1 Algorithms of Arithmetic

1.1 Lecture 1

TODO: Fill in this section.

2 Divide and Conquer Algorithms

2.1 Lecture 2

TODO: Fill in this section

2.2 Lecture 3

2.2.1 Recurrences and Master Theorem

The idea of divide-and-conquer algorithms are to divide the input inot smaller parts, recurse on parts, and combine the parts to build an answer.

To analyze the runtime of divide-and-conquer algorithms, it is useful to derive the following result.

Theorem 2.1 (Master Theorem)
Suppose we have a recurrence

$$T(n) = aT(\frac{n}{b}) + cn^d$$

then, we have

$$T(n) = \begin{cases} \Theta(n^d) & a < b^d \\ \Theta(n^d \log n) & a = b^d \\ \Theta(n^{\log_b a}) & a > b^d \end{cases}$$

Master's theorem can be shown by drawing a recursion tree, and then summing up the work done in each level (proof omitted for brevity). We can also think of the cases as symbolizing the following:

Case	Interpretation
$a < b^d$	The root does most of the work
$a = b^d$	The root and the leaves do an equal amount of work
$a > b^d$	The leaves do most of the work

2.2.2 Matrix Multiplication

Our first example of a divide and conquer algorithm is matrix multiplication.

Consider multiplying two n-by-n matrices A and B.

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{bmatrix}$$

Then the resultant C has entries given by:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

The natural implementation is then to to loop this summation over i and j. This means this will be three nested loop (a loop is needed for the summation). In flops, this runs in $\Theta(n^3)$ operations.

We can try to break our input instead into $n/2$-by-$n/2$ blocks, as shown:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

To find the runtime of this algorithm, let us realize there are 8 multiplications (recursively) and then finally a Θn^2 addition at the end. This means our recurrence is:

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Note that for our Master theorem setup, $8 > 2^2$, so we have

$$T(n) = n^{\log_2 8} = n^3$$

so this is no better than our naive approach.

Using a similar realization to Karatsuba, Strassen in 1969 found the following:

Algorithm 2.1 (Strassen's Algorithm)
Consider two matrices X and Y which are both n-by-n. Break them up into block matrix form of $n/2$-by-$n/2$ matrices as follows:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Define the following:

$$\begin{aligned} P_1 &= A(F - H) \\ P_2 &= (A + B)H \\ &\vdots \\ P_7 &= (A - C)(E + F) \end{aligned}$$

Then,

$$Z = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

Analyzing the runtime, we see that there are 7 $n/2$-by-$n/2$ multiplications, and some n^2 additions, meaning the total runtime is

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

which Master theorem brings to

$$T(n) = \mathcal{O}\left(n^{\log_2 7}\right) \approx \mathcal{O}\left(n^{2.81}\right)$$

However, Strassen has such a big constant factor that the normal n^3 algorithm is still the most widely used.

2.2.3 Sorting

Consider the problem of sorting a length n array A.

Algorithm 2.2 (MergeSort)
First, we define a procedure MERGE that takes two sorted lists and merges them in linear time. To do this, we first keep a pointer on both lists that starts at the beginning of each list. We then compare the pointed-to elements of each list. The lesser element is then added to the output, and the pointer of the list with that element is incremented by one place. This keeps going until all the elements are used. We then use merge to divide-and-conquer the list as follows:

```
function MERGE-SORT($A[1 \dots n]$)
    $B \leftarrow$ MERGE-SORT($A[1 \dots \frac{n}{2}]$)
```

```
        $C \leftarrow$ MERGE-SORT($A[\frac{n}{2}+1 \ldots n]$)
        return MERGE($B, C$)
```

We get the following recurrence for MERGE-SORT:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

, which through master theorem gives us a running time of

$$T(n) = \Theta(n \log n)$$

Another way of implementing this is a bottom up approach:

Algorithm 2.3 (Iterative Merge Sort)

```
  function MERGE-SORT-ITER($A[1 \ldots n]$)
      $Q \leftarrow$ Divide $A$ into $n$ lists of size one
      while $Q$.size() > 1 do
          $X, Y \leftarrow Q$.pop(), $Q$.pop()
          $Q$.push(MERGE($X, Y$))
      return $Q$.pop()
```

Now we can think about the runtime of this algorithm. Think of the algorithm as running in phases. Phase 0 is when lists popped have size 1. Phase 1 is when lists popped have size 2. Phase i is when lists popped have size 2^i. Note that in each phase, each element is looked at exactly once. Thus, the total runtime must be proportional to $n \cdot$ number of phases. How many phases are there? There are $\log n$ phases, giving us the same $\Theta(n \log n)$ runtime!

The best way to see this is with an example:

Example 2.1
Suppose our initial list is:

$$A = \begin{bmatrix} 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{bmatrix}$$

We then split this into sublists of size one in our Q and start iterating:

$$\begin{aligned}
Q &= [[8], [7], [6], [5], [4], [3], [2], [1]] \\
Q &= [[6], [5], [4], [3], [2], [1], [7, 8]] \\
&\vdots \\
Q &= [[7, 8], [5, 6], [3, 4], [1, 2]] \\
Q &= [[3, 4], [1, 2], [5, 6, 7, 8]] \\
Q &= [[1, 2, 3, 4], [5, 6, 7, 8]] \\
Q &= [[1, 2, 3, 4, 5, 6, 7, 8]]
\end{aligned}$$

Our list has been sorted!

Can we sort faster than $n \log n$? It turns out we cannot do any better with an algorithm that uses comparisons to sort (the problem is $\Omega(n \log n)$).

Theorem 2.2 (Fastest Shorting in Comparison Model, Lower Bound)
We will show that $\Omega(n \log n)$ comparisons are needed even if promised A is some permutation of $\{1, \ldots, n\}$ (all distinct as well).

Proof
The first comparison such an algorithm might make might be: is $A_i < A_j$? Then we branch off into two cases for each, where we require another comparison. We can construct a binary tree that models the situation.
First, notice that each leaf is when the algorithm terminates. Note that since every run of the algorithm with a different input produces a different output permutation of the input, there must be at least $n!$ leaves. Consider the maximum depth of this tree T. There are at most 2^T leaves, meaning that $2^T \geq n!$. This means

$$T \geq \log(n!)$$
$$T = \Omega(n \log n)$$

The last claim can be shown by realizing

$$n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$
$$\log(n!) \geq \frac{n}{2}\log\left(\frac{n}{2}\right)$$
$$\log(n!) \geq \frac{n}{2}\log n - \frac{n}{2}$$
$$\log(n!) = \Omega(n \log n)$$

However, there are way more operations than comparisons that can be used for sorting. An example of this is counting sort: if you have n integers and all the integers have values between 1 and b, then you can sort in $\Theta(n + b)$, by just keeping a hashmap of all the elements that fit in each value "bucket" between 1 and b.

The Word RAM model: suppose your machine can store words of size w and you can do any common C operations. The fastest known algorithm following this model (not just comparisons) is: $\mathcal{O}\left(n\sqrt{\log \log n}\right)$. This is also a randomized algorithm.

There are two types of randomized algorithm (which will be revisited). A Monte Carlo randomized algorithm is one whose output may be incorrect with small probability. A Las Vegas algorithm is one whose runtime is fast in expectation, but may be slow with small probability.

2.2.4 Selection/Medians

We now consider the problem of selection. Suppose we want to select the k smallest integer in a list A. Without loss of generality, assume all elements of A are distinct (since we could replace $A[i]$ with the tuple $(A[i], i)$). For selection, there is Quick Select, which we will explore later. Notably, quick select requires knowing the median in linear time. We will instead focus on that problem, (the same as the selection problem for $k = n/2$).

Algorithm 2.4 (Median of Medians)
Take an array A. Then break up the array into subarrays of size 5. Next, we will recursively compute the median of each subarray. Note that this takes constant time to complete since 5 is a constant. Now we have a $N/5$ size array. We then find the median recursively of this smaller problem. Call this median m_1.
Now, we change the array such that all the elements bigger than m_1 end up on the right of m_1, elements smaller than m_1 end up on the left, and m_1 is in the middle of these two parts (this only requires a linear scan). If m_1 is in position less than $n/2$, then the true median sits on its right, so we can recurse on the right half. If m_1 is in position greater than $n/2$, then the true median sits on its left, so we can recurse on the left half. Finally, if m_1 is at exactly position, $n/2$, then we have found the median.
The claim is that at least 30% of the elements are filtered out by comparisons to m_1. To show this, consider m_1 compared to the other medians. Note that it is bigger than 3 of the elements in every median it is bigger than, so, it is bigger than $\frac{3}{5} \cdot \frac{N}{10} = \frac{3}{10}$ of the elements. Thus, if m_1 is in the first half of the array, then it will filter out at least $\frac{3}{10}$ of the numbers. Similarly, you can make a symmetric argument that if m_1 is in the second half, then

it must be less than $\frac{3}{10}$ of the numbers and thus will filter out 30% of them. Either way, we can then produce the following recurrrence:

$$T(n) \leq T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + \Theta(n)$$

This recurrence gives $T(n) = \mathcal{O}(n)$. We can give an inductive argument:

Proof
We will show that $T(n) \leq Bn$ for sufficiently large B, which will imply our big-$\mathcal{O}$ runtime.
Base Case: if n is 1, then we just return the input, so if B is greater than the time needed to return then the base case holds.
Inductive Hypothesis: Suppose that the claim holds for $k < n$.
Inductive Step: By the recurrence and the inductive hypothesis, we have that

$$T(n) \leq B\frac{7n}{10} + B\frac{n}{5} + Cn$$

where C is some other constant. Now, we have

$$\begin{aligned} T(n) &\leq \left(\left(\frac{7}{10} + \frac{1}{5}\right)B + C\right)n \\ &\leq \left(\frac{9}{10}B + C\right)n \\ &\leq Bn \end{aligned}$$

as long as $C \leq \frac{B}{10}$. Since C is fixed, we can set $B = 10C$ to make this true. ■

2.3 Lecture 4

2.3.1 Polynomial Multiplication

Suppose we have two polynomials as inputs:

$$A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{d-1}x^{d-1}$$

$$B(x) = b_0 + b_1x + b_2x^2 + \cdots + b_{d-1}x^{d-1}$$

Then we want the output polynomial C in the following form:

$$C(x) = c_0 + c_1x + \cdots + c_{2d-2}x^{2d-2}$$

Define $N = 2d - 1$ for simplicity, and notice that these can all be considered $N - 1$ degree polynomials if we pad A and B with 0 coefficients on higher order coefficients.

There is a relationship between polynomial and integer multiplication. Given integers α, β, if we want $\gamma = \alpha \times \beta$, we first write them digit-wise as

$$\alpha = \alpha_{N-1}\alpha_{N-2}\dots\alpha_0$$

$$\beta = \beta_{N-1}\beta_{N-2}\dots\beta_0$$

$$A(x) = \alpha_0 + \alpha_1x + \cdots + \alpha_{N-1}x^{N-1}$$

$$B(x) = \beta_0 + \beta_1x + \cdots + \beta_{N-1}x^{N-1}$$

Note that $\alpha = A(10)$ and $\beta = B(10)$, and $\gamma = (A \cdot B)(10)$ plugging in integers for the polynomials is fairly fast (just some additions and multiplication). This shows that integer multiplication and polynomial multiplication are fairly connected.

Algorithm 2.5 ("Straightforward" Algorithm for Polynomial Multiplication)

$$C(x) = c_0 + c_1x + \cdots + c_{2d-2}x^{2d-2}$$

What are these coefficients in terms of a_i and b_i?

$$\begin{aligned} c_0 &= a_0b_0 \\ c_1 &= a_0b_1 + a_1b_0 \\ &\vdots \\ c_k &= \sum_{j=0}^{k} a_jb_{k-j} \end{aligned}$$

Then the algorithm looks something like this:

- Loop over $k = 0$ to $N - 1$
  - Compute c_k with a loop from $j = 0$ to k

Note that this algorithm runs $\Theta(N^2)$.

However, we can do better, since integer multiplication is close to polynomial multiplication.

Algorithm 2.6 (Karatsuba for Polynomials)
Call:

$$\begin{aligned}
A_l(x) &= a_0 + a_1x + a_2x^2 + \cdots + a_{N/2-1}x^{N/2-1} \\
A_h(x) &= a_{N/2}x^{N/2} + \cdots + a_{N-1}x^{N-1} \\
B_l(x) &= b_0 + b_1x + b_2x^2 + \cdots + b_{N/2-1}x^{N/2-1} \\
A_h(x) &= b_{N/2}x^{N/2} + \cdots + b_{N-1}x^{N-1}
\end{aligned}$$

Note that $A(x) = A_l(x) + x^{N/2}A_h(x)$ and $B(x) = B_l(x) + x^{N/2}B_h(x)$. Using the Karatsuba trick, you see that you need 3 multiplications, giving the recurrence:

$$T(N) \leq 3T\left(\frac{3}{2}\right) + \Theta(N)$$

which solves to: $T(N) = \Theta(N^{\log_2 3})$

Here is a fact from elementary algebra:

Note 2.1 (Polynomial Interpolation)
A degree $< N$ polynomial is fully determined by its evaluation on N distinct points.

Proof
Here is an argument for why interpolation works. Represent C as the vector

$$\begin{bmatrix} c = c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix}$$

Then consider the following matrix-vector multiplication:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{N-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N-1} & x_{N-1}^2 & \dots & x_{N-1}^{N-1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix}$$

I.e. we can call this $Vc = y$, solving this equation is only possible if V is full rank.
Fact for "Vandermonde" matrix V:

$$\det\{V\} = \prod_{i<j}(x_i - x_j) \neq 0$$

so we can solve $c = V^{-1}y$, so there is a unique polynomial that interpolates.

This gives rise to the following idea: rather than multiplying directly, we instead evaluate $C(x_0), C(x_1), \ldots, C(x_{N-1})$ for distinct x_i.

To do this evaluation, just evaluate $A(x_i)$ and $B(x_i)$, then finally combine to get $C(x_i)$. Finally, interpolate to set back coefficients from C in terms of these points.

However, interpolation is way too slow. You have to use inversion which takes $\mathcal{O}(n^3)$ flops. Instead, what if we choose V carefully such that it's faster to invert?

Let us establish some types:

1. The Discrete Fourier Transform (DFT) is a **matrix**.
2. The Fast Fourier Transform (FFT) is a **algorithm**.

Definition 2.1 (Discrete Fourier Transform (DFT))
Define $\omega = e^{2\pi\sqrt{-1}/N}$ (primitive root of unity). Now define the DFT matrix F such that $F_{ij} = (\omega^i)^j = \omega^{ij}$.
Imagine evaluating a polynomial at points $1, \omega, \omega^2, \dots, \omega^{N-1}$
This gives the Vandermonde matrix:

$$V = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \vdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

i.e. the DFT matrix.

Here is the most important property of the DFT matrix:

Note 2.2 (Inverse of the DFT matrix)

$$F^{-1} = \frac{1}{N}\overline{F}$$

Note that we can view the following algorithm in two lenses: either a fast way to multiply by F^{-1}, or a fast way to calculate $P(\omega), P(\omega^2), \dots$. We take the latter interpretation.

Algorithm 2.7 (Fast Fourier Transform (FFT))
The goal of this algorithm is to multiply by the DFT quickly. We will take a polynomial interpretation. Consider our polynomial, assuming N is a power of 2:

$$\begin{aligned} P(z) &= p_0 + p_1 z + p_2 z^2 + \dots + p_{N-1} z^{N-1} \\ &= \left(p_0 + p_2 z^2 + p_4 \left(z^2\right)^2 + \dots + p_{N-2}\left(z^2\right)^{N/2-1}\right) + z\left(p_1 + p_3 z^2 + p_5 \left(z^2\right)^2 + \dots\right) \\ &= P_{\text{even}}(z^2) + zP_{\text{odd}}(z^2) \end{aligned}$$

For all N roots of unity, squaring any of them will just give you another Nth root of unity. In fact, it'll give you a $N/2$ root of unity. This means that we only have to evaluate P_{odd} and P_{even} on $N/2$ roots of unity, which means our recurrence becomes:

$$T(N) = 2T\left(\frac{N}{2}\right) + \Theta(N)$$

which solves to:

$$T(N) = N \log N$$

Finally, we give an algorithm for our initial problem of multiplying polynomials.

Algorithm 2.8 (Polynomial Multiplication Via FFT)
First, we use the Fast Fourier Transpose to compute $\hat{a} = Fa$ and $\hat{b} = Fb$.
Then, for $i = 0$ to $N-1$, we compute, $\hat{c_i} = \hat{a_i} \times \hat{b_i}$.
Finally, we have to bring c back into the original basis, i.e. compute $c = F^{-1}\hat{c}$. To do this, notice

$$\begin{aligned} c &= \frac{1}{N}\overline{F}\hat{c} \\ &= \frac{1}{N}\overline{F\overline{\hat{c}}} \end{aligned}$$

which requires just one more use of FFT to multiply $F\overline{\hat{c}}$.
The runtime is thus dominated by the 3 FFTs, giving us: $\mathcal{O}(N \log N)$ runtime, assuming we can multiply/add/-conjugate complex numbers in constant time.

It may seem paradoxical that since it takes N^2 entries to write down F, how come we are faster? The heart of this is that we **never write down** F. Instead, we just use FFT to multiply F by a vector (quickly).

2.3.2 Cross Correlation

Suppose you have two vectors: x and y:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-1} \end{bmatrix}$$

$$y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

where $n \geq m$. Suppose we want all shifted dot products of x with y, i.e.

$$\begin{aligned} &x_0 y_0 + x_1 y_1 + \cdots + x_{m-1} y_{m-1} \\ &x_0 y_1 + x_2 y_2 + \cdots + x_{m-1} y_m \\ &\vdots \end{aligned}$$

However, we can reduce cross correlation to a problem we have already solved, polynomial multiplication!

Algorithm 2.9 (Cross Correlation Via Polynomial Multiplication)
Define the following:

$$\begin{aligned} X(z) &= x_{m-1} + x_{m-2} z + \cdots + x_0 z^{m-1} \\ Y(z) &= y_0 + y_1 z + y_2 z^2 + \cdots + y_{n-1} z^{n-1} \\ Q(z) &= (X \cdot Y)(z) = q_0 + q_1 z + \ldots \end{aligned}$$

Then let us investigate the coefficients of Q:

$$\begin{aligned}
q_0 &= x_{m-1}y_0 \\
q_1 &= x_{m-1}y_1 + x_{m-2}y_0 \\
&\vdots \\
q_{m-1} &= x_{m-1}y_{m-1} + \cdots + x_1y_1 + x_0y_0 \\
q_{(m-1)+1} &= x_{m-1}y_m + \cdots + x_1y_2 + x_0y_1 \\
&\vdots
\end{aligned}$$

Thus, we can multiply the polynomials X and Y, then take the coefficients $m-1$ and bigger to get all of our cross correlation terms.