Ch2: Divide and Conquer (P1)

Kevin Moy

Abstract

One of the most powerful strategies ever invented to solve a multitude of problems is the divide-and-conquer strategy. In its most basic form, divide and conquer involves breaking a problem into subproblems that are simpler versions of the same original problem. Then we recursively solving these subproblems, and finally combine their answers to produce a final solution. We'll analyze this in detail in these notes.

1 Multiplication

The product of 2 complex numbers

$$(a + bi)(c + di) = acbd + (bc + ad)i$$

seems to involve four real-number multiplications: ac, bd, bc, ad. However, Gauss found that we just need three: $ac, bd, (a + b)(c + d)$:

$$(a + bi)(c + di) = acbd + ((a + b)(c + d) - ac - bd)i$$

Damn, we can do one less multiplication with this algorithm. But when we apply such algorithms recursively, **every operation counts**.

Let's see how, first with standard multiplication of n-bit x and y once again. Let's also assume WLOG that n is a power of 2. First, we halve x and y into two $n/2$ -bit chunks x_L and x_R (and y_L and y_R):

$$x = 2^{n/2}x_L + x_R y = 2^{n/2}y_L + y_R$$

Now the product of x and y is:

$$xy = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

We know additions take $O(n)$ time, as do bit shifts. The key question is: how long do the four **recursive** n/2-bit multiplications $(x_L y_L, x_L y_R, x_R y_L, x_R y_R)$ take?

So if we express the runtime of multiplying 2 n-bit numbers as $T(n)$, we can develop this **recurrence relation**:

$$T(n) = 4T(n/2) + O(n)$$

Translated to english, this just says "the runtime to multiply two n-bit numbers is equal to to the total runtime of FOUR multiplications of n/2-bit numbers plus some extra linear time to put them all together into a final solution".

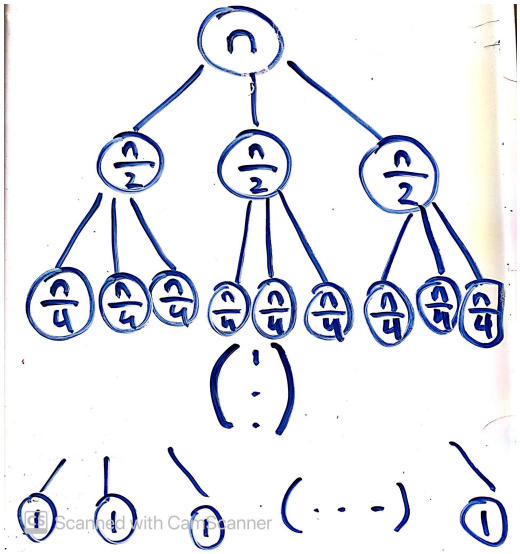
Unfortunately, this recurrence comes out to $O(n^2)$- making our fancy tricks no better. Can we improve this? Well, remember Gauss's method reduced $x_L y_R + x_R y_L$ to $(x_L + x_R)(y_L + y_R) - x_L x_R - y_L y_R$- only requiring 1 new multiplications instead of 2. This, then, means we only need 3 multiplications, resulting in recurrence relation

$$T(n) = 3T(n/2) + O(n)$$

which now has a runtime of around $O(n^{1.59})$. Improved!

The reason this improved so much is because **at each level of recursion** we went from doing 4 to 3 recursive multiplications. Whenever we can save the number of recursive calls, we'll improve our runtime drastically.

We can model an algorithm's recursive calls as a tree, each node representing a subproblem (recursive call to `multiply`) of size represented by node value:



Let's try to understand this tree. Each level of recursion **halves** the size of subproblems. At level $\log_2(n)$, this size becomes 1, where recursion stops. So the height of this tree is $\log_2 n$. The **branching factor** (number of children per node), or the number of smaller recursive problems produced with each call, is 3. Thus, at depth k, there are 3^k subproblems, with size $n/2^k$.

So the total time spent at recursion depth k is summing a row of tree nodes, or $3^k * O(n/2^k) = (3/2)^k * O(n)$. Our total runtime is the sum of all these nodes.

Thus we want:

$$\sum_{k=0}^{\log_2 n} (3/2)^k * O(n) = n \sum_{k=0}^{\log_2 n} (3/2)^k$$

which, to a constant factor, is the same as the last term in the series, $O(n^{\log_2 3})$. Thus Gauss's implementation has $O(n^{1.59})$ runtime- a solid improvement from $O(n^2)$.

Thus, even small changes to the number of recursive subproblems (branching factor) can have a big effect on runtime.

The **Fast Fourier Transform** (FFT) algorithm is even faster for multiplying. However, that's for another note and another day.

2 Recurrence Relations

Generally, we can systematize divide and conquer algorithms: solve a size n problem by recursively solving a subproblems of size n/b, and combining these solved subproblems in $O(n^d)$ time. Let us form the **master theorem**:

$$T(n) = aT(\lceil n/b \rceil) + O(n^d) T(n) = \begin{cases} O(n^d) & d > \log_b a \\ O(n^d \log n) & d = \log_b a \\ O(n^{\log_b a}) & d < \log_b a \end{cases}$$

given positive a, b, d.

2.1 Proof: Master Theorem

Let's assume WLOG n is a power of b.

The size of the subproblems is divided by b with each recursive call. Thus, the base case, or subproblems of size 1, are reached at tree depth $\log_b n$, or that recursive depth. Branching factor is a, so at depth k we have a^k subproblems, each of size n/b^k. So total runtime:

$$\sum_{k=0}^{\log_b n} a^k * O(\frac{n}{b^k})^d = \sum_{k=0}^{\log_b n} O(n^d) * (\frac{a}{b^d})^k$$

Another geometric series with ratio a/b^d. Then it's just casework to find the sum:

- $a/b^d < 1$: Decreasing series, first term dominates: $O(n^d)$
- $a/b^d = 1$: $\log_b n$ terms of $O(n^d)$ added: $O(n^d \log n)$.
- $a/b^d > 1$: Increasing series, last term dominates: $O(n^{\log_b a})$

3 Mergesort

Divide and conquer applies to sorting lists: halve each list, recursively sort, then merge the sorted sublists. This is precisely the `mergesort`algorithm, pseudocode below:

```
function mergesort(a[1...n]):
    if n > 1:
        return merge(mergesort(a[1...⌊n/2⌋]),
                     mergesort(a[⌊n/2⌋+1 ... n]))
   else:
        return a //single element.
```

What is `merge(x,y)`? Given two **sorted** arrays x and y, we want to use merge to output a single sorted array z with length $|x|+|y|$. We simply add the smallest element of x and y to z, one at a time. `merge`has runtime of $O(|x| + |y|)$, since we process both arrays completely- this is linear. Thus recurrence relation of `mergesort` is:

$$T(n) = 2T(n/2) + O(n)$$

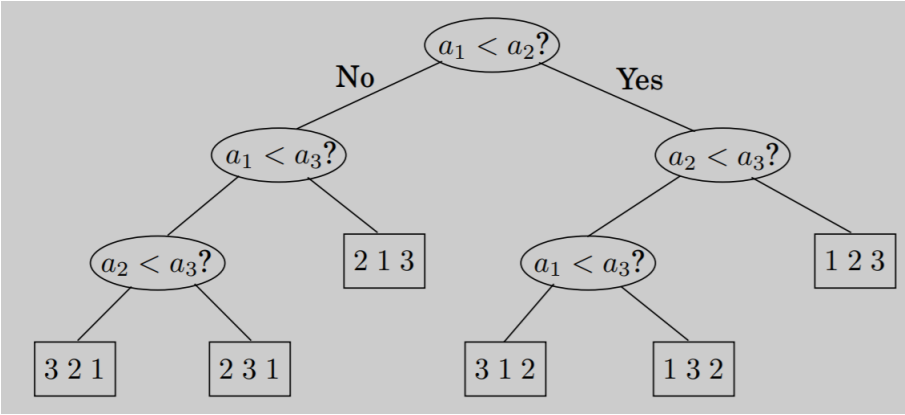
Hey, that looks pretty familiar. Master theorem gives **overall runtime of mergesort as**$O(n \log n)$.

Interestingly, merging doesn't happen until we've recursively broken down the array to **singletons** (single-element subarrays). These singletons are merged into 2-elements, then 4, etc. We can actually make `mergesort` *iterative* via priority queue: continuously merge "active arrays" in each queue and place the merged results at the end of the queue. Code below:

```
function mergesort-iterative(a[1...n]):
    Q = []
    for i=1 to n:
            inject(Q, a[i])
    while |Q| > 1:
            inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```

4 Sidenote: Decision Trees in Sorting

Most sorting algorithms can be represented as decision trees. Each node is represented by a two-element comparison. Starting at the root, if the comparison is false, we traverse left: if correct, right. Leaf nodes represent a "final decision": returned sorted list. Example below:



The height of this tree (the maximal number of edges from root to leaf) is exactly the worst-case time complexity of the sorting algorithm the tree represents. In the example above, height is 3.

Such tree structures allow us to argue that **mergesort is optimal**: we need *at least* $n \log n$ comparisons to sort n elements. How does this tree prove this? Well, each tree needs at least one leaf for every permutation of n elements- each represent a sorting order. There are $n!$ permutations, so we need *at least* $n!$ leaves. A **binary** tree of depth d (same as height) has at most 2^d leaves. So we establish inequality $n! \leq 2^d$, or $d \geq \log(n!)$. We also know $log(n!) \geq c * n \log n$ for constant c. We've lower-bounded our worst-case complexity on $n \log n$, so we know that `mergesort`, whose runtime is $O(n \log n)$, is optimal.

This is beautiful but only works to comparison-based sorting algorithms. Are there sorting algorithms that work some other magic to run in linear time? For certain exceptions, the answer is yes.