CS 3510 Algorithms Last updated: September 2, 2023

Lecture 4: Median Finding Continued and Fast Matrix Multiplication

Lecturer: Sahil Singla *Scribe(s): Joseph Gulian, Allen Chang*

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

1 Median Finding Continued

Recall the median of medians algorithm from last class, as well as the idea of using a pivot to "guess" a median. But how do you find a "good" pivot? Recall the following median of medians algorithm:

1. Arrange list in a matrix of size $5 \times n/5$ (that is, $n/5$ columns of height 5). This takes $\Theta(n)$ time.
2. Sort each column of size 5. Since each column takes a constant number of comparisons, comparing $n/5$ of them would take $\Theta(n)$ time.
3. Compute the median of $n/5$ elements in the "center" row.

Now, we're making the claim that the returned element is a good pivot, and by this we mean, we have at least 30% of the array on both sides.

Proof. Consider the median of the center row; call this M. There must be at least $\frac{n}{10}$ smaller elements in the center row (since there are $\frac{n}{5}$ elements in the center row, and half must be less if it's the median). Furthermore, each of these columns have three elements that are smaller than it. Thus, there are $\frac{3n}{10}$ elements smaller than the median (and another $\frac{3n}{10}$ elements greater than the median).

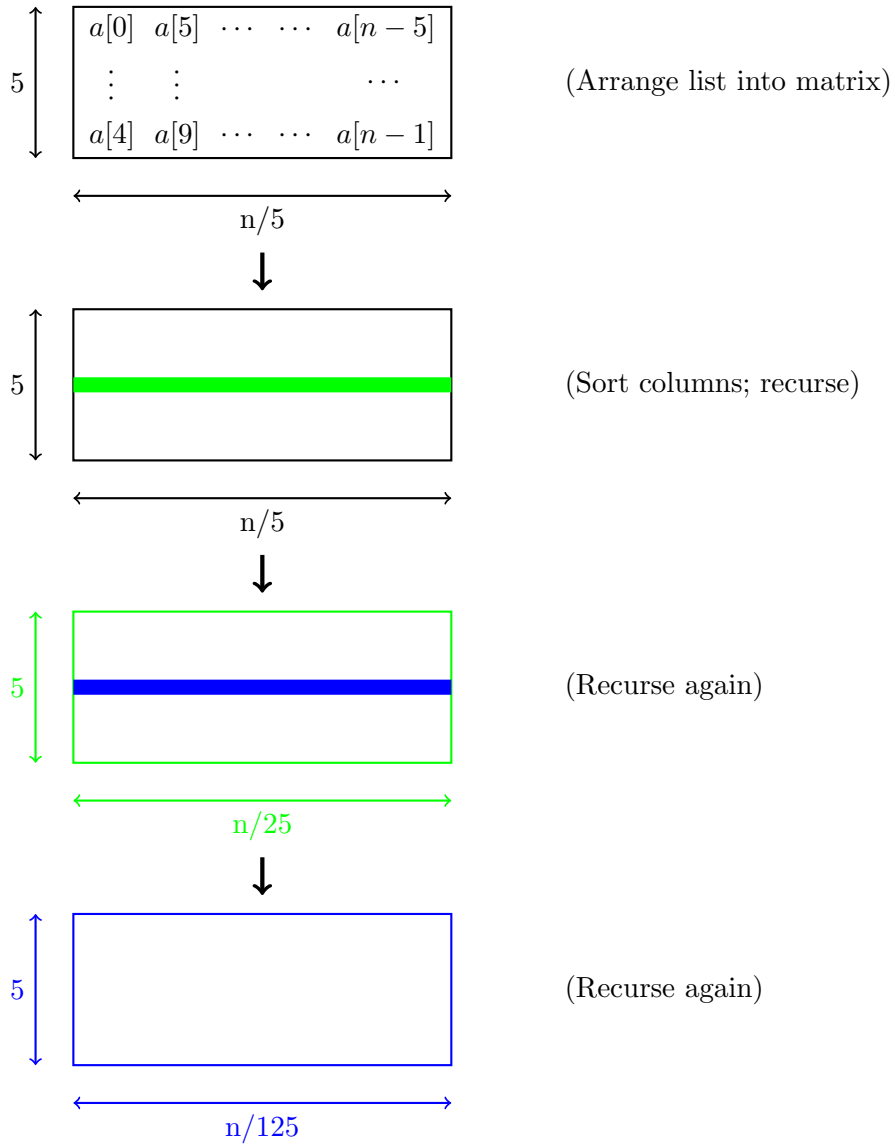

Figure 1: Visualization of Median of Medians

This completes our discussion of median of medians. If curious, the algorithm is due to Blum, Floyd, Pratt, Rivest, and Tarjan. [1]

Last last important note here is that it's important to note the power of randomization. We try to find the pivot manually, but if you randomly pick a number in the array, there's roughly a 40% chance it meets the characteristics we've discussed. However if you fail, you've only wasted linear time, which may sound like a lot, but if you repeat it three or four times, you have a very good chance at finding a good pivot in one of those. It turns

[1] Many famous people in our modern day; four of them are Turing award winners, and many of the algorithms we discuss are due to them.

out that this method is also roughly linear time, but proving that is very complicated. [2]

2 Matrix Multiplications

Divide and Conquer for matrix multiplication is like multiplication on n bit numbers. Also, matrix multiplication is very popular in many fields relating to math like search engines, social networks and artificial intelligence. So, what is the goal? Given two matrices A, B, your goal is to find the product AB. Of course there are different ways of representing matrices, so we'll be using a dense representation and each number is a constant number of bits meaning addition and subtraction over numbers is $\Theta(1)$. [3]

Now since we want to use some sort of divide and conquer, so we need to figure out where to divide. For matrices you could think of any matrix A as the combination of 4 quarters.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \text{ and } B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Now as it turns out the internal products work like scalars, so we can find the product by doing matrix multiplication on that.

$$AB = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} \tag{1}$$

It's worth noting that addition is very easy, it must be $\Theta(n^2)$. Now you see there are 8 sub-problems here of half size, and we're doing $\Theta(n^2)$ work to do the additions. Now if you solve this recursion it becomes $\Theta(n^3)$, but like for n-bit number multiplication, you can lower the number of sub-problems.

2.1 Strassen Matrix Multiplication Algorithm

It turns out that there is an even more efficient matrix multiplication algorithm that uses only 7 matrix multiplications. We define the matrix multiplications as follows:

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix}$$

[2] We'll be discussing randomized algorithms near the end of the semester.

[3] You could also represent matrices in a sparse format.

While this may look like it only works for 2×2 matrices, it turns out that it also works for arbitrary square matrices. For instance, take two square matrices each of size $n \times n$. We can divide each matrix into four submatrices, each of size $\frac{n}{2} \times \frac{n}{2}$. To compute each of the 7 matrix multiplications, we would perform matrix addition on the submatrices, then multiply them together (as illustrated above). During this process, we can continue the recursive division process until the submatrices divide down into the base case, a 2×2 matrix.

To analyze the time complexity of the Strassen matrix multiplication algorithm, we can write the recurrence as

$$T(n) = 7T(n/2) + \mathcal{O}(n^2)$$

because 7 recursive calls are made, each of size $\frac{n}{2} \times \frac{n}{2}$, and performing matrix addition/-subtraction to combine the multiplications is $\mathcal{O}(n^2)$ time. If we solve this recurrence using the Master theorem, we see that Strassen's algorithm takes $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$ time. [4]

Can we do better though? We're unsure if there is some sort of $\Omega(n^2)$, but it's possible that there's some lower bound; this is an open question.

3 Modular Arithmetic

As we've talked about, if you're dealing with small numbers (like 64 bits or 128 bits), modular arithmetic is easy. There are also applications in cryptography, where you may want to take exponents of large numbers (and as a result, numbers can become extremely large) where it is important to carefully consider each operation we make. Often in cryptography, one may want to take exponents in some modulus N; this is called modular exponentiation. [5]

This leads us to the question: for some n-bit numbers x, y and a modulus N, how do you efficiently find $x^y (\mathrm{mod} N)$? Naively, you could multiply x by itself y times. You could do better by multiplying a power of x by itself to increase faster; for instance if you have x^3 and you want x^7, you could multiply by x four times, or you could multiply x^3 by itself and then multiply by x.

We can formalize this in the following algorithm

$$x^y = \begin{cases} (x^{y/2})^2 & \text{if y is even} \\ x \cdot (x^{(y-1)/2})^2 & \text{if y is odd} \end{cases} \tag{2}$$

[4] There are some results that we can get $\Theta(n^{2.3})$. Actually a former Georgia Tech professor named Richard Peng found an interesting result in this area.

[5] It turns out that some algebraic structures have some properties that are very useful in cryptography, and modular arithmetic is fundamental in computations on these structures. Examples include RSA ($\mathbb{Z}/n\mathbb{Z}$) or elliptic curves (finite fields).