Algorithms & Complexity: Lecture 11: Divide and Conquer

Sam Barrett

March 17, 2021

Divide and conquer is a very natural and useful algorithmic technique.

It breaks a problem down in to smaller subproblems which can be solved recursively and recombined into a final solution.

Abstractly they can be thought as following the steps:

1. Divide

   decompose the problem into smaller subproblems

2. Conquer

   Solve the smaller subproblems, usually done recursively

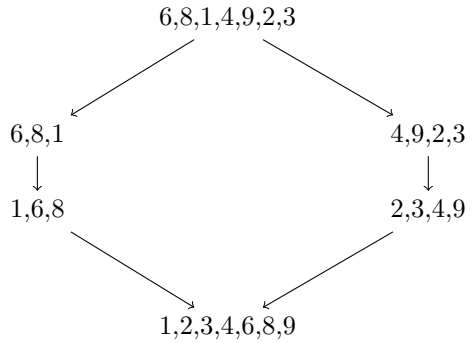
3. Combine

   Recombine the solutions to the subproblems into the final solution we return.

1 MergeSort algorithm

Problem 1 *(MergeSort) Given a list of n numbers, we want to sort them in ascending order.*

As specified earlier, we must first divide the problem into smaller problems. We will start by dividing in two (as it is the most trivial way)



Above you can see a very naive approach to merge sort. We split the initial problem in half, sort the halves and recombine.

Recombination in this example is done as follows:

1. Initialise pointers p_1, p_2 at the start of both sub-arrays
2. If $A_1[p_1] < A_2[p_2]$ then append $A_1[p_1]$ to the return array and move the p_1 to the right, otherwise add $A_2[p_2]$ and move p_2 to the right
3. repeat until both sub-arrays are empty (pointers cannot move to the right)

A more formal definition for merge sort is:

Algorithm 1: MergeSort(A)

```
1  Divide the given list A on n numbers into two lists B and C of equal
     size
2  Let B' = MERGESORT(B) and B' = {b_1 < b_2 < ... < b_{n/2}}
3  Let C' = MERGESORT(C) and C' = {c_1 < c_2 < ... < c_{n/2}}
4  Initialise i = 1, j = 1, k = 1
5  Initialise D to be an empty array
6  while i ≠ n/2 ∩ j ≠ n/2 do
7  |  if b_i < c_j then
8  |  |  Set d_k = b_i
9  |  |  k, i++
10 |  else
11 |  |  Set d_k = c_j
12 |  |  k, j++
13 |  end
14 end
15 If one of the lists becomes empty, append the remainder of the other
     list to D
16 return D
```

Clearly this algorithm does not simply divide the array once, it will divide recursively until the arrays are a single element in length, relying on the recombination process to actually sort the array.

The time needed to recombine B' and C' to obtain D is $O(n)$.

Therefore, the recurrence is $T(n) \leq 2 \cdot T(n/2) + O(n)$.

We can show (but won't) that $T(n) = O(n \log n)$ satisfied this recurrence.

2 Solving recurrences

At the end of the merge sort section we saw that we formulated a recurrence to describe the complexity of our divide and conquer algorithms.

There are two main methods for solving these recurrences.

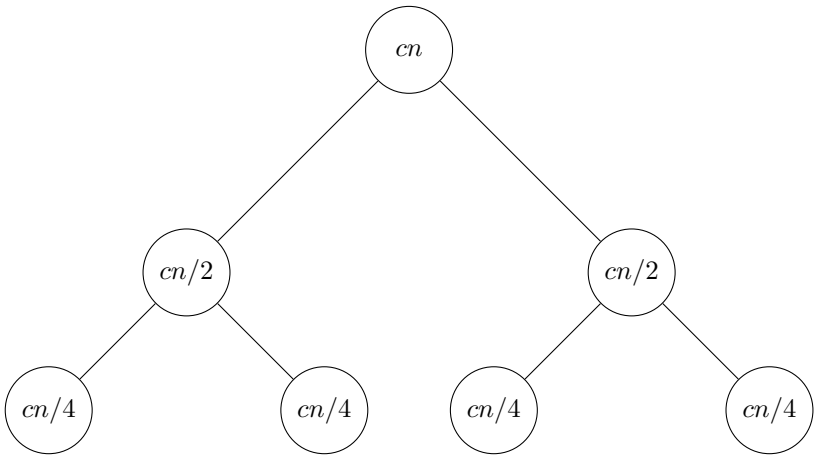
2.1 Method 1: *Unrolling* the recurrence

In this method we *open up* the recurrence step-by-step. It does not require any knowledge of the final result of the process.

We start by writing our recurrence for some constant value c:

$$T(n) \leq 2 \cdot T(n/2) + cn, \forall n > 2$$

Base Case: $T(2) \leq c$



Level 0: cn

Level 1: $cn/2 + cn/2 = cn$ total

Level 2: $4(cn/4) = cn$ total

Note that at each step/level we have exactly cn running time. This is our pattern, and would continue until we reach a level in which each node is of size 2 in which case we can use our base case.

We must now sum over all levels of the recursion. We will have $\log_2 n$ levels as our base case is that $n = 2$. It will take $\log_2 n$ splits until we reach sub-arrays of size 2.

We can therefore say that as we have $\log_2 n$ levels and each level has cn running time, that our overall running time is $cn \cdot \log_2 n$.

We can formalise this process as follows:

1. Write the recurrence explicitly for some constant c

   $T(n) \leq 2 \cdot T(n/2) + cn$

2. Construct our base case

   $T(2) \leq c$

3. Analyse the first few levels

   Level 0 takes cn time, and delegates two recursive calls of size $n/2$

   Level 1 takes $c(n/2) + c(n/2)$ time and delegates four recursive calls of size $n/4$

4. Identify a pattern

   The number of subproblems doubles at each level

   The size of each subproblem halves at each level

   Level j has 2^j subproblems each having size $n/2^j$

   So level j takes a total of $2^j \cdot (c \cdot (n/2^j)) = cn$ time and delegates some tasks to the next level

5. Sum up over all levels of the recursion

   There are $O(\log_2 n)$ levels of the recursion

   Each level requires a running time of cn

   $T(n) = cn \cdot \log_2(n) = O(n \cdot log_2 n)$

2.2 Verifying by substitution in the recurrence

This works well if you already have a *guess* for the running time. You can then check by induction whether your guess satisfies the recurrence.

- **Base Case:** $n = 2$

  $T(2) \leq c \leq cn \log n$

- **Inductive Hypothesis**

  For each $2 \leq m \leq n$ we have $T(m) \leq cm \cdot \log m$

- **Inductive step**

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn && \text{(by the recurrence)} \\ &\leq 2c \cdot (n/2) \cdot \log(n/2) + cn && \text{(by inductive hypothesis since } n/2 < n) \\ &= cn(\log n - 1) + cn && \text{(since } \log(n/2) = \log n - 1) \\ &= cn \cdot \log n \end{aligned}$$

NOTE: we have here verified that $T(n) = cn \cdot \log n$ is <u>one</u> possible solution for this recurrence, unlike the unrolling method, this method may lead to verifying that a much higher running time is correct.