8 Divide and Conquer

Written by Ethan Pronovost. epronovost@caltech.edu

8.1 Mergesort

The divide and conquer technique is a recursive technique that splits a problem into 2 or more subproblems of equal size. General problems that follow this technique are sorting, multiplication, and discrete Fourier Transforms. For an example of this, consider the Mergesort algorithm.

Algorithm 8.1 (Mergesort)**.** Take a list $x[n]$ of integers. In the base case, if $n \leq 1$, return x.

Otherwise, split the list into the left and right halves, $l[\lceil \frac{n}{2} \rceil]$ and $r[\lfloor \frac{n}{2} \rfloor]$. Recursively sort both these lists. To combine these two lists into one list $a[n]$, set $i, j, k = 0$. For each k in $[0, n)$, if $l[i] \leq r[j]$, set $a[k] \leftarrow l[i]$ and increment i. Otherwise, set $a[k] \leftarrow r[j]$ and increment j. Finally, return the array a.

Proof of Correctness. Let us show that the returned array will always be sorted for all n. In the base case, if $n \leq 1$, then the list is already sorted. Inductively, take a list of length n. By induction, the sorted sublists l and r will be correctly sorted. For any indexes i and j, if $l[i] \leq r[j]$, then $l[i'] \leq r[j']$ for all $i' \leq i$ and $j' \geq j$. Therefore, merging the two lists in the described way will produce a total combined list in sorted order.

Complexity. Analyzing the recurrence relation, we make 2 calls to sort lists of half the size. The merge step requires iterating over both lists, which involves at most $n - 1$ comparisons. Thus, $T(n) = 2 \cdot T(\frac{n}{2}) + (n-1)$. Using the *Master Theorem* defined below, it follows that $T(n) \in \Theta(n \log n)$.

8.2 Generic Algorithm Design

A *Divide and Conquer* algorithmic strategy applies when we can divide a problem into parts, solve each part, and then combine the results to yield a solution to the overall problem. The generic design is as follows:

Algorithm 8.2 (Divide and Conquer)**.**

1. For positive integer $b > 1$, divide the problem into b parts.
2. (Divide) Recursively solve the subproblems.
3. (Conquer) Consider any situations that transcend subproblems.

8.3 Complexity

By the construction, the time complexity of the algorithm $T(n)$ satisfies the recurrence relation:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \tag{8.1}$$

where $f(n)$ is the time it takes to compute step 3 (above). In class, we looked at the following theorem about runtime:

Theorem 8.3. *(Master Theorem) If $T(n)$ satisfies the above recurrence relation, then*

- *if $\exists\ c > 1, n_0$ such that for $n > n_0$, $af(n/b) \geq cf(n)$ then $T(n) \in \Theta(n^{\log_b a})$*
- *if $f(n) \in \Theta(n^{\log_b a})$ then $T(n) \in \Theta(n^{\log_b a} \log n)$*
- *if $\exists\ c < 1, n_0$ such that for $n > n_0$, $af(n/b) \leq cf(n)$ then $T(n) \in \Theta(f(n))$*

Proof. Convince yourself that $a^{\log_b n} = n^{\log_b a}$. By induction, its easy to see that

$$T(n) = a^j \cdot T\left(\frac{n}{b^j}\right) + \sum_{k=0}^{j} a^k f\left(\frac{n}{b^k}\right) \tag{8.2}$$

Apply to $j = \log_b n$ and recognize $T(1)$ is a constant so

$$T(n) = \Theta(a^{\log_b n}) + \sum_{k=0}^{\log_b n} a^k f\left(\frac{n}{b^k}\right) \tag{8.3}$$

Let's consider the first case. We apply the relation to get[1]

$$\begin{aligned} T(n) &= \Theta(a^{\log_b n}) + a^{\log_b n} f(1) \sum_{k=0}^{\log_b n} c^{-k} \\ &= \Theta(a^{\log_b n}) + \Theta\left(a^{\log_b n} \frac{c}{c-1}\right) \\ &= \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}) \end{aligned} \tag{8.4}$$

[1] I've made the simplifaction here that $n_0 = 0$. As an excercise, convince yourself this doesn't effect anything, just makes the algebra a little more complicated.

For the second case,

$$\begin{aligned} T(n) &= \Theta(a^{\log_b n}) + \sum_{k=0}^{\log_b n} a^k \Theta\left(\left(\frac{n}{b^k}\right)^{\log_b a}\right) \\ &= \Theta(n^{\log_b a}) + \sum_{k=0}^{\log_b n} a^k \left(\frac{\Theta(n^{\log_b a})}{a^k}\right) \\ &= \Theta(n^{\log_b a} \log n) \end{aligned} \tag{8.5}$$

For the third case, recognize that it's nearly identical to the first except the summations are bounded in the other direction which leaves $f(n)$ as the dominating term. □

8.4 Quicksort

In most cases, the preferred sorting algorithm of choice is *Quicksort*. This algorithm appears very similar to Mergesort, but it partitions the inputs not by location, but by value. The space complexity is also very efficient.

In the quicksort algorithm, we choose a pivot value p. We then sort the array in a cursory way, such that all entries with value $\leq p$ are to the left of all entries with value $> p$. Recursively, then, we know that the final sorted list will be the sorted "less than" list, followed by the sorted "greater than" list, and so we recursively sort both parts.

An algorithm to sort an array X of length n in place around a pivot p is:

- Initialize $i \to 0$ and $j \to n-1$
- Increment $i++$ while $X[i] < p$ and decrement $j--$ while $X[j] > p$. If $i \geq j$, then return j, the index of the "boundary". Otherwise, swap $X[i]$ and $X[j]$, and repeat this loop.

In many naive implementations, the pivot value is taken to be an arbitrary value of the array (e.g. the first, last, or a random index). Clearly, this algorithm will perform best when the pivot divides the array roughly in half, and so we want the pivot to be as close to the median as possible.

8.5 Rank Selection

The *rank* of any element is the number of elements in the list with lesser or equal value. For example, the median will have rank $\frac{n}{2}$. With this definition, we give the following algorithm to find the element of a given rank in a list.

Algorithm 8.4. Rank Selection: Select(r, L) Write out the array L into 5 rows, and consider each column separately. We can trivially sort each column in constant time, so after an $\Theta(n)$ operation

we can have a list of columns, sorted, such that the middle row has the medians of each column. Recursively sort these columns by their median value, yielding a final "median of medians" z.

Compare all of L to this value z, compute rank(z) and generate two lists: $L_1 := \{\ell \in L \mid \ell < z\}$ and $L_2 := \{\ell \in L \mid \ell > z\}$. If rank($z$) $< r$, recursively call Select(r, L_1). Otherwise, call Select($r - \text{rank}(z), L_2$).

Clearly, applying this algorithm with an initial input $r = \frac{\text{len}(L)}{2}$ will yield the median. To explore the efficiency of this algorithm, we first give the following lemma.

Lemma 8.5 (Rank of z)**.** *The rank of the element z selected by the above algorithm is bounded by $\frac{3n}{10} \leq \text{rank}(z) \leq \frac{7n}{10}$, where n is the length of L.*

To show this, consider the diagram.

small	small	small	small	small	*	*	*	*
small	small	small	small	small	*	*	*	*
small	small	small	small	z	large	large	large	large
*	*	*	*	large	large	large	large	large
*	*	*	*	large	large	large	large	large

By the construction of the algorithm, each median value will be greater than the rows above it in that column, and to the median values of other columns to the left. Therefore, z must be greater than all the entries labeled "small" in the diagram. A similar argument goes for the larger case.

There are $\frac{3n}{10}$ such "smaller" values and "larger" values each, which gives the bound on the rank of z.

Runtime Let $T_{\text{SEL}}(n)$ be the runtime of this algorithm on a list of length n. In each iteration, we must sort each column (in $\Theta(n)$ time), and then recursively sort the median values (a subproblem of size $\frac{n}{5}$). Given the above lemma, we can bound the size of L_1 and L_2 by $\frac{7n}{10}$. Thus, an equation describing the runtime is

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

This is a slightly different form than the Master Theorem as given. However, it is a straightforward exercise to note that if $T(n) \leq \Theta(n) + \sum T(c_i n)$, where $\sum c_i < 1$, then indeed $T(n) \in \Theta(n)$. As this applies for the above case, we get that the complexity of this algorithm is linear in the length of the list.

Application to Quicksort Sorting the list about the pivot is already a $\Theta(n)$ computation, so adding this step beforehand to compute the median does not affect the asymptotic runtime. However, as you can probably imagine, the constants for this algorithm are rather large, so doing this at each step is by no means necessarily "quick".

8.6 Randomized Quicksort

Consider a different algorithmic approach, in which we select an item from the list to serve as the pivot uniformly at random.

The course slides give one way of showing that the expected number of comparisons performed by this algorithm is $T_Q(n) = 2(n+1)H_n - 4n$, where $H_n = \sum_{\ell=1}^{n} 1/\ell$. (We understand this function well: H_n converges to $\gamma + \log n$ where log is natural logarithm, and $\gamma \cong 0.577$ is the Euler-Mascheroni constant.) So, roughly $(2/\lg e)n \lg n$ (where lg is logarithm base 2).

Here is another way of doing the analysis. Think of the elements laid out in their sorted order $1, \ldots, n$. Think of the process of picking pivots as follows. We pick a random element i to be the first pivot—so let's give it the label $\ell(i) = 1$. Now all of $1, \ldots, i-1$ are going to be quicksorted separately, and all of $i+1, \ldots, n$ are going to be quicksorted separately. So in each of these regions we are going to uniformly pick a pivot; give each of these a label of 2. And so forth, in rounds, in each existing nonempty interval we pick a new element and label it with the number of the round.

Now, think of each comparison from the point of view of the non-pivot element of the pair. That is, from the point of view of the element which will receive the higher label. We can count all comparisons by "charging" them to the higher-label element. So think of some element j, $1 \leq j \leq n$. To how many lower-label pivots will it be compared?

An easy way to upper bound this is to use the case in which j itself has the highest possible label. And moreover, we'll bound the comparisons by the number of those against pivots $k < j$, plus the number against pivots $k > j$. Let's just write down the second case. What is the probability that a specific k occurs as a pivot of level ℓ that is compared to i? This is precisely the probability that when k is chosen to be a pivot, none of $i+1, \ldots, k-1$ have yet been chosen to be pivots. But since $i+1, \ldots, k$ were equally likely to be chosen at this juncture, this probability is at most $1/(k-i)$. So, the expected number of comparisons of j against k's in the range $k > j$, is at most $\sum_{k=i+1}^{n} \frac{1}{k-i} = H_{n-k}$. Now throwing in also $k < j$, and then summing over j, we have the upper bound $2\sum_{1 \leq j < k \leq n} \frac{1}{k-j} = 2\sum_{m=1}^{n-1} \frac{n-m}{m} \leq 2n \sum_{1}^{n-1} \frac{1}{m} \leq 2nH_n$.

8.7 Lower bound on Sorting

We'll describe here a lower bound on deterministic comparison-based sorting algorithms. (An extension of this argument works even for randomized comparison-based sorting algorithms.)

In a comparison-based sorting algorithm, the inputs are some $x_1, \ldots, x_n$, and all we learn about them we learn from comparisons "is $x_i < x_j$?".
When the inputs are restricted to be from a small domain, say $x_i \in \{0, 1\}$, sorting actually gets easier. So for the purpose of a lower bound, we rely on the fact that they come from a large enough domain that all might be distinct; since our algorithm must handle that case, we can just simplify our exposition by assuming that all are indeed distinct; and in fact we may as well assume they are just the numbers $1, \ldots, n$ in some order. That means we can think of the input x simply as a permutation of the numbers $1, \ldots, n$.

Now, our algorithm must put the inputs in correct order no matter what permutation they arrived in. That means that for any two distinct permutations x, x', at some point our algorithm must "tell them apart"—that is, there is some first comparison at which we ask "is $x_i < x_j$?" and get a different answer than we get for "is $x'_i < x'_j$?". (Since the algorithm can be adaptive, the queries from that point and on might be different in the two algorithms.)

But this means that if you look at the binary sequence of answers the algorithm gets to its comparison queries, that binary sequence is different for every permutation x. The number of permutations of n elements is $n! \cong e^{n \log n - n} = 2^{n \lg n - n/\log 2}$, so in order to distinguish them all, the number of comparisons on a worst-case input (and in fact on most inputs) has to be at least $n \lg n - n/\log 2$.

This $\Omega(n \log n)$ lower bound for sorting is within a constant factor of the upper bounds provided by the algorithms we have seen.

8.8 Fast Integer Multiplication

Consider two n digit integers X and Y. Let $X = X_L X_R$ as digits, where X_L is the left half of X, and X_R is the right half, and similarly $Y = Y_L Y_R$. Numerically, $X = 2^{\frac{n}{2}} \cdot X_L + X_R$, and similarly for Y. The naive grade-school algorithm for multiplication would be to use

$$X \cdot Y = 2^n \cdot X_L \cdot Y_L + 2^{\frac{n}{2}} \cdot (X_L \cdot Y_R + X_R \cdot Y_L) + X_R \cdot Y_R. \tag{8.6}$$

Multiplying by an exponent of 2 can be easily performed by a bit shift. However, this approach uses 4 multiplications of $\frac{n}{2}$ digit numbers. Using the Master Theorem, the runtime would thus

be $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$, and so $T(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$. This naive divide and conquer application does not yield a better runtime.

An improved algorithm is *Karatsuba Multiplication*. In this approach, we set $A = X_L \cdot Y_L$, $B = X_R \cdot Y_R$, and $C = (X_L + X_R) \cdot (Y_L + Y_R)$. Note that the middle term of (8.6) is given by $X_L \cdot Y_R + X_R \cdot Y_L = C - A - B$. Therefore, with only three multiplications, we rewrite this equation as

$$X \cdot Y = 2^n \cdot A + 2^{\frac{n}{2}} \cdot (C - A - B) + B. \tag{8.7}$$

This algorithm makes only 3 recursive calls, so $T(n) = 3 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$, so that $T(n) \in \Theta(n^{\log_2 3}) \approx \Theta(n^{1.58})$. This demonstrates the power of the divide and conquer approach.

8.8.1 Convolution

Given two arrays s and t of size n and m, respectively, the *convolution* $s \star t$ of the two arrays is an array of length $n + m$, with the terms given by

$$(s \star t)_k = \sum_{i+j=k} s_i \cdot t_j \tag{8.8}$$

If you think of the arrays as coefficients of polynomials (i.e. $s(x) = \sum_{i=0}^{n} s_i x^i$), then convolution is simply polynomial multiplication.

Observe that the binary representation of a number a can be thought of the coefficients $s_i \in \{0, 1\}$ that satisfy $s(2) = a$. The Karatsuba algorithm for multiplication can be generalized to convolution, yielding an $\Theta(n^{\log_2 3})$ algorithm.

8.9 Fast Division, Newton's Method

For a given $0 \neq t \in \mathbb{R}$, we must compute the inverse $\frac{1}{t}$. Noting that this inverse is the root of $f(x) = t - \frac{1}{x}$, we can use the Newton Raphson method of successive approximation to find this root. Concretely,

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{t - \frac{1}{x_i}}{\frac{1}{x_i^2}} = x_i - x_i^2 t + x_i = 2x_i - t x_i^2.$$

Algorithm 8.6 (Fast Division)**.** Let p be the unique exponent for which $2^p \in [t, 2t)$. Initialize a seed $x_0 = 2^{-p}$. Iteratively, compute $x_{i+1} = 2x_i - t x_i^2$ until a sufficient accuracy is reached.

Proof of Correctness. By definition, $\frac{1}{2t} < x_0 \leq \frac{1}{t}$. We can define the associated error of this approximation as $\epsilon_0 = 1 - t x_0 < \frac{1}{2}$.

Iteratively, $\epsilon_{i+1} = 1 - tx_{i+1} = 1 - t(2x_i + tx_i^2) = (1 - tx_i)^2 = \epsilon_i^2$, and so $\epsilon_i \leq 2^{-2^i}$. In other words, each iteration doubles the number of significant digits in the approximation.

Thus, after sufficient iterations, any arbitrary accuracy can be achieved.

8.10 Fast Fourier Transform

We saw above in Section 8.8.1 an $\Theta(n^{\log_2 3})$ time algorithm for polynomial multiplication. Let us now consider a better approach for multiplying polynomials in $\mathbb{C}_n[x]$ (polynomials of degree at most $n-1$ with complex coefficients).

8.10.1 A Change of Basis

$\mathbb{C}_n[x]$ can be thought of as an n dimensional vector space over $\mathbb{C}$, with the vector components simply the coefficients of the polynomial. For example, for $n = 3$, we would map

$$(4 - 2i) + ix + 2x^2 \mapsto \begin{pmatrix} 4 - 2i \\ i \\ 2 \end{pmatrix}. \tag{8.9}$$

Simple checking will verify that this definition observes the definitions of a vector space (i.e. group addition and scalar multiplication by $\mathbb{C}$). This mapping defines the basis $\mathcal{B} = \{1, x, \ldots, x^{n-1}\}$. This basis makes polynomial addition trivially $\Theta(n)$, but requires convolution to perform multiplication. However, just like any vector space, we can choose an alternative basis for $\mathbb{C}_n[x]$.

Theorem 8.7. *(Evaluation Map) Pick any n distinct points $\{\zeta_1, \ldots, \zeta_n\} \in \mathbb{C}$. Then the map*

$$\mathbb{C}_n[x] \to \mathbb{C}^n \qquad t(x) \mapsto (t(\zeta_1), \ldots, t(\zeta_n)) \tag{8.10}$$

is a linear isomorphism (i.e. injective and surjective).

Proof. To see that this map is linear, note that for any polynomial $t(x) \in \mathbb{C}_n[x]$ and scalar $\lambda \in \mathbb{C}$, $\lambda \cdot t(x) \mapsto (\lambda t(\zeta_1), \ldots, \lambda t(\zeta_n)) = \lambda \cdot (t(\zeta_1), \ldots, t(\zeta_n))$.

To see that this map is surjective, pick some set $\{y_i\}_{i=1}^n$ of desired values that a polynomial should

take at $\{\zeta_i\}$ respectively.[2] Using *Lagrange interpolation*, define a list of n special polynomials

$$p_i(x) = \prod_{k \neq i} \frac{x - \zeta_k}{\zeta_i - \zeta_k}. \tag{8.11}$$

Note that for any $j \neq i$, then $p_i(\zeta_j) = 0$, since there will be a $\zeta_j - \zeta_j$ term in the numerator. On the other hand, $p_i(\zeta_i) = 1$, since each of the terms becomes 1. Therefore, restricted to the $\{\zeta_i\}$, each p_i is a "delta function". Since there are n points, and thus $n-1$ points where $k \neq i$, the degree of these polynomials is $n-1$, and so $p_i \in \mathbb{C}_n[x]$ (as we would hope).

Given these polynomials, and a set of desired values $\{y_i\}$, define

$$p(x) = \sum_{i=1}^{n} y_i \cdot p_i(x). \tag{8.12}$$

Then $p(x) \in \mathbb{C}_n[x]$ and $p(\zeta_i) = y_i$ for all i.

To show that this map is an isomorphism (i.e. 1-1), we consider the change of basis matrix that maps the vector of the coefficients to the new vector of the evaluations at $\{\zeta_i\}$.

$$\begin{pmatrix} t(\zeta_1) \\ t(\zeta_2) \\ \vdots \\ t(\zeta_n) \end{pmatrix} = \begin{pmatrix} 1 & \zeta_1 & \zeta_1^2 & \cdots & \zeta_1^{n-1} \\ 1 & \zeta_2 & \zeta_2^2 & \cdots & \zeta_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \zeta_n & \zeta_n^2 & \cdots & \zeta_n^{n-1} \end{pmatrix} \begin{pmatrix} t_0 \\ t_1 \\ \vdots \\ t_{n-1} \end{pmatrix} \tag{8.13}$$

The square matrix used in this change of basis is the *Vandermonde matrix* generated by $\{\zeta_i\}$. This matrix is nonsingular (the determinant is $\prod_{0 \leq i < j \leq n-1} (\zeta_j - \zeta_i) \neq 0$ since all the points are distinct), and thus invertible. Therefore, this map is bijective.[3] □

Because this map is an isomorphism, we can take polynomials in the conventional representation, convert them to the *evaluation basis*, perform operations on them, and then convert them back.

8.10.2 Better Multiplication

In the original polynomial basis, addition was easy, but multiplication required convolution. However, in this new basis representation, $f(x) \cdot g(x)$ becomes $(f(\zeta_1) \cdot g(\zeta_1), \ldots, f(\zeta_n) \cdot g(\zeta_n))$, and so

[2] What about the value at other points $x \notin \{\zeta_i\}$? The vector of the $\{y_i\}$ is the vector over the new basis that we are trying to create. With this basis, this is the most amount of detail we can contain.

[3] This proof also shows surjectivity by simply comparing the dimensions of the vector spaces. However, it is good to see how to explicitly construct a preimage for any set of points $\{y_i\}$.

we can simply multiply each component. Since the degree of the product is at most $2n-2$, the product will be contained in $f(x) \cdot g(x) \in \mathbb{C}_{2n-1}[x]$, and so we should use $2n-1$ evaluation points.

This insight gives us an alternative approach to perform polynomial multiplication.

Algorithm 8.8 (Convolution via point-wise multiplication)**.** Let $s(x)$ and $t(x)$ be two polynomials in $\mathbb{C}_n[x]$.

1. Choose $2n-1$ points $\{\zeta_i\} \subset \mathbb{C}$.
2. Convert s and t to $(s(\zeta_1), \ldots, s(\zeta_{2n-1}))$ and $(t(\zeta_1), \ldots, t(\zeta_{2n-1}))$.
3. Multiply each component, yielding a vector $a = (s(\zeta_1)t(\zeta_1), \ldots, s(\zeta_{2n-1})t(\zeta_{2n-1}))$.
4. Invert the evaluation map to yield a polynomial $a(x) = s(x) \cdot t(x)$.

This algorithm looks great (no convolution!), but let us consider the complexity. The evaluation map and its inverse are matrix-vector multiplications. Simply writing down the full Vandermonde matrix is $\Theta(n^2)$, which is already worse than the $\Theta(n^{\log_2 3})$ runtime we got using the Karatsuba approach.

8.10.3 Fourier Transform

There are two key aspects of Algorithm 8.8 that allow us to improve the performance. The entire matrix is uniquely determined by the $2n-1$ points $\{\zeta_i\}$. Furthermore, we can choose these points in any way we like, so long as they are all distinct.

Let m be the smallest power of 2 greater than $2n-2$. Then clearly $\mathbb{C}_{2n-1}[x] \subset \mathbb{C}_m[x]$, so let us treat the problem of two polynomials of degree less than m. Let $\omega = e^{2\pi i/m}$ be a primitive m^{th} root of unity.[4] Define $\zeta_j = \omega^{j-1}$.

Given this choice, the Vandermonde matrix becomes

$$F_m(\omega) = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{m-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(m-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{m-1} & \omega^{2(m-1)} & \omega^{2(m-1)} & \ldots & \omega^{(m-1)^2} \end{pmatrix}, \tag{8.14}$$

[4] A primitive m^{th} root of unity ω is a number $\omega \in \mathbb{C}$ such that $\omega^m = 1$, but $\omega^i \neq 1$ for all $1 \leq i < m$.

called the Fourier transform over $\mathbb{Z}/m$.[5]

Given the specific structure of this matrix, we will show a way to multiply a vector by this matrix, or its inverse, in $\Theta(m \log m)$ time.

8.10.4 FFT Divide and Conquer Approach

To divide the problem of multiplication by $F_m(\omega)$ into smaller subproblems, we'll split the columns of the matrix into the even and odd columns ($F_{m\ \text{EVEN}}(\omega)$ and $F_{m\ \text{ODD}}(\omega)$). To see how this approach works more clearly, we will look at the case of $m = 8$.

First consider the even columns. Since $\omega^8 = 1$, the even columns simplify down to

$$F_{8\ \text{EVEN}}(\omega) = \left(\begin{array}{cccc} 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^6 & \omega^4 & \omega^2 \\ \hline 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^6 & \omega^4 & \omega^2 \end{array}\right) = \begin{pmatrix} F_4(\omega^2) \\ \hline F_4(\omega^2) \end{pmatrix} = \begin{pmatrix} I_4 \\ I_4 \end{pmatrix} F_4(\omega^2) \tag{8.15}$$

where I_4 is the 4×4 identity matrix. Similarly, the odd columns are

$$F_{8\ \text{ODD}}(\omega) = \left(\begin{array}{cccc} 1 & 1 & 1 & 1 \\ \omega & \omega^3 & \omega^5 & \omega^7 \\ \omega^2 & \omega^6 & \omega^2 & \omega^6 \\ \omega^3 & \omega & \omega^7 & \omega^5 \\ \hline \omega^4 & \omega^4 & \omega^4 & \omega^4 \\ \omega^5 & \omega^7 & \omega & \omega^3 \\ \omega^6 & \omega^2 & \omega^6 & \omega^2 \\ \omega^7 & \omega^5 & \omega^3 & \omega \end{array}\right). \tag{8.16}$$

Observe that the i^{th} row of this matrix is ω^{i-1} times the corresponding i^{th} row of the even column

[5] $\mathbb{Z}/m$ is the addative group of intergers modulo m. For example, the classic "clock arithmetic" where $12 \equiv 0$ and so $9 + 9 = 18 \equiv 6$ is $\mathbb{Z}/12$. This structure arrises from the multiplicative group of the primitive m^{th} root of unity because $\omega^m = 1 = \omega^0$, and so for any power i, $\omega^i = \omega^{i \mod m}$.

matrix. So we can express this matrix as

$$F_{8\ \mathrm{ODD}}(\omega) = \left(\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & \omega & 0 & 0 \\ 0 & 0 & \omega^2 & 0 \\ 0 & 0 & 0 & \omega^3 \\ \hline \omega^4 & 0 & 0 & 0 \\ 0 & \omega^5 & 0 & 0 \\ 0 & 0 & \omega^6 & 0 \\ 0 & 0 & 0 & \omega^7 \end{array}\right) \cdot F_4(\omega^2). \tag{8.17}$$

Using these facts, to multiply by $F_8(\omega)$, we can multiply by

$$\underbrace{\begin{pmatrix} 1 & & & & & & & \\ & & 1 & & & & & \\ & 1 & & & & & & \\ & & & & 1 & & & \\ & & & 1 & & & & \\ & & & & & & 1 & \\ & & & & & 1 & & \\ & & & & & & & 1 \end{pmatrix}}_{C^{-1}} \times \underbrace{\left(\begin{array}{cccc|cccc} 1 & & & & 1 & & & \\ & 1 & & & & \omega & & \\ & & 1 & & & & \omega^2 & \\ & & & 1 & & & & \omega^3 \\ \hline 1 & & & & \omega^4 & & & \\ & 1 & & & & \omega^5 & & \\ & & 1 & & & & \omega^6 & \\ & & & 1 & & & & \omega^7 \end{array}\right)}_{A} \times \underbrace{\left(\begin{array}{cc|cc} F_4(\omega^2) & & & \\ & & & \\ \hline & & F_4(\omega^2) & \\ & & & \end{array}\right)}_{B} \times \underbrace{\begin{pmatrix} 1 & & & & & & & \\ & & 1 & & & & & \\ & & & & 1 & & & \\ & & & & & & & 1 \\ & 1 & & & & & & \\ & & & 1 & & & & \\ & & & & & 1 & & \\ & & & & & & & 1 \end{pmatrix}}_{C} \tag{8.18}$$

The first matrix C shuffles around the rows of the column vector into the even and odd indices. We then recursively do two matrix-vector multiplications of size $\frac{m}{2}$, and use the result to get the even and odd rows. Applying C^{-1} restores the rows to the final correct order.[6]

Consider the time complexity of these operations. The first step, reordering the rows, is $\Theta(m)$. We then apply two subproblems (B) of half the size, $2 \cdot T(\frac{m}{2})$. Multiplication by a diagonal matrix (A) can be done in $\Theta(m)$, as can the row reshuffling C^{-1}. The overall recursive relation is thus

$$T(m) = 2 \cdot T\left(\frac{m}{2}\right) + \Theta(m) \tag{8.19}$$

[6]This last step isn't strictly necessary. Ommitting it simply uses a reordering of the evaluation basis.

Applying the Master Theorem (8.3), we see that the time complexity of this algorithm is $\Theta(m \log m)$.

The last step of algorithm 8.8 requires multiplying by the inverse of $F_m(\omega)$.

Claim 8.9. $(F_m(\omega))^{-1} = \frac{1}{m} F_m(\omega^{-1})$.

Proof. To show that two matrices are inverses, it suffices to show that their product is the identity matrix, and so the terms of the product matrix are the Kronecker delta function $m_{i,j} = \delta(i-j)$.[7]

Solving for the terms of the product matrix,

$$\left(F_m(\omega) \cdot \frac{1}{m} F_m(\omega^{-1})\right)_{i,j} = \frac{1}{m} \sum_{\ell=0}^{m-1} \omega^{j\ell} \omega^{-i\ell} = \frac{1}{m} \sum_{\ell=0}^{m-1} (\omega^{j-i})^{\ell}.$$

As ω is an m^{th} root of unity, this sum will be 0 so long as $j \neq i$. If $j = i$, then this sum is simply one, and so

$$\frac{1}{m} \sum_{\ell=0}^{m-1} (\omega^{j-i})^{\ell} = \delta(j-i).$$

□

Given this, we can give an improved way to perform polynomial convolution.

Algorithm 8.10 (FFT)**.** Take two polynomials $t(x), s(x) \in \mathbb{C}_n[x]$. Let m and ω be defined as above.

1. Convert $t(x), s(x)$ to $(t(\omega^1), \ldots, t(\omega^{2m-1}))$ and $(s(\omega^1), \ldots, s(\omega^{2m-1}))$ with FFT. Time: $\Theta(n \log n)$
2. Multiply the corresponding values, yielding $(t(\omega^1)s(\omega^1), \ldots, t(\omega^{2m-1})s(\omega^{2m-1}))$. Time: $\Theta(n)$.
3. Perform the inverse evaluation map with the FFT. Time: $\Theta(n \log n)$.

This algorithm yields a $\Theta(n \log n)$ algorithm for convolution (or equivalently polynomial multiplication).

8.11 Matrix Computations

8.12 Matrix Determinant and Inverse as hard as Multiplication

8.13 Strassen's Laser for Matrix Multiplication

[7] The Kronecker delta function is defined by $\delta(x) = \begin{cases} 1 \text{ if } x = 0 \\ 0 \text{ else} \end{cases}$