CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Lecture 5

September 19, 2019

Instructor: Sepehr Assadi

1 Solving Recurrences: Recursion Trees

Recall that in the last lecture, we wrote a *recurrence* $T(n)$ for the worst-case running time of the merge sort algorithm. The recurrence was $T(n) \leq 2T(n/2) + O(n)$ and we briefly pointed out $T(n) = O(n \log n)$. In this lecture, we show how to "solve" this (and other similar recurrences) using a general technique called *recursion trees*.

A recursion trees is a simple and pictorial tool to solve recurrences for runtime of algorithms (or at least devise a good guess on what the runtime should be). A recursion tree is a *rooted tree* with one node for each recursive subproblem. The *value* of each node is the amount of time spent on the corresponding subproblem *excluding* recursive calls. Thus, the overall running time of the algorithm is the sum of the values of all nodes in the tree. We now solve the recurrence for merge sort using recursion trees.

Let us write the recurrence for merge sort as $T(n) \leq 2 \cdot T(n/2) + c \cdot n$ where $c > 0$ is some *constant*: it is important to change the $O(n)$ notation at this point to $c \cdot n$ (both definitions are equivalent) so that the following equations are computed correctly. The tree for this recurrence is a *binary tree* (as the algorithm has *two* recursive subproblems) and can be defined as follows:

- The root has a value $c \cdot n$ because the work we do in the first level excluding recursive calls is at most $c \cdot n$; moreover, the root has two child-nodes corresponding to the two recursive calls;
- Each child-node of the root has a value $c \cdot n/2$ (as for them we work on an array of size $n/2$ and hence spend $c \cdot n/2$ time); moreover, each of these nodes has two further child nodes (with value $c \cdot n/4$ and so on and so forth).
- The above pattern is continued the same way until we reach the nodes with value $c \cdot 1 = c$ which form the leaf-nodes of this tree.

The following is an illustration of this tree.

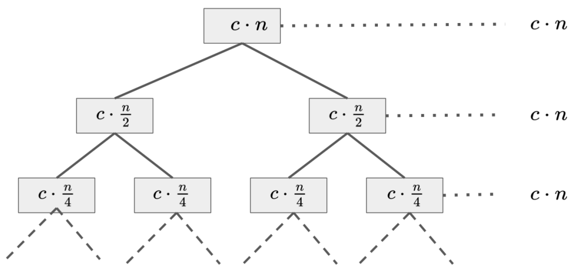


Figure 1: The first three layers of the recursion tree for merge sort. The *total time* spent for solving the subproblems at each level, excluding recursive calls, is also written on the right hand side.

By this definition, the nodes at level i of the tree (assuming the root is at level 0) have value $c \cdot n/2^i$. Moreover, there are 2^i nodes at level i of any binary tree. As such, the total time spent at each level of this tree is at most $c \cdot n$. Finally, this tree has at most $\lceil \log n \rceil$ levels because we stop the tree at a level with values c which is the layer i where $n/2^i \leq 1$ which implies $i = \lceil \log n \rceil$.

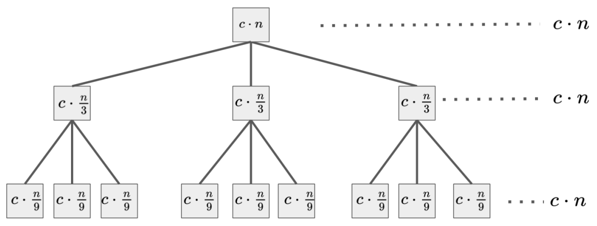
To conclude, the recurrence for merge sort takes $c \cdot n$ time per each level of the tree and there are only $\lceil \log n \rceil$ levels; hence, $T(n) \leq c \cdot n \cdot \lceil \log n \rceil = O(n \log n)$.

In general, using recursion trees is an easy and versatile method for solving recurrences once you get used to them. You should just remember that *always* to write the $O(f(n))$-notations as $c \cdot f(n)$ (i.e., replace $O(n^2)$ with $c \cdot n^2$). In chapters 4.3, 4.4, and 4.5 of the CLRS book, three different methods for solving recurrences, including the recursion tree method are discussed. The other two methods are induction (as always!) and the master theorem; we will (most likely) not discuss these methods in this course explicitly, however you are encouraged to take a look at these methods as well – moreover, even though none of your homeworks or exams will ask you a question about these methods, you are allowed to use either method instead of recursion trees when writing your solutions (just make sure you are comfortable with those techniques if you decide to do so).

More Examples. Before we move on, let us solve a couple of more recurrences. A typical question for recurrences is as follows: Find the *tightest* bound for the following recurrences (you do *not* need to prove that your bound is tightest).

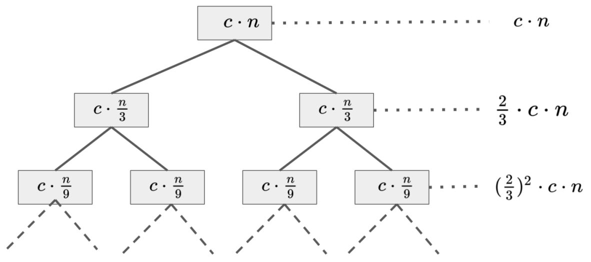
$$T(n) \leq 3T(n/3) + O(n) \qquad T(n) \leq 2T(n/3) + O(n) \qquad T(n) \leq 3T(n/3) + O(n^2).$$

- $T(n) \leq 3T(n/3) + O(n)$. We first replace $O(n)$ with $c \cdot n$ and now solve for the recurrence. The first three level of the recursion tree for this recurrence are:



  In particular, at level i of the tree, we have 3^i nodes, each with a value of $c \cdot \frac{n}{3^i}$. This means that the total value for each level is $3^i \cdot c\frac{n}{3^i} = c \cdot n$. The depth of the tree (total number of levels) is also $\lceil \log_3 n \rceil$. Hence, the total value of the tree is $T(n) \leq c \cdot n \cdot \lceil \log_3 n \rceil = O(n \log n)$.

- $T(n) \leq 2T(n/3) + O(n)$. We again replace $O(n)$ with $c \cdot n$ and solve for the recurrence using the following recursion tree:



  At level i of the tree, we have 2^i nodes each with a value of $c \cdot \frac{n}{3^i}$, hence the total value of level i is $(\frac{2}{3})^i \cdot c \cdot n$ (note that unlike the previous cases, this time the value of a level depends on its index).

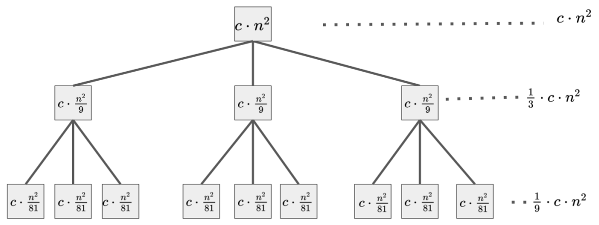
The total number of levels in this tree is also $\lceil \log_3 n \rceil$ (but as it will become evident, we actually do not need this). As such, the total value of the tree is:

$$\sum_{i=0}^{\lceil \log_3 n \rceil} (\frac{2}{3})^i \cdot c \cdot n = c \cdot n \sum_{i=0}^{\lceil \log_3 n \rceil} (\frac{2}{3})^i \leq c \cdot n \sum_{i=0}^{\infty} (\frac{2}{3})^i = c \cdot n \cdot \frac{1}{1 - 2/3} = 3 \cdot c \cdot n.$$

(the second last equality is by using the formula for sum of geometric series)

Hence, $T(n) = O(n)$ in this case.

- $T(n) \leq 3T(n/3) + O(n^2)$. We replace $O(n^2)$ term with $c \cdot n^2$ and write the following recursion tree:



At level i of the tree, there are 3^i nodes each with a value of $c \cdot (\frac{n}{3^i})^2$, hence the total value of level i is $(\frac{1}{3^i}) \cdot c \cdot n^2$. Again, there are $\lceil \log_3 n \rceil$ levels (and again this is not needed) and thus the total value of the tree is:

$$\sum_{i=0}^{\lceil \log_3 n \rceil} (\frac{1}{3^i}) \cdot c \cdot n^2 = c \cdot n^2 \cdot \sum_{i=0}^{\lceil \log_3 n \rceil} (\frac{1}{3^i}) \leq c \cdot n^2 \cdot \sum_{i=0}^{\infty} (\frac{1}{3^i}) = c \cdot n^2 \cdot \frac{1}{1 - 1/3} = \frac{3}{2} \cdot c \cdot n^2.$$

(again, the second last equality is by sum of the geometric series)

Hence, $T(n) = O(n^2)$ in this case.

2 Another Sorting Algorithm: Quick Sort

Another efficient sorting algorithm is quick sort. Quick sort is also based on the technique of divide and conquer. Remember the technique: we partition the instance into several *disjoint* subproblems, we solve each subproblem recursively, and we combine the solutions together. In the merge sort algorithm, the partitioning step was trivial (pick the first and second half of the array simply) and the main idea was on how to combine the solutions (using the merge algorithm). On the other hand, in quick sort, the main step is in the partition part (using an algorithm called partition) and after that there is essentially no combine step.

The Partition Algorithm

The partition algorithm solves the following problem: Given an array $A[1:n]$ and an index p in $[1:n]$ (called the *pivot*), *rearrange* A such that $A[p]$ is in its correct position q in the sorted order, and in the rearranged array, for all $i < q$, $A[i] \leq A[q]$ and for all $j > q$, $A[j] \geq A[q]$.

Example: For an input array $A = [20, 30, 70, 10, 80, \underline{40}, 50, 60]$ and $p = 6$ (the underlined index, i.e., $A[6] = 40$), the output can be any of the following arrays:

$[20, 30, 10, \underline{40}, 80, 50, 60, 80]$ $\quad$ $[10, 20, 30, \underline{40}, 50, 60, 70, 80]$ $\quad$ $[2, 3, 1, \underline{40}, 70, 60, 50, 80]$ $\quad$ $\cdots$

The important thing to note that the pivot element should always be placed in its correct position in the sorted order; for instance in the examples above, we will always place the number 40 in the 4-th position of the output array.

We now design an algorithm for this problem. The idea is to first swap $A[p]$ with the last element of the array, i.e., $A[n]$ for simplicity. So from now on, we will work with $A[n]$ as the pivot and can forget about p. Then go over all the elements and count how many of them are smaller than $A[n]$ to be able to find in which position we should place $A[n]$. During the same procedure, we should also start swapping elements that are smaller than $A[n]$ with the larger ones in a way to ensure that at the end, all smaller elements appear before the correct position of $A[n]$. There is a very slick way to do this using a single for-loop iteration as we describe below.

Partition Algorithm: The input is an array $A[1:n]$ and a pivot p in $[1:n]$.

```
1. Swap $A[p]$ and $A[n]$ with each other. Let $j = 1$.
2. For $i = 1$ to $n - 1$ do as follows:
    (a) If $A[i] < A[n]$, then swap $A[i]$ and $A[j]$ and let $j \leftarrow j + 1$.
3. Swap $A[n]$ and $A[j]$.
```

Example: Let us consider running this algorithm on the array $[20, 50, 70, 80, 10, 40, 30, 60]$ with pivot $p = 6$ ($A[6] = 40$) again. In the following, the underline denote the index i of the for-loop (before being incremented in that iteration) and the overline corresponds to index j.

- At the beginning (after swapping $A[6]$ with $A[8]$):
$$A = [\overline{\underline{20}}, 50, 70, 80, 10, 60, 30, 40]$$
- In iteration $i = 1$ we swap $A[i = 1]$ with $A[j = 1]$ (!) since $A[1] < A[8]$ and increase j to 2.
$$A = [\underline{20}, \overline{50}, 70, 80, 10, 60, 30, 40]$$
- In iteration $i = 2$ we do nothing since $A[i = 2] > A[8]$:
$$A = [20, \overline{\underline{50}}, 70, 80, 10, 60, 30, 40]$$
- In iteration $i = 3$ we do nothing since $A[i = 3] > A[8]$:
$$A = [20, \overline{50}, \underline{70}, 80, 10, 60, 30, 40]$$
- In iteration $i = 4$ we do nothing since $A[i = 4] > A[8]$
$$A = [20, \overline{50}, 70, \underline{80}, 10, 60, 30, 40]$$
- In iteration $i = 5$ we swap $A[i = 5]$ with $A[j = 2]$ since $A[5] < A[8]$ and increase j to 3:
$$A = [20, 10, \overline{70}, 80, \underline{50}, 60, 30, 40]$$
- In iteration $i = 6$ we do nothing since $A[i = 6] > A[8]$:
$$A = [20, 10, \overline{70}, 80, 50, \underline{60}, 30, 40]$$
- In iteration $i = 7$ we swap $A[i = 7]$ with $A[j = 3]$ since $A[7] < A[8]$ and increase j to 4:
$$A = [20, 10, 30, \overline{80}, 50, 60, \underline{70}, 40]$$
- Finally, we swap $A[n]$ with $A[j = 4]$:
$$A = [20, 10, 30, \overline{40}, 50, 60, 70, 80]$$

Remark: This algorithm is *not* for sorting the array (yet) as can be seen by the example above.

Proof of Correctness: We now prove that the partition algorithm correctly partitions any array $A[1:n]$ and given pivot p for all values of n. This is done using an induction over the index i of the for-loop of the array. In the following, we use j_i to denote the value of index j at the end of the iteration i.

Our induction hypothesis is that after every iteration i of the partition algorithm, the *current* array $A[1:j_i-1]$ contains all the elements smaller than $A[p]$ in the *original* array $A[1:i]$. Before proving the induction hypothesis, let us note why this concludes the proof. For $i=n$, the induction hypothesis implies that all the elements smaller than $A[p]$ (in the original array) are now placed in $A[1:j_n-1]$. The algorithm then swaps $A[j_n]$ with $A[n]$ (recall that $A[n]$ contains the element $A[p]$ in the original array). Since $A[j_n] \geq A[n]$ (by induction hypothesis), we now know that all elements in $A[1:j_n-1]$ are smaller than $A[j_n]$ (again $A[j_n]$ now contains the pivot $A[p]$) and all elements in $A[j_n+1:n]$ are at least as large as $A[j_n]$, as desired. This also implies that we placed $A[p]$ in its correct position which is $A[j_n]$, proving the correctness.

We now prove the induction hypothesis. The base case of $i=1$ is as follows: If $A[1] < A[n]$, we swap $A[1]$ with $A[1]$ (again!) and increase j by one, i.e., $j_1 = 2$ – this ensures that that the induction hypothesis holds in this case as now $A[1]$ contains the only element smaller than $A[n]$ in $A[1:1]$. Otherwise, if $A[1] \geq A[n]$ we do nothing which again ensures the induction hypothesis. The proof of the induction step is also similar: Suppose this is true up until iteration i and we prove it for iteration $i+1$:

- If $A[i+1] < A[n]$, we swap $A[i+1]$ and $A[j_i]$ and let $j_{i+1} = j_i + 1$: by induction hypothesis, $A[1:j_i-1]$ contains all the smaller elements in $A[1:i]$ and thus $A[j_i]$ is at least as large as $A[n]$. Hence, after this swap $A[1:j_i] = A[1:j_{i+1}-1]$ contains all the smaller elements from $A[1:i+1]$ proving the step.
- If $A[i+1] \geq A[n]$, we do nothing and $j_{i+1} = j_i$: in this case, since by induction $A[1:j_i-1]$ contained all the smaller elements in $A[1:i]$ and $A[i+1]$ is *not* smaller, $A[1:j_{i+1}-1]$ will continue to contain all the smaller elements in $A[1:i+1]$.

This proves the induction step and concludes the correctness proof of this algorithm.

Runtime analysis: The runtime is clearly $\Theta(n)$ because we run a simple $n-1$ iteration loop and each iteration takes $\Theta(1)$ time.

The Quick Sort Algorithm

Considering all we learned about the partition algorithm and the divide and conquer technique, the way the quick sort algorithm should work is almost straightforward: we pick an *arbitrary* pivot and partition the array around the pivot using the partition algorithm; we then recursively sort the array for the two different subarrays consisting of smaller elements than pivot and larger ones that are identified already by partition. The algorithm is formally as follows:

Quick Sort Algorithm: The input is an array $A[1:n]$.

1. If $n=0$ or $n=1$ return the current array.
2. Let $p=n$ (or *any* other index – at this point, this does *not* matter).
3. Use the partition algorithm to partition A with pivot p. Let q be the correct position of $A[p]$ in the sorted array computed by the partition algorithm.
4. Recursively quick sort $A[1:q-1]$ and $A[q+1:n]$.

Example: Let us consider running the quick sort algorithm on the array $[20, 50, 70, 80, 10, 40, 30, 60]$ (by always picking the last element of the array as pivot in the recursive calls). In the following, the underline denote the pivot element.

- We pick 40 as pivot and this partitions the array into two arrays for recursive calls:

$$[20, 50, 70, 80, 10, 60, 30, \underline{40}] \quad \Longrightarrow \quad [20, 10, 30, \underline{40}, 50, 70, 80, 60]$$

- We pick 30 as pivot in the first recursive call (over $[20, 10, 30]$) and 60 as pivot in the second recursive call (for $[50, 70, 80, 60]$):

$$\begin{aligned}
&[20, 10, \underline{30}] \quad \Longrightarrow \quad [20, 10, \underline{30}] \\
&[50, 70, 80, \underline{60}] \quad \Longrightarrow \quad [50, \underline{60}, 70, 80] \\
&[20, 10, \underline{30}, 40, 50, 70, 80, \underline{60}] \Longrightarrow [20, 10, \underline{30}, 40, 50, \underline{60}, 70, 80]
\end{aligned}$$

- By continuing like this, we sort the entire array.

Proof of Correctness: Proof is by ... induction! (Seriously, almost all divide and conquer algorithms are analyzed using induction). Our inductive hypothesis is simply that the quick sort correctly sorts any array of length n. The base case for $n = 0$ and $n = 1$ follows immediately from the algorithm since any array of size 0 or 1 is sorted anyway.

We now prove the induction step: suppose quick sort can sort all arrays of length $n \leq k$ and we prove it also correctly sorts any array of size $n = k + 1$. Let A be any array of size $n = k + 1$. Quick sort first runs partition (with an arbitrarily chosen pivot) to partition the array into $A[1 : q - 1]$, $A[q]$ and $A[q + 1 : n]$, where $A[q]$ is in its correct position and every element in $A[1 : q - 1]$ is at most as large as $A[q]$ while every element in $A[q + 1 : n]$ is at least as large as $A[q]$. The algorithm then recursively sorts $A[1 : q - 1]$ and $A[q + 1 : n]$ – since size of both array is smaller than $k + 1$, by induction hypothesis, both arrays are sorted correctly. By the guarantee of the partition algorithm and this sorting step, we have $A[1 : q - 1]$ is sorted array of elements $\leq A[q]$, and $A[q + 1 : n]$ is the sorted array of elements $\geq A[q]$; hence A is also now sorted. This proves the induction step and the correctness of the algorithm.

Runtime of Quick Sort: We will analyze the runtime of the quick sort in the next lecture.