CS 577: Introduction to Algorithms Fall 2020

How to Develop a Divide & Conquer Algorithm

Instructor: Dieter van Melkebeek TA: Ryan Moreno and Drew Morgan

This handout is meant to provide a basic structure for the problem solving and write-up steps for divide and conquer problems. Please note that these are not required patterns; they are merely suggestions. Any solution that is clear, correct, sufficiently efficient, and analyzed properly (correctness and complexity) will get full credit.

1 Suggested Write-up Format

1. Clearly state the input and output of your algorithm.
2. Describe your algorithm at a level that explains the intuition and key ideas. We should understand your basic approach without having to reverse-engineer your pseudocode.
3. Include pseudocode (with comments) if it adds clarity.
4. Prove the correctness of your algorithm.
5. Analyze the runtime of your algorithm.

2 Algorithm Description

I like to think of breaking the divide and conquer approach into three pieces:

1. **Split** your input into smaller-sized inputs that you can recursively call your algorithm on. As seen in class, this will often take the form of splitting your input in half (for example, splitting the input array in half or breaking the input integer into the left-most digits and the right-most digits). You need to split your input such that the results of these recursive calls are useful in calculating the output for the original problem.
2. **Stitch** together the results of your recursive calls. You need to explain how you would combine the results of your recursive calls into something that helps you answer the original question. For example, in MergeSort we used Merge to combine two sorted subarrays into one sorted array.
3. **Base Case(s)**. You need to consider when you should stop recursing. At what point is your input small enough that you cannot split it further? Explain what your algorithm will return in these cases.

These are the three key ideas that you need to communicate to the reader so that we understand how your algorithm works. To make sure that you cover all corner cases and don't gloss over any details, I suggest also including pseudocode (with comments) for your algorithm. However, make sure to describe your algorithm first so that we do not have to reverse-engineer your pseudocode. You are free to use algorithms from class, just state which algorithm you are using and its specifications.

3 Correctness

Because we are using divide-and-conquer, which is inherently recursive in nature, you will want to use proof by induction to prove your algorithm's correctness. You need to prove that your implementation correctly computes the specification.

1. **Specifications** (estimated 1-2 sentences)
   Make sure that you have clearly stated your algorithm's specifications, which are made up of the inputs to your algorithm (along with any restrictions to your inputs, such as restricting the elements of an array to positive integers) and the outputs of your algorithm (along with what you guarantee to be true about the outputs, such as guaranteeing an output array to be sorted).

2. **Base Case(s)** (estimated 1-2 sentences)
   At this point you are ready to use induction. Start with your inductive base case, which corresponds to your algorithm's base case. This will often look like something of the form "input $n = 1$" or "the input is of size $n = 1$". You must explain that, in the base case, what your algorithm returns matches the problem specification. Your base case might seem obvious (for example, "an array of size $n = 1$ is already sorted, so by returning the input as it is, we are correctly returning the sorted input array"). Even so, you need to include the base case for completeness.

3. **Inductive Hypothesis** (estimated 1 sentence)
   Next, state the inductive hypothesis, which is that the algorithm correctly returns the specified output for $n \leq k$ (note that this is strong induction, which you will most likely need). This will look something like "for an input array of size $n = k$, the algorithm correctly returns the sorted version of the input array." The inductive hypothesis corresponds to your algorithm's recursive calls because the inductive hypothesis claims the calls' correctness.

4. **Step** (estimated 1 paragraph)
   Finally, the bulk of your proof comes in the inductive step, which corresponds to the recursive case of your algorithm (everything that's not the base case). In the inductive step, you consider the case $n = k + 1$. You need to explain why the value your algorithm returns matches the specification's output. Your key tool is that you get to assume that the recursive calls are correct (by the inductive hypothesis).

Something of note: if you are having significant trouble proving the correctness of your algorithm, it may be that your algorithm is not correct. It is impossible to prove something that is false! If you think this might be the case, return to the algorithm development stage and try to see where your algorithm might be going wrong (specific example inputs might help).

4 Complexity Analysis

To analyze the runtime, you will want to use the recursion tree method we've been using in class in which you draw a tree with each node representing a call to your algorithm. The steps are as follows:

1. Model your algorithm's recursion through the shape of the tree. All non-leaf nodes will have a number of children corresponding to the number of recursive calls in your algorithm.

2. Calculate the work done at each node of the recursion tree by analyzing the runtime of your algorithm line-by-line, excluding the recursive calls (because they are accounted for by the shape of the recursion tree). You will also need to specify the input size at a given node. For example, if your algorithm makes recursive calls with input size half of the original input, then the input size for a node at level d is $\frac{n}{2^d}$. If this same algorithm takes $\mathcal{O}(n)$ time other than the recursive calls, then the amount of work done for a node at level d would be $c \cdot \frac{n}{2^d}$.

3. Determine the amount of work done over the entire recursion tree. There are some problems that require a unique approach for this step (such as in HW01 #1 when the work done at a given non-leaf node was proportional to the number of its children, so determining the total work done over the recursion tree involved the number of edges). However, this step will most often consist of determining the amount of work done at each level of your recursion tree and then summing up over all of the levels of your tree. The common approach is detailed below.

   (a) Determine the amount of work done at each level of your recursion tree. You will need to specify the number of nodes at a given level in terms of the depth d. For example, if your algorithm takes $\mathcal{O}(n)$ time other than the recursive calls and makes two recursive calls, each of input size half of the original input, then the amount of work done at level d of your recursion tree is $c \cdot \frac{n}{2^d} \cdot 2^d$.

   (b) Finally, sum over all of the levels of your recursion tree to calculate the total work done by your algorithm. This will often come in the form of a geometric series.

Reminder: Relevant Geometric Series Equations

$$\text{For } \alpha > 1: \quad \sum_{i=0}^{d} \alpha^i = \frac{\alpha^{d+1} - 1}{\alpha - 1} \quad \text{which is } \Theta(\alpha^d)$$

$$\text{For } \alpha = 1: \quad \sum_{i=0}^{d} \alpha^i = d + 1 \quad \text{which is } \Theta(d)$$

$$\text{For } 0 < \alpha < 1: \quad \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1 - \alpha} \quad \text{which is } \Theta(1)$$

5 Example Write-up

The solutions we hand out are intended to help all students, so they cover a number of possible avenues and are verbose. We don't expect students to do that. Here is a sample solution for HW01 #3a that doesn't have all of the extra material, follows the above outline, and would get full credit.

D&C Write-up Example: HW1 #3a

(Given a perfect binary tree with #s as the leaves, compute the minimum number of inversions possible when swapping at internal nodes of the tree.)

Problem Specifications

Input: A perfect binary tree T with height h
Output: The minimum number of inversions possible by swapping at internal nodes of T

Intuition

Think of the leaf values as an array A. This will simplify our algorithm, so our algorithm will take array A as input rather than tree T. We can break the inversions into categories: (1) within left half of A, (2) within right half, (3) between halves. Swaps on nodes in left subtree only affect (1). Swaps on nodes in right subtree only affect (2). A swap on the root node only affects (3). We'll minimize (1) and (2) through recursion. We will use COUNT-CROSS from class to decide if we should swap at the root node. We need sorted subarrays to perform COUNT-CROSS, so our alg will implement MERGESORT along the way and we will add to our outputs the sorted version of A. Our base case is an array with one element.

Pseudocode

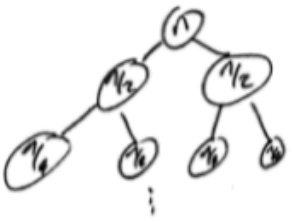
INPUT: A list of 2^h numbers (the leaves' numbers in order), A

OUTPUT: (i) the minimum # of inversions as in the problem statement
(ii) a sorted copy of A

```
Solve HW1#3 ( A ):
    If h = 0:
        return (0, A)
    Else:
        Let L, R ← left & right halves of A.
        Let c_L, L' ← Solve HW1#3 (L).
        Let c_R, R' ← Solve HW1#3 (R).
        Let c_1 ← Count-Cross (L', R')
        Let c_2 ← Count-Cross (R', L')
        Let A' ← MERGE (L', R')
        Return (c_L + c_R + min(c_1, c_2), A')
```

Solve the original problem by calling Solve HW1#3 & ignoring (ii).

Running Time Analysis



Depth: $O(\log n)$.
work per node: linear in input size
work per level: $O(n)$
Total work: $O(n \log n)$.

Note: n is the size of array A / the number of leaf nodes in T.

(COUNT-CROSS is from class.)
INPUT: two sorted arrays L, R
OUTPUT: # inversions in concat. LR

(MERGE is from class.)
INPUT: two sorted arrays L, R
OUTPUT: # sorted copy of their concatenation

Correctness: By induction on h

Base case (h=0): There are no swaps possible, so the answer is 0 inversions and A is already sorted.

Ind. Step (h>0): Each ~~inversion~~ potential inversion in A (pair of positions i, j with i<j) either has i,j pointing into L, i,j pointing into R, or ~~i,j pointing~~ i pointing into L and j pointing into R.
We minimize all three independently. The first by recursing on L, the second by recursing on R, and the latter by choosing whether to swap at the root. By the inductive hypothesis, c_L and c_R are the first two counts, and L' and R' are sorted copies of L and R. By correctness of COUNT-CROSS, $\min(c_1, c_2)$ is the minimum number of the third type. By correctness of MERGE, A' is a sorted copy of A.