What’s the fastest way to sort?

Mergesort has an impressive running time of $O(n \log n)$ for sorting $n$ elements, but there are nevertheless situations which make us wish for something even faster — for instance, when we need to sort a database of hundreds of thousands of medical records or credit card transactions. Is it conceivable that a linear-time sorting algorithm exists?

In certain special cases, the answer is yes: the canonical such example is when the elements to be sorted are integers which are all within $O(n)$ of each other (Exercise).

But in general, barring these kinds of lucky circumstances, sorting does require $\Omega(n \log n)$ operations. The model we have in mind is a very general one, where the elements are of arbitrary type and information about their ordering is provided in the form of a comparison function which takes any two elements and decides which is the smaller of the two.

A sorting algorithm performs a series of such comparisons, and then outputs an ordering. We can depict this process as a binary tree, with each node representing a comparison operation, and the two branches coming out of it representing the possible outcomes.

Each leaf of the tree corresponds to a particular ordering of the $n$ elements, and so there are at least $n!$ leaves. Thus the height of the tree must be at least $\log n! = \Omega(n \log n)$, which means that some input arrays require at least this many comparison operations to sort.

5 Medians

The median of a list of numbers is its 50th percentile: half the numbers are bigger than it, and half are smaller. For instance, the median of $[45, 1, 10, 30, 25]$ is 25, since this is the middle element when the numbers are arranged in order. If the list has even length, there are two choices for what the middle element could be, in which case we pick the smaller of the two, say.

The purpose of the median is to summarize a set of numbers by a single, typical value. The mean, or average, is also very commonly used for this, but the median is in a sense more typical of the data: it is always one of the data values, unlike the mean, and it is less sensitive to outliers. For instance, the median of a list of a hundred 1’s is (rightly) 1, as is the mean. However, if just one of these numbers gets accidentally corrupted to 10000, the mean shoots up above 100, while the median is unaffected.

Computing the median of $n$ numbers is easy: just sort them. The only problem is that this
takes $O(n \log n)$ time, whereas we would ideally like something linear. We have reason to be hopeful, because sorting is doing far more work than we really need – we just want the middle element, and don’t care about the relative ordering of the rest of them.

When looking for a recursive solution, it sometimes helps to consider a more general version of the problem, because this gives a more powerful step to recurse upon. In our case, the generalization we will consider is selection.

**Selection**

*Input:* A list of numbers $S$; an integer $k$

*Output:* The $k^{th}$ smallest element of $S$

For instance, if $k = 1$, the minimum of $S$ is sought, whereas if $k = \lfloor |S|/2 \rfloor$, it is the median.

Here’s a divide-and-conquer approach to selection. For any number $v$, imagine splitting list $S$ into three categories: elements smaller than $v$, those equal to $v$ (there might be duplicates), and those greater than $v$. Call these $S_L$, $S_v$, and $S_R$ respectively. For instance, if the array

$$S : \begin{array}{ccccccccccc} 2 & 36 & 5 & 21 & 8 & 13 & 11 & 20 & 5 & 4 & 1 \end{array}$$

is split on $v = 5$, the three subarrays generated are

$$S_L : \begin{array}{cccc} 2 & 4 & 1 \end{array} \quad S_v : \begin{array}{cc} 5 & 5 \end{array} \quad S_R : \begin{array}{cccccc} 36 & 21 & 8 & 13 & 11 & 20 \end{array}$$

It is easy to hone the search down to one of these sublists. If we want, say, the eighth-smallest element of $S$, we know it must be the third-smallest element of $S_R$ since $|S_L| + |S_v| = 5$. That is, $\text{select}(S, 8) = \text{select}(S_R, 3)$. More generally, by checking $k$ against the sizes of the subarrays, we can quickly determine which of them holds the desired element:

$$\text{select}(S, k) = \begin{cases} \text{select}(S_L, k) & \text{if } k \leq |S_L| \\ v & \text{if } |S_L| < k \leq |S_L| + |S_v| \\ \text{select}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

The three sublists $S_L, S_v, S_R$ can be computed from $S$ in linear time. As an aside, notice that this computation can be done in place – that is, without allocating new memory for them – using three pointers (can you figure out this neat trick?). Once these sublists are obtained, exactly one of the three scenarios can hold, and so the original array $S$ effectively shrinks to one of size at most $\max\{|S_L|, |S_R|\}$.

The choice of $v$ is crucial. It should be picked quickly, and it should shrink the array substantially, the ideal situation being $|S_L| \approx |S_R| \approx \frac{1}{2}|S|$. If we could always guarantee this situation, we would get a running time of

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

which is linear as desired. But this requires picking $v$ to be the median, which is what we are trying to do in the first place! Instead, our method of choosing $v$ is much simpler: we just pick it randomly from $S$. 

48
Naturally, the running time of our algorithm depends on the random choices of \( v \). It is possible that due to persistent bad luck we keep picking \( v \) to be the largest element of the array (or the smallest element), and thereby shrink the array by only one element each time. In the example above, we might first pick \( v = 36 \), then \( v = 21 \), and so on. This worst case scenario would force our selection algorithm to perform

\[
n + (n - 1) + (n - 2) + \cdots + n/2 = \Theta(n^2)
\]

operations, but it is extremely unlikely to occur. Equally unlikely is the best possible case we discussed above, in which each randomly chosen \( v \) just happens to split the array perfectly in half, resulting in a running time of \( O(n) \). Where, in this spectrum from \( O(n) \) to \( \Theta(n^2) \), does the average running time lie? Fortunately, it lies very close to the best-case time.

To distinguish between lucky and unlucky choices of \( v \), we will call \( v \) good if it lies anywhere in 25th to 75th percentile of the array that it is chosen from. We like these choices of \( v \) because they ensure that the sublists \( S_L, S_R \) have size at most 3/4 that of \( S \) (do you see why?), so that the array shrinks substantially. Fortunately, good \( v \)'s are abundant: half the elements of any list must fall between the 25th to 75th percentile!

Given that a randomly chosen \( v \) has a 50% chance of being good, how many \( v \)'s do we need to pick on average before getting a good one? Here’s a more familiar reformulation:

**Lemma** On average a fair coin needs to be tossed two times before a “heads” is seen.

**Proof.** Let \( E \) be the expected number of tosses before a heads is seen. We certainly need at least one toss, and if it’s heads, we’re done. If it’s tails (which occurs with probability 1/2), we need to repeat. Hence \( E = 1 + \frac{1}{2}E \), which works out to \( E = 2 \).

Therefore, after two split operations on average, the array will shrink to at most 3/4 of its size. Letting \( T(n) \) be the expected running time on an array of size \( n \), we get

\[
T(n) \leq T(3n/4) + O(n),
\]

so \( T(n) \) is linear. On any input, our algorithm returns the correct answer in an average of \( O(n) \) steps. In conclusion, one does not need to sort an array to find its median: there is a faster, linear way. (And, incidentally, sorting does require \( n \log n \) comparisons, see the box.)

### 6 The fast Fourier transform

We have seen so far how divide-and-conquer leads to faster algorithms for multiplying integers and matrices; our next target is polynomials. Given two degree-\( d \) polynomials \( A(x) = a_0 + a_1 x + \cdots + a_d x^d \) and \( B(x) = b_0 + b_1 x + \cdots + b_d x^d \) we want to compute their product, the degree-2\( d \) polynomial \( C(x) = A(x) \cdot B(x) = c_0 + c_1 x + \cdots + c_{2d} x^{2d} \), whose order-\( k \) coefficient is

\[
c_k = a_0 b_k + a_1 b_{k-1} + \cdots + a_k b_0 = \sum_{i=0}^{k} a_i b_{k-i}
\]

(for \( i > d \), define \( a_i \) and \( b_i \) to be zero). Notice that each \( c_k \) takes \( O(k) \) steps to compute, and adding these up for all \( 2d + 1 \) coefficients of \( C(x) \) we get a baseline running time of \( O(d^2) \). Can we multiply polynomials faster than this?
The Unix sort command

Comparing the algorithms for sorting and median-finding we notice that, beyond the common divide-and-conquer philosophy and structure, they are exact opposites. Mergesort splits the array in two in the most convenient way (first half, second half), without any regard to the magnitudes of the elements in each half; but then it works hard to put the sorted subarrays together. In contrast, the median algorithm is careful about its splitting (smaller numbers first, then the larger ones), but its work ends with the recursive call. Quicksort is a sorting algorithm that splits the array in exactly the same way as our median algorithm; and once the subarrays are sorted, by two recursive calls, there is nothing more to do. Its worst-case performance is $\Theta(n^2)$, like that of median-finding. But its average case is $O(n \log n)$, and empirically it outperforms most other sorting algorithms. This has made quicksort a favorite in many applications – for instance, it is the basis of the code by which really large files are sorted.

6.1 An alternative representation of polynomials

To arrive at our fast algorithm for polynomial multiplication we take a radically different approach, suggested by an important property of polynomials.

Fact A degree-$d$ polynomial is uniquely characterized by its values at any $d + 1$ distinct points.

A familiar instance of this is that “any two points determine a line.” We will later see why the more general statement is true, but for the time being it gives us an alternative representation of polynomials. Fix any distinct points $x_0, \ldots, x_d$. We can specify a degree-$d$ polynomial $A(x) = a_0 + a_1 x + \cdots + a_d x^d$ either by

1. its coefficients $a_0, a_1, \ldots, a_d$; or
2. the values $A(x_0), A(x_1), \ldots, A(x_d)$.

Of these two representations, the second is the more attractive because it makes polynomial multiplication trivial. Since the product $C(x)$ has degree $2d$, we only need its value at any $\geq 2d + 1$ points. And its value at any given point $x = x_0$ is easy enough to figure out, just $A(x_0)$ times $B(x_0)$. Thus polynomial multiplication takes linear time in the value representation.

The problem is that we expect the input polynomials, and also their product, to be specified by coefficients. So we need to first translate from coefficients to values – which is just a matter of evaluating the polynomial at the chosen points – then multiply in the value representation, and finally translate back to coefficients, a process called interpolation.

The resulting algorithm is shown in Figure 6.1.
Why multiply polynomials?

Weren’t three years of high school algebra enough? It turns out that the fastest algorithms we have for multiplying integers rely heavily on polynomial multiplication; after all, polynomials and binary integers are quite similar — just replace \( x \) by the base 2, and watch out for carries. But perhaps more importantly, multiplying polynomials is crucial for signal processing. And the technique we shall develop for polynomial multiplication, the fast Fourier transform, has revolutionized — indeed, defined — this field of engineering.

Recall that a signal is any quantity which is a function of time or of position (Figure (a)). It might, for instance, capture a human voice by measuring fluctuations in air pressure close to the speaker’s mouth; or alternatively, the pattern of stars in the night sky, by measuring brightness as a function of angle. In order to extract information from the signal, we need to first digitize it by sampling (Figure (b)) — and, then, to put it through a system which will transform it in some way. The output is called the response of the system.

![Diagram](image.png)

An important class of systems are those that are linear — the response to the sum of two signals is just the sum of their individual responses — and time invariant — shifting the input signal by time \( t \) produces the same output, also shifted by \( t \). Any system with these properties is completely characterized by its response to the simplest possible input signal: the unit impulse \( \delta(t) \), consisting solely of a “jerk” at \( t = 0 \) (Figure (c)). To see this, first consider the close relative \( \delta(t - i) \), a shifted impulse in which the jerk occurs at time \( i \). Any signal \( a(t) \) can be expressed as a linear combination of these, letting \( \delta(t - i) \) pick out its behavior at time \( i \),

\[
a(t) = \sum_{i=0}^{T-1} a(i) \delta(t - i),
\]

where we assume that the signal consists of \( T \) samples. By linearity, the system response to input \( a(t) \) is determined by the responses to the various \( \delta(t - i) \). And by time invariance, these are in turn just shifted copies of the impulse response \( b(t) \), the response to \( \delta(t) \). In other words, the output of the system at time \( k \) is

\[
c(k) = \sum_{i=0}^{k} a(i) b(k - i),
\]

exactly the formula in equation (1) for polynomial multiplication!
Figure (d) shows the example of a system with impulse response

\[ b(t) = \begin{cases} 
1/s & \text{if } t = 0, 1, \ldots, s-1 \\
0 & \text{otherwise}
\end{cases} \]

Can you see what it computes? An average over a window of size \( s \),

\[ c(t) = \frac{1}{s} (a(t) + a(t-1) + \cdots + a(t-s+1)) \].

The equivalence of the two polynomial representations makes it clear that this high-level approach is correct, but how efficient is it? Certainly the selection step and the \( n \) multiplications are no trouble at all, just linear time.\(^2\) But (leaving aside interpolation, about which we know even less...) how about evaluation? Evaluating a polynomial of degree \( d \leq n \) at one point takes \( O(n) \) steps (can you check this?), and so the baseline for \( n \) points is \( O(n^2) \). We'll now see that the fast Fourier transform (FFT) does it in just \( O(n \log n) \) time, for a particular setting for polynomial multiplication, the coefficients of the polynomials are real numbers, and moreover are small enough that basic arithmetic operations (adding and multiplying) take unit time. We will assume this to be the case without any great loss of generality; in particular, the time bounds we obtain are easily adjustable to situations with larger numbers.

\(^2\)In a typical setting for polynomial multiplication, the coefficients of the polynomials are real numbers, and moreover are small enough that basic arithmetic operations (adding and multiplying) take unit time. We will assume this to be the case without any great loss of generality; in particular, the time bounds we obtain are easily adjustable to situations with larger numbers.

**Figure 6.1 Polynomial multiplication**

Input: Coefficients of two polynomials \( A(x), B(x) \), of degree \( d \)

Output: Their product \( C = A \cdot B \)

**Selection**
- Pick some points \( x_0, x_1, \ldots, x_{n-1} \), where \( n \geq 2d + 1 \)

**Evaluation**
- Compute \( A(x_0), A(x_1), \ldots, A(x_{n-1}) \) and \( B(x_0), B(x_1), \ldots, B(x_{n-1}) \)

**Multiplication**
- Compute \( C(x_k) = A(x_k)B(x_k) \) for all \( k = 0, \ldots, n-1 \)

**Interpolation**
- Recover \( C(x) = c_0 + c_1 x + \cdots + c_{2d} x^{2d} \)
larly clever choice of \(x_0, \ldots, x_{n-1}\) in which the computations required by the individual points overlap with one another and can be shared.

### 6.2 Evaluation by divide-and-conquer

Here’s an idea for how to pick the \(n\) points at which to evaluate a polynomial \(A(x)\) of degree \(\leq n - 1\). If we choose them to be positive-negative pairs, that is,

\[
\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1},
\]

then the computations required for each \(A(x_i)\) and \(A(-x_i)\) overlap a lot, because the even powers of \(x_i\) coincide with those of \(-x_i\).

To investigate this, we need to split \(A(x)\) into its odd and even powers, for instance

\[
3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4).
\]

Notice that the terms in parentheses are polynomials in \(x^2\). More generally,

\[
A(x) = A_e(x^2) + xA_o(x^2),
\]

where \(A_e(\cdot)\), with the even-numbered coefficients, and \(A_o(\cdot)\), with the odd-numbered coefficients, are polynomials of degree \(\leq n/2 - 1\) (assume for convenience that \(n\) is even). Given paired points \(\pm x_i\), the calculations needed for \(A(x_i)\) can be recycled towards computing \(A(-x_i)\):

\[
A(x_i) = A_e(x_i^2) + x_iA_o(x_i^2) \\
A(-x_i) = A_e(x_i^2) - x_iA_o(x_i^2).
\]

In other words, evaluating \(A(x)\) at \(n\) paired points \(\pm x_0, \ldots, \pm x_{n/2-1}\) reduces to evaluating \(A_e(x)\) and \(A_o(x)\) (which each have half the degree of \(A(x)\)) at just \(n/2\) points, \(x_0^2, \ldots, x_{n/2-1}^2\).

<table>
<thead>
<tr>
<th>Evaluate:</th>
<th>(A(x)) degree (\leq n - 1) at:</th>
<th>(+x_0)</th>
<th>(-x_0)</th>
<th>(+x_1)</th>
<th>(-x_1)</th>
<th>(\cdots)</th>
<th>(+x_{n/2-1})</th>
<th>(-x_{n/2-1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalently, evaluate:</td>
<td>(A_e(x)) and (A_o(x)) degree (\leq n/2 - 1) at:</td>
<td>(x_0^2)</td>
<td>(x_1^2)</td>
<td>(\cdots)</td>
<td>(x_{n/2-1}^2)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The original problem of size \(n\) is in this way recast as two subproblems of size \(n/2\), followed by some linear time arithmetic. If we could recurse, we would get a divide-and-conquer procedure with running time

\[
T(n) = 2T(n/2) + O(n),
\]

which is \(O(n \log n)\), exactly what we want.

But we have a problem: The plus-minus trick only works at the top level of the recursion. To recurse at the next level, we will need the \(n/2\) evaluation points \(x_0^2, x_1^2, \ldots, x_{n/2-1}^2\) to be
themselves plus-minus pairs. But how can a square be negative? The task seems impossible! Unless, of course, we use complex numbers.

Fine, but which complex numbers? To figure this out, let us “reverse engineer” the process. At the very bottom of the recursion, we have a single point. This point might as well be 1, in which case the level above it must consist of its square roots, $\pm \sqrt{1} = \pm 1$.

\[
\begin{array}{cccc}
\vdots \\
\pm 1 & -1 \\
\pm i & -i \\
+1 & -1 \\
+1 & \\
\end{array}
\]

The next level up then has $\pm \sqrt{\pm 1} = \pm 1$ as well as the complex numbers $\pm \sqrt{-1} = \pm i$, where $i$ is the imaginary unit. By continuing in this manner, we eventually reach the initial set of $n$ points. Perhaps you have already guessed what they are: the complex $n^{th}$ roots of unity, that is, the $n$ complex solutions to the equation $z^n = 1$.

Figure 6.2 is a pictorial review of some basic facts about complex numbers. The third panel of this figure introduces the $n^{th}$ roots of unity: the complex numbers $1, \omega, \omega^2, \ldots, \omega^{n-1}$, where $\omega = e^{2\pi i/n}$ (in polar coordinates, $(1, 2\pi/n)$). If $n$ is even,

1. the $n^{th}$ roots are plus-minus paired, $\omega^{n/2+j} = -\omega^j$; and

2. squaring them produces the $(n/2)^{nd}$ roots of unity.

Therefore, if we start with these numbers for some $n$ which is a power of two, then at successive levels of recursion we will have the $(n/2^k)^{th}$ roots of unity, for $k = 0, 1, 2, 3, \ldots$. All these sets of numbers are plus-minus paired, and so our divide-and-conquer, as shown in the last panel, works perfectly. The resulting algorithm is the fast Fourier transform (Figure 6.3).
Figure 6.2 The complex roots of unity are ideal for our divide-and-conquer scheme.

**The complex plane**

\[ z = a + bi \] is plotted at position \((a, b)\).

Polar coordinates: rewrite as \[ z = r \cos \theta + i \sin \theta = re^{i\theta}, \] denoted \((r, \theta)\).

- **length** \( r = \sqrt{a^2 + b^2} \),
- **angle** \( \theta \in [0, 2\pi) \): \( \cos \theta = a/r, \sin \theta = b/r \)
- \( \theta \) can always be reduced modulo \( 2\pi \).

Examples:

<table>
<thead>
<tr>
<th>Number Polar coords</th>
<th>-1</th>
<th>( i )</th>
<th>( 5 + 5i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>((1, \pi))</td>
<td>((1, \pi/2))</td>
<td>((5\sqrt{2}, \pi/4))</td>
</tr>
</tbody>
</table>

**Multiplying is easy in polar coordinates**

Multiply the lengths and add the angles:

\[ (r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2). \]

For any \( z = (r, \theta) \),
- \(-z = (r, \theta + \pi) \) since \(-1 = (1, \pi)\);
- if \( z \) is on the **unit circle** (i.e. \( r = 1 \)), then \( z^n = (1, n\theta) \).

**The \( n^{th} \) complex roots of unity**

Solutions to the equation \( z^n = 1 \).

By the multiplication rule: solutions are \( z = (1, \theta) \), for \( \theta \) a multiple of \( 2\pi/n \). (Shown here for \( n = 16 \)).

For even \( n \):
- These numbers are **plus-minus paired**: \(-1, \theta = (1, \theta + \pi)\).
- Their squares are the \( (n/2)^{nd} \) roots of unity, shown here with boxes around them.

**Divide-and-conquer step**

Evaluate \( A(x) \) at \( n^{th} \) roots of unity

\[
\begin{align*}
\text{Evaluate } A(x) \text{ at } n^{th} \text{ roots of unity} & \quad \text{divide and conquer} \quad \text{Evaluate } A(x) \text{ at } (n/2)^{nd} \text{ roots} \\
\text{paired} & \quad \text{still paired} \\
(n \text{ is a power of } 2) & \quad (n \text{ is a power of } 2)
\end{align*}
\]
function FFT(A, ω)
Input: Coefficient representation of a polynomial A(x)
of degree ≤ n − 1, where n is a power of two
ω, an n<sup>th</sup> root of unity
Output: Value representation A(ω<sup>0</sup>),...,A(ω<sup>n−1</sup>)

if ω = 1 then return A(1)
express A(x) in the form A<sub>e</sub>(x<sup>2</sup>) + xA<sub>o</sub>(x<sup>2</sup>)
call FFT(A<sub>e</sub>,ω<sup>2</sup>) to evaluate A<sub>e</sub> at even powers of ω
call FFT(A<sub>o</sub>,ω<sup>2</sup>) to evaluate A<sub>o</sub> at even powers of ω
for j = 0 to n−1:
compute A(ω<sup>j</sup>) = A<sub>e</sub>(ω<sup>2j</sup>) + ω<sup>j</sup>A<sub>o</sub>(ω<sup>2j</sup>)
return A(ω<sup>0</sup>),...,A(ω<sup>n−1</sup>)

### 6.3 Interpolation

Let’s take stock of where we are. We developed a high-level scheme for multiplying polynomials (Figure 6.1), based on the observation that polynomials can be represented in two ways, in terms of their coefficients or in terms of their values at a selected set of points.

The value representation makes it trivial to multiply polynomials, but we cannot ignore the coefficient representation since it is the form in which the input and output of our overall algorithm are specified.

So we designed the FFT, a way to move from coefficients to values in time just O(n log n), when the points {x<sub>i</sub>} are complex roots of unity.

\[
\langle \text{values} \rangle = \text{FFT}(\langle \text{coefficients} \rangle, \omega).
\]

The last remaining piece of the puzzle is the inverse operation, interpolation. It will turn out, amazingly, that

\[
\langle \text{coefficients} \rangle = \frac{1}{n} \text{FFT}(\langle \text{values} \rangle, \omega^{-1}).
\]

Interpolation is thus solved in the most simple and elegant way we could possibly have hoped for — using the same FFT algorithm, but called with ω<sup>-1</sup> in place of ω! This might seem like a miraculous coincidence, but it will make a lot more sense when we recast our polynomial operations in the language of linear algebra. Meanwhile, our O(n log n) polynomial multiplication algorithm (Figure 6.1) is now fully specified.
A matrix reformulation

To get a clearer view of interpolation, let’s explicitly set down the relationship between our two representations for a polynomial \( A(x) \) of degree \( \leq n - 1 \). They are both vectors of \( n \) numbers, and one is a linear transformation of the other:

\[
\begin{bmatrix}
A(x_0) \\
A(x_1) \\
\vdots \\
A(x_{n-1})
\end{bmatrix} =
\begin{bmatrix}
1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\
1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1}
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
\vdots \\
a_{n-1}
\end{bmatrix}.
\]

Call the matrix in the middle \( M \). Its specialized format – a Vandermonde matrix – gives it many remarkable properties, of which the following (Exercise) is particularly relevant to us.

If \( x_0, \ldots, x_{n-1} \) are distinct numbers then \( M \) is invertible.

The existence of \( M^{-1} \) allows us to invert the matrix equation above, so as to express coefficients in terms of values. In brief,

Evaluation is multiplication by \( M \), while interpolation is multiplication by \( M^{-1} \).

This reformulation of our polynomial operations reveals their essential nature more clearly. Among other things, it finally justifies an assumption we have been making throughout, that \( A(x) \) is uniquely characterized by its values at any \( n \) points – in fact, we now have an explicit formula which will give us \( A(x) \) in this situation. Vandermonde matrices also have the distinction of being quicker to invert than more general matrices, in \( O(n^2) \) time instead of \( O(n^3) \). However, using this for interpolation would still not be fast enough for us, so once again we turn to our special choice of points – the complex roots of unity.

Interpolation resolved

In linear algebra terms, the FFT multiplies an arbitrary \( n \)-dimensional vector – which we were calling the coefficient representation – by the \( n \times n \) matrix

\[
M_n(\omega) =
\begin{bmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\
1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^j & \omega^{2j} & \cdots & \omega^{(n-1)j} \\
1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)}
\end{bmatrix}
\]

where \( \omega \) is any complex \( n^{th} \) root of unity, and \( n \) is a power of two. Notice how simple this matrix is to describe: its \((j,k)^{th}\) entry (starting row- and column-count at zero) is \( \omega^{jk} \).

Multiplication by \( M = M_n(\omega) \) maps the \( k^{th} \) coordinate axis (more specifically, the vector with all zeros except for a one at position \( k \)) onto the \( k^{th} \) column of \( M \). Now here’s the crucial observation, which we’ll prove shortly: the columns of \( M \) are orthogonal (at right angles) to each other. Therefore they can be thought of as the axes of an alternative coordinate system,
Inversion transformation

When we write out the orthogonality condition precisely, we will be able to read off this inverse transformation with ease:

**Inversion formula** \( M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1}) \).

But \( \omega^{-1} \) is also an \( n^{th} \) root of unity, and so interpolation – or equivalently, multiplication by \( M_n(\omega)^{-1} \) – is itself just an FFT operation, but with \( \omega \) replaced by \( \omega^{-1} \).

Now let’s get into the details. Take \( \omega \) to be \( e^{2\pi i/n} \) for convenience, and think of the columns of \( M \) as vectors in \( \mathbb{C}^n \). Recall that the **angle** between two vectors \( u = (u_0, \ldots, u_{n-1}) \) and \( v = (v_0, \ldots, v_{n-1}) \) in \( \mathbb{C}^n \) is just a scaling factor times their **inner product**

\[
    u \cdot v^* = u_0 v_0^* + u_1 v_1^* + \cdots + u_{n-1} v_{n-1}^*,
\]

where \( z^* \) denotes the complex conjugate\(^3\) of \( z \). This quantity is maximized when the vectors lie in the same direction, and is zero when the vectors are orthogonal to each other.

The fundamental observation we need is that

**Lemma** The columns of matrix \( M \) are orthogonal to each other.

**Proof.** Take the inner product of any columns \( j \) and \( k \) of matrix \( M \),

\[
1 + \omega^{j-k} + \omega^{2(j-k)} + \cdots + \omega^{(n-1)(j-k)}.
\]

This is a geometric series with first term 1, last term \( \omega^{(n-1)(j-k)} \) and ratio \( \omega^{(j-k)} \). Therefore it evaluates to \( (1 - \omega^{n(j-k)})/(1 - \omega^{(j-k)}) \), which is 0 — except when \( j = k \), in which case all terms are 1 and the sum is \( n \). \( \square \)

\(^3\)For a single complex number \( z = re^{i\theta} \), its **complex conjugate** \( z^* \) is \( re^{-i\theta} \). The complex conjugate of a vector (or matrix) is obtained by taking the complex conjugates of all of its entries.