CS 170: Efficient Algorithms and Intractable Problems

Note 2 - Divide and Conquer

Vishnu Iyer January 30, 2019

Throughout this class you will see several examples of algorithmic design techniques. Among the most ubiquitous and elegant of these techniques is **divide-and-conquer.** From a high level, every divide-and-conquer algorithm functions in the same way: break a complex problem down into simpler parts, solve those recursively, and then combine the solutions to those problems to obtain a solution to the original problem.

Obviously, divide-and-conquer is only useful if the cost of solving the whole problem otherwise outweighs the process of breaking it down and pieceing it together again. This suggests the use of the Master Theorem in runtime analysis. As we will see, some methods of dividing and conquering are more efficient than others. While the underlying principles are the same, each problem will require its own bit of cleverness and customization.

1 Fixed Points

Given a sorted length n array A of distinct integers, a *fixed point* is an integer k such that $A[k] = k$. We want to find the index of a fixed point if one exists, and otherwise return -1 . We can do this naively in linear time by iterating through the array and checking each element. However, there is a more efficient method employing the basic structure of binary search.

In particular, if we have that $A[k] > k$, then the fixed point (if it exists) must be at an index that is less than k. For the array is strictly increasing, and if $A[k] > k$, then $A[j] > j$ for any $j > k$. On the other hand, if $A[k] < k$, the fixed point must be at an index greater than k. With these splits, we can halve the search space on every iteration. The complete algorithm is given on the next page.

Is this algorithm correct? Well, we just argued that if $A[k] > k$ then the fixed point cannot be at index k or greater and if $A[k] < k$ then the fixed point cannot be at index k or less. Thus by splitting the array in half every time and checking for a fixed point, we are eliminating all possibilities until perhaps we reach such a point. If we do not reach it by the time we have eliminated all but one element and it turns out that that element is not a fixed point, then we see that the array has no fixed points.

Now what's the runtime? We break the problem of searching a length n array down to searching a length $n/2$ array with some constant time overheads. The recurrence is thus

$$T(n) = T(n/2) + O(1)$$

and the master theorem gives us that the runtime is $O(\log n)$.

Now take a minute and notice the striking similarity to the binary search algorithm. Only the

Algorithm 1 Fixed Point

```
1: procedure FixedPoint(A, n)
2:     ℓ = 1
3:     h = n
4:     m = (ℓ+h)/2
5:     while h > ℓ do
6:         if A[m] = m then return m
7:         else if A[m] > m then
8:             h = m
9:             m = (ℓ + h)/2
10:        else
11:            h = ℓ
12:            m = (ℓ + h)/2
       return −1
```

manner in which we split is different. This brings to attention an important technique in the design of approximation algorithms - modifying existing algorithms. We will talk more about this later.

2 Fast Integer Multiplication

In many of our analyses, we take basic arithmetic tasks to be constant-time operations. However, as you might imagine, this is not necessarily the case. Intuitively, it should take longer to multiply larger numbers. For example, multiplying a double digit number in the naive way requires around four times as many operations on single digit numbers. Concretely, let n be the number of digits in our 2 numbers. You can verify that the grade-school multiplication method of multiplying digit by digit takes time $O(n^2)$.

Now let's try a fancier method. Call our 2 numbers X and Y. Let's assume without loss of generality that we are working in base 2. We can write the numbers as follows:

$$X = X_L \cdot 2^{n/2} + X_R$$

$$Y = Y_L \cdot 2^{n/2} + Y_R$$

Intuitively, X_L and Y_L are the "left halves" of the numbers and X_R and Y_R are the "right halves." For example, if $X = 1101$, then $X_L = 11$ and $X_R = 01$. Can we break down the multiplication of the n bit numbers down to the multiplication of $n/2$ bit numbers? Well, notice that

$$XY = X_L Y_L \cdot 2^n + (X_L Y_R + X_R Y_L) \cdot 2^{n/2} + X_R Y_R$$

In other words, to multiply the two numbers, we need to perform 4 smaller operations. Since the combinations are multiplied by powers of 2, in binary this just means we shift over the digits. Therefore after we perform the smaller multiplications, combining the answer takes time linear in the number of digits. Therefore, our recurrence is

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

A simple application of the master theorem gives us that the runtime is, again $O(n^2)$. All that work for no significant improvement! Is $O(n^2)$ the best we can do? Well, for a while, we thought that this was the limit. However, we just have to be a little clever in our calculations. Notice that we don't care about $X_L Y_R$ or $X_R Y_L$ alone. We care about their sum. With this in mind, observe the following:

$$X_L Y_R + X_R Y_L = (X_L + X_R)(Y_L + Y_R) - X_L Y_L - X_R Y_R$$

Therefore, if we save the results of $X_L Y_L$ and $X_R Y_R$, we just need to compute one more multiplication of size n. The incurred addition overhead is still linear in n, so the recurrence relation becomes

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

and we have a multiplication algorithm that runs in time $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.6})$.

3 Finding a Majority Element

Suppose we have a list A of n (not necessarily comparable) objects. As one might imagine, it would be useful to find a *majority* element. That is, an element which occurs at more than half of the indices. Keep in mind that we can't simply sort the list since the elements aren't comparable, but assume we can check for equality in constant time. Naively, we can count the number of occurrences of each element, which will take quadratic time. However, can we do better than this?

A key observation here is that if we split the array in two equal parts, an element cannot be a majority element if it is not the majority element in either half. Immediately, this suggests a divide and conquer algorithm where we split the array in two, find the majority elements in each half (if they exist), and then count the number occurrences of each of those elements in the original array.

Our base case is when the array has 1 element, which is clearly the majority element for the array. In our implementation below, we will use nil to indicate the lack of a majority element and assume the array contains no nil entries.

Since searching the array for the majority in the left and majority in the right takes linear time, our recurrence is

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

and thus our runtime is $\Theta(n \log n)$. This is a classic example of a divide-and-conquer algorithm where we divide a problem into smaller *subproblems* and use information from the solutions to the subproblems to piece together the original solution. We will discuss this more in the next section, but several divide-and-conquer algorithms follow this paradigm.

As it turns out, we can do even better than this. In the Karatsuba algorithm, we took a basic idea and modified it a little to get a better runtime. Here, we will discard our previous algorithm and consider something completely different. Suppose we paired up the first and the second element of the array, and then the third and fourth, and so on.

We now have $n/2$ pairs. Considering any pair, we discard the pair if the two elements are different and keep one of the elements if they are the same. Then we combine the remaining elements

Algorithm 2 Majority Element

```
1: procedure MAJORITY(A, n)
2:     if n = 1 then return A[1]
3:     left = A[1, ..., n/2]
4:     right = A[n/2 + 1, ..., n]
5:     x = MAJORITY(left, n/2)
6:     y = MAJORITY(left, n/2)
7:     a = 0
8:     b = 0
9:     for z in A do
10:        if z = x then
11:            a = a + 1
12:        else if z = y then
13:            b = b + 1
14:    if a > n/2 then return x
15:    else if b > n/2 then return y
16:    else
17:        return nil
```

into an array It is clear that the new array created by this process has at most $n/2$ elements, but we can show something even stronger.

Proposition 3.1. The original array contains a majority element only if the array created by the filtering process does.

Proof. Call the original array A and the array created by our filtering process B. Suppose that A has a majority element and call it x. Suppose further that in our pairing, r pairs of equal elements other other than x have been grouped. Let y be the number of pairs of x. Amongst the remaining $n - 2r$ elements, we know that fewer than $n/2 - 2r$ are not equal to x.

Thus the the number of x elements that get filtered out is less than $n/2 - 2r$, and so the number of x terms that remain is greater than $n/2 - n/2 + 2r = 2r$. Since greater than $2r$ copies of x remain to be paired, there are at least $r + 1$ pairs of x elements in the filtered array. This gives us that x is a majority element in the filtered array and we are done. □

Therefore, our algorithm will pair up elements in this order until we reach an array where either every element is the same or the array is empty, where we deduce that the original array had a majority element or no majority element, respectively.
At each step of the recurrence, we still perform linear work in checking that every element is equal and filtering the array. However, the two subproblems of size $n/2$ now only become one. Our recurrence relation is thus

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

and our runtime is $O(n)$.

Algorithm 3 Majority Element 2

```
1: procedure MAJORITY(A, n)
2:     if A is empty then return nil
3:     else if all the elements of A are identical then return A[1]
4:     instantiate B to be a dynamic array
5:     ℓ = 0
6:     for i = 1 to n do
7:         if A[2i − 1] = A[2i] then
8:             ℓ = ℓ + 1
9:             append A[2i] to B
10:    return MAJORITY(B, ℓ)
```

4 A Second Look

Let us take a moment and look over our examples. Like we mentioned in the introduction, we used the master theorem in the analysis of essentially every one of the above algorithms. This stands to reason as we break a problem of size n into a problems of size n/b and use time $f(n)$ to put them together. While not every divide-and-conquer algorithm can be analyzed this way, the master theorem will allow us to quickly analyze almost every such example we see.

Now think about the different techniques we demonstrated. The first algorithm we saw was only a slight modification of binary search, an algorithm that was already in our repository. As we saw later, not every problem can be solved this way, but it would do us much good to determine whether the problem at hand strongly resembles other problems we have encountered.

From that point, the solution usually follows from a quick adjustment of the original algorithm. If anything, this emphasizes the power of working through example problems for a unit such as this - the more examples you have in mind, the more similarities you can draw from a new problem.

Next, when we considered the problem of integer multiplication, we took a basic divide-and-conquer idea and, when it yielded no significant improvement, changed it a little bit to make it more efficient. Many problems can be tackled in this way.

Another such example is Strassen's matrix multiplication algorithm. When breaking the problem down doesn't seem to get anywhere, we try to exploit some underlying algebraic relations to make our lives easier. In the end, our algorithm remained the same but one small manipulation allowed us to extract all the necessary information more efficiently.

Our majority element algorithm similarly relied on the classic divide-and-conquer principle of looking at the problem when divided into halves. However, the lesson we can extract from this example is that when we sought a divide-and-conquer algorithm to do even better, we changed our design principles entirely. This shows that divide-and-conquer algorithms for the same problem can look very different and that there is not always one way to approach a problem. Thankfully, the master theorem allows us to easily determine which methods perform better.

A final note about algorithm design: these algorithms, particularly Karatsuba's algorithm and the linear-time majority algorithm rely on seemingly arbitrary ideas. Where does one get the inspiration for such things? It may seem, like a slew of other algorithms we will encounter, that the main principles simply appear from thin air! Well, be rest assured that a lot of these algorithms took years to develop and were the result of trial and error.

While you certainly would never be expected to come up with such ideas instantly, analyzing the main ideas will solidify the fundamentals. The moral of the story is exposure. The more you have to go off of, the more ideas you will be able to come up with.