CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 4

January 24, 2022

Instructor: Sepehr Assadi

1 Recursive Algorithms

A recursive algorithm is an algorithm that may call itself on smaller parts of the inputs[1]. An "algorithmic variant" of induction can be seen as *recursive algorithms*. In induction, the main thing we had to do, beside setting up a base case, was to show how use correctness of statement for smaller values to prove the statement for larger values. Similarly, in a recursive algorithm, "all" we have to do is to show how to use the algorithm for smaller inputs to get an algorithm for larger inputs as well. Let us see a simple example of a recursive algorithm: the *binary search* algorithm.

Binary Search Algorithm. Binary search allows us to determine whether a given integer exists in a sorted array of length n in only $O(\log n)$ time (assuming the array is already in the memory and we do not need to 'read' it of course).

Binary Search Algorithm: The input is a sorted array $A[1:n]$ of integers and a single target integer t.

1. If $n = 0$ return 'No' (this is the base case of this recursive algorithm).
2. Let $m = \lfloor n/2 \rfloor$:
   - if $A[m] = t$, output 'Yes' and terminate;
   - if $A[m] > t$, recursively solve the problem for t in the array $A_{<m} := A[1:m-1]$;
   - if $A[m] < t$, recursively solve the problem for t in the array $A_{>m} : A[m+1:n]$.

Proof of Correctness: The proof is by induction on size of the array A, i.e., n. The base case of the induction is when $n = 0$ for which the algorithm is definitely correct as an array of size 0 contains no element. Now suppose by induction that the algorithm correctly solves the problem on every array of size up to i and we prove it for $i + 1$.

Consider an array A of size $n = i + 1$ and define $m = \lfloor n/2 \rfloor$ as in the algorithm. If $A[m] = t$, the algorithm correctly returns t belongs to the array. Otherwise, if $A[m] > t$, since A is sorted in non-decreasing order, we know that all entries $A[j]$ for $j > m$ are also *larger* than t; so if t belongs to A it can only because t also belongs to $A_{<m}$ defined in the algorithm. By induction hypothesis, the algorithm correctly determines whether t belongs to $A_{<m}$ or not, hence proving the induction step in this case also. The proof of the case $A[m] < t$ is symmetric.

Runtime Analysis. Let $T(n)$ denote the worst-case runtime of the algorithm on any input of size n. We write the following **recursion** for the runtime of the algorithm:

$$T(n) \leq T(n/2) + \Theta(1)$$

(since we spend $\Theta(1)$ time to decide whether we terminate or recurse on an array of size $n/2$)

$$T(1) = \Theta(1). \qquad \text{(base case)}$$

[1]Smaller or different input is the key here; calling an algorithm recursively on the same input is definitely going to result in an infinite loop.

To solve this recursion, we do as follows. Firstly, let k be the *smallest integer* such that $n \leq 2^k$. So, by definition, $k = O(\log n)$. Moreover, define the recursion:

$$S(k) = S(k-1) + \Theta(1)$$
$$S(0) = \Theta(1).$$

We claim that $T(n) \leq S(k)$; this can be proven for instance by induction since, $T(n) \leq T(n/2) + \Theta(1) \leq S(k-1) + \Theta(1) = S(k)$.

Now, using the substitution, we can write:

$$S(k) = S(k-1) + \Theta(1) = S(k-2) + \Theta(1) + \Theta(1) = \ldots = \underbrace{\Theta(1) + \cdots + \Theta(1)}_{k} = \Theta(k).$$

This implies that $T(n) = O(k)$ (notice that we switched from Θ-notation to O-notation simply because the inductive proof above gives us an inequality for $T(n)$ not an equality). Finally, since $k = O(\log n)$, we also have $T(n) = O(\log n)$.

Before we move on, let us mention two important thing about analyzing recursive algorithm:

- Firstly, the proof of correctness of recursive algorithms is always via induction. Moreover, unlike many other algorithms that we need to be creative in specifying the exact statement we want to prove via induction, for recursive algorithms, the statement is almost always that "the algorithm is correct on every input".
- Secondly, the runtime analysis of recursive algorithms in this course is always based on writing a recurrence formula for the worst-case runtime (which is a rather mechanical task) and then *solving* the recurrence to get a closed-form formula instead of a recursive one. We will talk more about solving recurrences next week.

2 Divide and Conquer Algorithms: Merge Sort

A special case of recursive algorithms is *divide and conquer*. The overall idea is as follows:

1. Divide the given instance of the problem into several smaller instances of the same problem.
2. Recursively solve the problem on each smaller instance.
3. Combine the solutions for the smaller instances into the final solution.

If the size of any instance falls below some constant threshold, we abandon recursion and solve the problem directly, by brute force, in constant time.

The proof of correctness of a divide and conquer algorithm, similar to other recursive algorithms, almost always requires induction. Analyzing the running time requires setting up and solving a recurrence as we already saw for binary search above. We now introduce a canonical example of a divide and conquer algorithm for the *sorting* problem, one of the most important problems in algorithm design.

Sorting Problem. Let us start with the definition of the problem (recall that this is always the first step of algorithm design). *Sorting* is the process of placing elements from a collection in some kind of order. For example, a list of words could be sorted alphabetically or an array of numbers could be sorted by their values. Sorting is a key primitive in computer science with various applications: the most basic one being a necessary step for efficient *searching* (if you really need a compelling reason as an application of sorting, imagine looking up a some words in a dictionary that contains words in no particular order).

More formally, we define sorting as the following two problems:

Problem 1 (Sorting)**.**

- **Input:** An array A of n integers;
- **Output:** The same array A ordered in *non-decreasing*[2] order of its entries.

Note that we defined a very basic variant of sorting above; it turns out this is already enough to capture most other interesting sorting scenarios (sorting with different objective orders, sorting based on a particular key in <key, value> pairs, etc.).

There are numerous efficient and not-so-efficient algorithms for sorting. We will see several canonical examples of these algorithms in this course. In this lecture, we focus on *merge sort*, an efficient sorting algorithm using the divide and conquer technique.

2.1 Merge Sort

Recall the 3 steps of dividing, recursively solving, and combining the solutions in a divide and conquer algorithm. The second step is always the same across the algorithms, so our goal is to design the first and third step for our particular approach. For the merge sort problem, almost all the work is done in the third step, using an algorithm called **Merge**, that we first define.

The Merge Algorithm

The merge algorithm solves the following problem. Suppose we are given an array $A[1:n]$ where $A[1:\lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1 : n]$ are *sorted separately.* The goal of this algorithm is to sort the entire array A.

Example 1: For instance, the input to merge can be the following array $A = [5, 6, 7, 8, 1, 2, 3, 4]$. The first half of the array is $[5, 6, 7, 8]$ and the second one is $[1, 2, 3, 4]$ – each half is sorted separately but together they are not sorted anymore. The goal is then to return the array $[1, 2, 3, 4, 5, 6, 7, 8]$.

Example 2: Another example is the input $A = [1, 2, 7, 8, 3, 4, 5, 6]$ – again the first half and the second half are sorted separately but the whole array is not. The goal is then to return the array $[1, 2, 3, 4, 5, 6, 7, 8]$.

Merging an array that each of its half are sorted seems like an easier task than sorting the array from the scratch (after all, some part of the work is already done!). Intuitively, we can simply do as follows: We pick another empty array B; then we go over the elements in $A[1:\lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1 : n]$ one by one simultaneously and place the smaller one in array B and move the pointer in the array where this element come from one step further and continue like this.

We formalize the algorithm as follows:

Merge Algorithm: The input is an array A where $A[1:\lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1 : n]$ are sorted separately.

```
1. Create an empty array $B$ and two pointers $p_1 = 1$ and $p_2 = \lfloor n/2 \rfloor + 1$.
2. For $i = 1$ to $n$ do as follows:
    (a) If ($p_1 \neq \lfloor n/2 \rfloor + 1$ AND $A[p_1] < A[p_2]$) OR $p_2 = n + 1$, let $B[i] = A[p_1]$ and set $p_1 \leftarrow p_1 + 1$;
    (b) Otherwise, let $B[i] = A[p_2]$ and set $p_2 \leftarrow p_2 + 1$.
3. Return the array $B$ as the answer.
```

Example: Let us consider running this algorithm on the input $A = [1, 2, 7, 8, 3, 4, 5, 6]$. In the following, the underlined numbers denote the target of the pointers p_1 and p_2.

[2]The reason for calling this non-decreasing instead of increasing is that A may contain repeated entries that have the same value and thus it is not technically correct to call them increasing.

- At the beginning:

$$A[1:4] = [\underline{1}, 2, 7, 8] \qquad A[5:8] = [\underline{3}, 4, 5, 6] \qquad B = [-,-,-,-,-,-,-,-];$$

- After iteration $i = 1$ (since $A[1] = 1 < 3 = A[5]$):

$$A[1:4] = [1, \underline{2}, 7, 8] \qquad A[5:8] = [\underline{3}, 4, 5, 6] \qquad B = [1,-,-,-,-,-,-,-];$$

- After iteration $i = 2$ (since $A[2] = 2 < 3 = A[5]$):

$$A[1:4] = [1, 2, \underline{7}, 8] \qquad A[5:8] = [\underline{3}, 4, 5, 6] \qquad B = [1,2,-,-,-,-,-,-];$$

- After iteration $i = 3$ (since $A[3] = 7 > 3 = A[5]$):

$$A[1:4] = [1, 2, \underline{7}, 8] \qquad A[5:8] = [3, \underline{4}, 5, 6] \qquad B = [1,2,3,-,-,-,-,-];$$

- After iteration $i = 4$ (since $A[3] = 7 > 4 = A[6]$):

$$A[1:4] = [1, 2, \underline{7}, 8] \qquad A[5:8] = [3, 4, \underline{5}, 6] \qquad B = [1,2,3,4,-,-,-,-];$$

- After iteration $i = 5$ (since $A[3] = 7 > 5 = A[7]$):

$$A[1:4] = [1, 2, \underline{7}, 8] \qquad A[5:8] = [3, 4, 5, \underline{6}] \qquad B = [1,2,3,4,5,-,-,-];$$

- After iteration $i = 5$ (since $A[3] = 7 > 6 = A[8]$):

$$A[1:4] = [1, 2, \underline{7}, 8] \qquad A[5:8] = [3, 4, 5, 6], \underline{*} \qquad B = [1,2,3,4,5,6,-,-];$$

- After iteration $i = 6$ (since $p_2 = 9$):

$$A[1:4] = [1, 2, 7, \underline{8}] \qquad A[5:8] = [3, 4, 5, 6], \underline{*} \qquad B = [1,2,3,4,5,6,7,-];$$

- After iteration $i = 6$ (since $p_2 = 9$):

$$A[1:4] = [1, 2, 7, 8], \underline{*} \qquad A[5:8] = [3, 4, 5, 6], \underline{*} \qquad B = [1,2,3,4,5,6,7,8].$$

Proof of Correctness: We now prove the correctness of this algorithm. Even though the algorithm is not recursive, we again prove this by *induction* over the index i of the for-loop. Our induction hypothesis is that for every iteration $i = 1$ to n, $B[1:i]$ contains the smallest first i elements of A in the sorted array (after the iteration). By proving this hypothesis we will be done as it means that after iteration $i = n$, $B[1:n]$ contains the sorted version of the array A. We now prove the induction hypothesis.

For $i = 1$, the algorithm places $\min(A[1], A[\lfloor n/2 \rfloor + 1])$ in $B[1]$: since both $A[1:\lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1 : n]$ are sorted, $B[1]$ is the minimum of the array and hence the induction hypothesis holds. Suppose now that the induction hypothesis holds till iteration i and we prove it for iteration $i+1$ (the induction step).

Consider $A[p_1 : \lfloor n/2 \rfloor]$ and $A[p_2 : n]$ at the beginning of iteration $i+1$. Since by induction hypothesis $B[1:i]$ already contains the first smallest i integers in A, and since all elements in $A[1:p_1-1]$ and $A[\lfloor n/2 \rfloor : p_2 - 1]$ are already placed in the array B (by definition of how we update the pointers p_1 and p_2), we only need to ensure that in this iteration, $B[i+1]$ becomes the minimum of the elements in $A[p_1 : \lfloor n/2 \rfloor] \cup A[p_2 : n]$. This happens since both $A[p_1 : \lfloor n/2 \rfloor]$ and $A[p_2 : n]$ are sorted and thus minimum of the union is either $A[p_1]$ or $A[p_2]$: considering the algorithm checks for out of range values of p_1 and p_2 and then places $\min(A[p_1], A[p_2])$ in $B[i+1]$, we obtain that $B[1:i+1]$ also consists of the first $(i+1)$ elements of A in the sorted order. This proves the induction and the correctness of the merge algorithm.

Runtime Analysis: This is quite easy: the algorithm simply has a for-loop of n iterations and each iteration takes $O(1)$ time, so the runtime is $O(n)$.

The Merge Sort Algorithm

We now use the merge algorithm we already designed in the divide and conquer framework to obtain an algorithm for sorting. Recall that the merge algorithm could sort an array A in $O(n)$ time *assuming each half of the array were sorted already.* But what if we want to sort an arbitrary array? Can we first sort each of its half separately and then provide this new array to the merge algorithm to finish the problem? But of course we can do that! How? By using *recursion.*

The entire merge sort algorithm is simply as follows: if the size of the array is a small constant simply solve the problem by brute force. Otherwise, recursively merge sort the first half and the second half separately. Finally, run the merge algorithm on each part to obtain the final sorted array. Formally,

Merge Sort Algorithm: The input is an array A.

1. If the size of the array is $n = 0$ or 1 return the array as it is.
2. Recursively run merge sort on $A[1 : \lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1 : n]$.
3. Run the merge algorithm on A, copy the returned array of this algorithm back into A, and return A.

Example: Let us consider running this algorithm on the input $A = [5, 3, 7, 1, 8, 2, 4, 6]$.

- The splitting (recursive call) steps are as follows:

$[5, 3, 7, 1, 8, 2, 4, 6]$							
$[5, 3, 7, 1]$	$[8, 2, 4, 6]$						
$[5, 3]$	$[7, 1]$	$[8, 2]$	$[4, 6]$				
$[5]$	$[3]$	$[7]$	$[1]$	$[8]$	$[2]$	$[4]$	$[6]$

- Then these elements will start to get merged together in the following manner:

$[5]$	$[3]$	$[7]$	$[1]$	$[8]$	$[2]$	$[4]$	$[6]$
$[3, 5]$	$[1, 7]$	$[2, 8]$	$[4, 6]$				
$[1, 3, 5, 7]$	$[2, 4, 6, 8]$						
$[1, 2, 3, 4, 5, 6, 7, 8]$							

Proof of Correctness: As we proved the correctness of the merge algorithm already, this step is quite easy. We prove, by induction, that for every n, the merge sort algorithm sorts any given array A of n numbers. The base case for $n \in \{0, 1\}$ is true since any array of size 0 or 1 is already sorted and hence the correct answer is just to return the array as it is.

We now prove the induction step: assuming the hypothesis is true for all $n \leq i$, we prove this for $n = i + 1$. The algorithm firstly partitions the array into $A[1 : \lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1 : n]$ and recursively merge sort each one; by induction hypothesis, since size of each of these arrays is smaller than n, the recursive calls correctly sort each of them. At this point, the array A satisfies the requirement of the input for the merge algorithm and thus the output of the merge is the array A sorted entirely correctly. This implies that the output of the merge sort algorithm is correct, proving the induction step.

Runtime Analysis: We will write a *recurrence* for analyzing the runtime of this algorithm. Let $T(n)$ denote the worst-case time of running merge sort on any array of length n. We will prove that,

$$T(n) \leq T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + O(n).$$

(note that $T(n)$ is a recursive formula and technically we should also specify what is $T(n)$ for $n = \Theta(1)$ for the above formula to make sense; however, it is *always* the case that the running time of an algorithm on a constant size input is constant, namely, $T(\Theta(1)) = \Theta(1)$. Hence, *unlike* many other base cases, say, induction or recursive algorithms, here we can actually omit the recurrence base of a running time because it is always constant and so we can just implicitly assume that).

Let us first argue that $T(n)$ correctly captures the running time of the merge sort. The merge sort algorithm involves two recursive calls on arrays $A[1 : \lfloor n/2 \rfloor]$ (with size $\lfloor n/2 \rfloor$) and $A[\lfloor n/2 \rfloor + 1 : n]$ (with size $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$); moreover, it also involves running the merge algorithm that requires $O(n)$ time, and copying the resulting array to A which again can be done in $O(n)$ time. As such, the formula above for $T(n)$ is correct.

Having computed $T(n)$ recursively, we now want to turn it into a *closed-form* formula, namely, a non-recursive and "easy to state" function of n – this is crucial for better understanding the running time of the algorithm and potentially comparing it with other algorithms. Before we get to this however, we make an important remark about floors and ceilings in computing recurrences.

Ignoring Floors and Ceilings in Recurrences: When working with recurrences for analyzing runtime of algorithms, it is quite cumbersome to take into account the exact details of floors and ceilings. Fortunately, we can safely strip out the floors and ceilings in these formulas and for instance write that the worst-case running time of merge sort follows:

$$T(n) \leq T(n/2) + T(n/2) + O(n) = 2 \cdot T(n/2) + O(n).$$

Intuitively, this is because we are anyway upper bounding this function asymptotically and one can "charge" the extra numbers coming out of floors and ceilings to the non-recursive term in the formula which itself is stated asymptotically (here the $O(n)$-term). To be sure, this requires a formal proof but we will not do that in this course (the proof can be found in your textbook in Chapter 1.7[3]). Consequently, throughout this course, *we will always drop the floors and ceilings from recurrence relations for running times.*

Now back to analyzing runtime of merge sort. After ignoring floor and ceilings, our goal is to compute a closed-form formula for $T(n) \leq 2T(n/2) + O(n)$. This is a common recurrence and its correct answer is $T(n) = O(n \log n)$. This implies that merge sort runs in $O(n \log n)$ time (which is much faster than simple sorting algorithms such as selection sort or insertion sort). But how did we end up "solving" $T(n)$? This is the topic of the next lecture.

[3]`http://jeffe.cs.illinois.edu/teaching/algorithms/book/01-recursion.pdf`