious from how the base case is defined.

If you have defined the base case correctly, and correctly shown how your algorithm solves the base case, the proof is self-explanatory, and **you will not be required to provide this proof for homework or exams**. Whenever you write a base case, just spend a few minutes convincing yourself that the base case is correct.

Induction hypothesis. Assume that your algorithm is correct for any problem instance "smaller" than I. Here "smaller" can have different meanings and it is upto you to define. For example, for the LCS problem there are several suitable ways to define what a smaller instance is. We can define (i, j) is smaller than (k, ℓ) if $i < k$, or $i = k, j < \ell$ (i.e., the lexicographic ordering). Alternatively, we could define (i, j) is smaller than (k, ℓ) if $i + j$ is less than $k + \ell$. Both would work.

Induction argument. We need to show the following:

- $(I_1, \ldots, I_k)$ that we are making recursive calls to are indeed smaller than our instance I with respect to our definition of "smaller". If your "smaller" is defined properly, this will be easy to see and you will not need to prove it.

  This is something that is for you to check, and not required to prove formally. For example, if you wrote a recursive solution for Edit$[i, j]$, that made recursive calls to Edit$[i, j - 1]$ and to Edit$[i, j + 1]$, this would be a recipe for disaster (a.k.a. infinite loop).

- By induction hypothesis, we assume that the solutions of $(I_1, \ldots, I_k)$ are correct. You will then need to prove that the solution obtained indeed gives the correct proof for the instance I. For most examples that we saw in Lecture 2, 3, and 4, this is quite obvious **and does not require a proof**. For example SubSum(i, T) = SubSum$(i-1, T)$ OR SubSum$(i-1, T-X[i])$ can be easily explained in half a sentence by just stating that either $X[i]$ belongs to a subset that adds to T or it doesn't. You needn't worry about writing such a proof (but be sure it is half a sentence and not 5 sentences :)). If you want to be careful, just write such a half sentence to be safe.

  The only problems in lectures for which this proof required a little bit of work were LCS and Edit Distance. For both these problems, we proved a theorem in the class that shows this.

To summarize, if you have a correct recursive solution, **you only need to write a proof to show why a correct solution to problem instances $I_1, \ldots, I_k$ implies a correct solution to problem I.** There is no rule or protocol to write such a proof and every recursive solution will have a different argument. The only way to learn how to do this is practice enough problems.

3.3 How to analyze the time complexity of recursive algorithms

Let size of the given instance I be n and the size of the instances $I_1, \ldots, I_k$ that we make recursive calls to be $n_1, \ldots, n_k$.

Recursive/Backtracking algorithms without Dynamic Programming/Memoization Here, we get a recurrence for the time complexity

$$T(n) = T(n_1) + \cdots + T(n_k) + f(n) ,$$

where $f(n)$ is the total time for the reduction/division step that produces instances $I_1, \ldots I_k$ (this could be trivial for a lot of problems and could take just a constant amount of time) and the time taken to combine the solutions of $I_1, \ldots, I_k$ to get the solution to the instance I.

For example, for MergeSort, the reduction step takes constant time, since you just have to return the indices $(1, n/2)$ and $(n/2 + 1, n)$ that the recursive calls have to be made to, whereas for QuickSort the reduction step takes $O(n)$ time since we have to separate smaller than the pivot from the elements larger than the pivot. On the other hand, the combine step for MergeSort is the Merge operation that takes $O(n)$ time, whereas the combine step for QuickSort just doesn't have to do anything since we already have the elements smaller than the pivot separated from the elements larger than the pivot. So, $f(n)$ in both cases is $O(n)$.

For Edit Distance, LCS, Subset Sum, etc. (without DP/Memoization), $f(n)$ is a constant since both the divide/reduce step and the combine step takes a constant amount of time. For Edit Distance, the divide step just computes $i - 1, j - 1$ given i, j, and the combine step just has to compute the minimum of three different quantities.

Bottom-up Dynamic Programming. Here, instead of solving problem instance I, we make use of our recursive procedure to solve all problem instances smaller than I. So, the total time complexity is

$$B + \sum_{I'} f(\text{size}(I')) \ ,$$

where I' varies over all possible instances "smaller" than I, and B is the time taken to solve the base case (which is always much less than the other term and can be ignored). For example, for LCS and Edit Distance $f(n)$ is a constant, and hence the total time complexity is $O(nm)$, as nm is the total number of problem instances smaller than the given problem instance (given by $1 \leq i \leq m$ and $1 \leq j \leq n$).

Memoization. The time complexity of this is exactly the same as bottom-up DP. The time complexity is upper bounded by

$$B + \sum_{I'} f(\text{size}(I')) \ ,$$

where as above, I' varies over all possible instances "smaller" than I, and B is the time taken to solve the base case.

The reason for this is the following. In the entire execution of the memoization algorithm, any subproblem that is solved is included in the set of problems solved by the bottom-up DP algorithm, and every subproblem is solved at most once. So, instead of computing the time complexity of solving problem instance I, let us just overestimate it by computing the total time to solve problem instance and **all** instances "smaller" than I. Since each problem instance is solved only once, for evaluating the total time complexity, for every instance, we only need to account for the total time needed in addition to solving the corresponding subproblems.

Here, I am being slightly imprecise and I have ignored the time required to check whether a problem has already been solved or not, but that only contributes a constant factor, and can be ignored if we are interested in asymptotic bounds.

Just remember that the memoization time complexity is the same as the time complexity for the corresponding bottom-up Dynamic Programming Algorithm.