CS 3510 Algorithms Last updated: September 2, 2023

Lecture 3: Multiplication and Median Finding

Lecturer: Sahil Singla *Scribe(s): Joseph Gulian*

Disclaimer: These notes have not been carefully verified and might contain mistakes.

As we discussed last time, the idea of this unit is to take a problem and split it into sub-problems of the same problem. If the sub-problem is the same as the larger problem, then you can use recursion. Then you can combine solutions to the sub-problems to get the solution to the overall problem.

We've also discussed several methods of showing the runtime given the recurrence relation. You can use the tree method which is nice. The first step of a proof by induction is to give the induction hypothesis; following that, you give the base step and the induction step. The last method is the Master's theorem which takes recurrences of the form $T(n) = aT(n/b) + \mathcal{O}(n^d)$ and you use the following function to find the overall runtime.

$$T(n) = \begin{cases} \mathcal{O}(n^d) & d > \log_b a \\ \mathcal{O}(n^d \log n) & d = \log_b a \\ \mathcal{O}(n^{\log_b a}) & d < \log_b a \end{cases} \qquad \text{(Master Theorem)}$$

1 Multiplication

Given two n bit numbers $A = a_1, a_2, \ldots, a_n$ and $B = b_1, b_2 \ldots, b_n$. The problem is to find $A \cdot B$.

The first question to ask with this problem is what operations can we use, and what are their runtimes. Often we think about operations like multiplication and addition as constant time because in our computers, the size of the numbers are limited (32 bit or 64 bit). The size of the numbers never changes though, so it's constant. However in this algorithm, we can scale the size of the numbers (think hundreds or thousands of bits), and we are worried about the runtime of multiplication. Considering these, addition is clearly $\Theta(n)$ because you need to iterate over each bit and calculate the sum and carry. For a naive multiplication (standard multiplication), it will take $\Theta(n^2)$ because you take each bit in A and mask/shift the bits in B, and then sum those. How can we make multiplication faster?

Obviously this lecture is about divide and conquer, so let's try to use divide and conquer. A simple approach is to split the problem into the first and second half and then try on those halves. As notation, we'll split A into $A_{i,j} = a_i, a_{i+1}, \ldots, a_j$, and the same for B. Take $A_L = A_{1,n/2}$, $A_R = A_{n/2+1,n}$, $B_L = B_{1,n/2}$, and $B_R = B_{n/2+1,n}$. Now $A \cdot B = A_L \cdot B_L \cdot 2^n + A_R \cdot B_R + (A_L \cdot B_R + A_R \cdot B_L) \cdot 2^{n/2}$. [1] Now we have 4 subproblems of half size, four additions, and 2 shifts; since shifts and additions take linear time, our recurrence

[1] Multiplying by $2^{n/2}$ in base 2 is a "shift" operation; Like in base 10, when you have 123, and you multiply by some power of 10, you can just write n zeros after, in base 2, you can just shift the digits you have over n places.

relation is $T(n) = 4T(n/2) + \mathcal{O}(n)$. If you plug this recursion into the Master's Theorem, the solution will be $T(n) = \Theta(n^2)$. Then we haven't made any progress.

Looking at the recursive relation, we likely won't get better than linear time at each level, so we will try to lower a or raise b. This is challenging, but as it turns out, we can lower a to 3, and improve our runtime which would make our recurrence $T(n) = 3T(n/2) + \Theta(n)$. This would make our overall runtime $\Theta(n^{\log_2 3})$ which is $o(n^2)$.

To only make three multiplications we'll do the following multiplications $A_R \cdot B_R$, $A_L \cdot B_L$ and $(A_L + A_R) \cdot (B_L + B_R) = A_L \cdot B_L + A_R \cdot B_R + A_L \cdot B_R + A_R \cdot B_L$. We actually already have the first two terms of this product, $A_L \cdot B_L$, and $A_R \cdot B_R$, meaning we can subtract those from the product of $(A_L + A_R) \cdot (B_L + B_R)$, to get $A_L \cdot B_R + A_R \cdot B_L$. Returning to the above multiplication, $A \cdot B = A_L \cdot B_L \cdot 2^n + A_R \cdot B_R + (A_L \cdot B_R + A_R \cdot B_L) \cdot 2^{n/2}$, we have all the terms. Rewriting this in terms of the multiplications we're doing, we get $A \cdot B = A_L \cdot B_L \cdot 2^n + A_R \cdot B_R + ((A_L + A_R) \cdot (B_L + B_R) - A_L \cdot B_L - A_R \cdot B_R) \cdot 2^{n/2}$. [2] Notice here that the only multiplcations we're doing are on the $n/2$ bit numbers $A_L \cdot B_L$, $A_R \cdot B_R$, and $(A_L + A_R) \cdot (B_L + B_R)$, so our recurrence is $T(n) = 3T(n/2) + \Theta(n) = \Theta(n^{\log_2 3})$.

2 Median Finding

The second problem we'll be discussing today is the median problem where you're given n numbers $A = a[0], \ldots, a[n-1]$, you're given a $k \in 1, \ldots, n$. The goal is to find the kth smallest number. This problem can easily be solved by sorting the numbers and giving the kth entry in the sorted array. However it turns out there are $\Theta(n)$ algorithms.

The first step towards $\Theta(n)$ is asking if there is a $\Theta(kn)$ time algorithm. To do this, we'll iterate over the array at each step and remove the smallest number, then after $k-1$ iterations the smallest number will be the kth smallest number in the original array, so we can return that. However in the worst case for k, this algorithm can take $\Theta(n^2)$, which is worse than merge sort.

Let us introduce a concept called the pivot which, while it may not be the exact median, it is close to somewhere in the middle of the sorted array. Let's say there must be at least $n/10$ elements that are smaller and at least $n/10$ elements that are larger than the array. Once you've found your pivot, you could all the numbers that are less than the pivot, or $n_p = \#$ elems $< p$ If $k < n_p$, then the numbers that are larger than the pivot must be useless. However if $k > n_p$, then we throw out the numbers that are less than the pivot, and on the remaining elements, we find the $k - n_p$th smallest element.

For instance if we have the array $5, 4, 2, 6, 0, 3, 1$ and we're looking for the fourth largest element. If we take the first element, we'll see that 5 numbers are less than 5. Since $5 > 4$, throw out numbers that are greater than or equal to 5. We'll get $4, 2, 0, 3, 1$; now if we randomly take the pivot 2, we'll go through the array and find that 2 numbers are less than 2. Since this is less than k, we'll drop those numbers to get $4, 3$. At this point, you could proceed with the rest.

Given that we have good pivots, we'll be reducing the problem size to $9/10n$ at each step because we'll be able to get rid of $n/10$ of the array. Then our recurrence will be

[2] There are better algorithms which can get $\Theta(n \log n)$; this is actually a very recent discovery by Harvey and Hoeven in 2021. It's worth noting that sometimes runtime isn't everything and sometimes the constants aren't great. Of course no matter the constants, this is a very interesting asymptotic result.

$T(n) = T(\frac{9}{10}n) + \Theta(n)$, making our overall runtime $\Theta(n)$.

By the end, our recurrence will be $T(n) = T(\frac{7}{10}n) + T(\frac{1}{5}n) + \Theta(n)$. If you solve this recurrence by induction your overall runtime will be $\Theta(n)$. As you see, this runtime requires that we be stricter though in that we make the pivot have at least 30% of the elements on either size.

Think about arranging your n elements into a 5 by $n/5$ grid. Each of the columns you can sort in constant time, and since there are order $n/5$ columns, it will take $\Theta(n)$ time. Now we'll be able to find the pivot of the middle row by using the middle row as a subproblem. All the elements on the top left or the bottom right will be smaller or larger than the median

For instance below imagine a grid layed out by it's ordering. We will actually get something like this by going through the following steps. Sort each column in constant time (there's only $5 \log 5$ comparisons that maximally need to be done). Find the medium of the middle row, and call that the "median of medians". Now compare the median of median with each other element in the middle row; if the median is larger, put it on the left, and if it is greater put it in the right. Now, your middle row isn't exactly sorted, but there are an equal number of elements on the left and right of the median of medians.

$a[0]$	$a[5]$	$a[10]$	$a[15]$	$a[20]$	$a[25]$	$a[30]$
$a[1]$	$a[6]$	$a[11]$	$a[16]$	$a[21]$	$a[26]$	$a[31]$
$a[2]$	$a[7]$	$a[12]$	$a[17]$	$a[22]$	$a[27]$	$a[32]$
$a[3]$	$a[8]$	$a[13]$	$a[18]$	$a[23]$	$a[28]$	$a[33]$
$a[4]$	$a[9]$	$a[14]$	$a[19]$	$a[24]$	$a[29]$	$a[34]$

Figure 1: Visualization of Median of Medians

Even more importantly though, we know that the elements in the blue must be smaller, and the elements in the red must be larger (the element in both is the median of medians). Say an element is greater than the median of medians, but it is in the blue. We know that that element must be less than or equal to the element in the same column but the middle row. We also now that the element from the middle row must be less than or equal to the median of medians. This creates a contradiction, meaning the elements in the blue box must less than or equal to the pivot. You could do something similar for the red box.