Ceterum in problematis natura fundatum est, ut methodi quaecunque *continuo prolixiores evadant, quo maiores sunt numeri, ad quos applicantur.*
[It is in the nature of the problem that any *method will become more prolix as the numbers to which it is applied grow larger.]*

— Carl Friedrich Gauß, *Disquisitiones Arithmeticae* (1801)
English translation by A.A. Clarke (1965)

Illam vero methodum calculi mechanici taedium magis minuere, praxis tentantem docebit. [Truly, that method greatly reduces the tedium of mechanical calculations; practice will teach whoever tries it.]

— Carl Friedrich Gauß, "Theoria interpolationis methodo nova tractata" (c. 1805)

After much deliberation, the distinguished members of the international committee decided unanimously (when the Russian members went out for a caviar break) that since the Chinese emperor invented the method before anybody else had even been born, the method should be named after him. The Chinese emperor's name was Fast, so the method was called the Fast Fourier Transform.

— Thomas S. Huang, "How the fast Fourier transform got its name" (1971)

A

Fast Fourier Transforms

[Read Chapters 0 and 1 first.]
Status: Beta

A.1 Polynomials

Polynomials are functions of one variable built from additions, subtractions, and multiplications (but no divisions). The most common representation for a polynomial $p(x)$ is as a sum of weighted powers of the variable x:

$$p(x) = \sum_{j=0}^{n} a_j x^j.$$

The numbers a_j are called the *coefficients* of the polynomial. The *degree* of the polynomial is the largest power of x whose coefficient is not equal to zero; in the example above, the degree is *at most* n. Any polynomial of degree n can be represented by an array $P[0..n]$ of $n+1$ coefficients, where $P[j]$ is the coefficient of the x^j term, and where $P[n] \neq 0$.

Here are three of the most common operations performed on polynomials:

- **Evaluate:** Give a polynomial p and a number x, compute the number $p(x)$.
- **Add:** Give two polynomials p and q, compute a polynomial $r = p + q$, so that $r(x) = p(x) + q(x)$ for all x. If p and q both have degree n, then their sum $p + q$ also has degree n.
- **Multiply:** Give two polynomials p and q, compute a polynomial $r = p \cdot q$, so that $r(x) = p(x) \cdot q(x)$ for all x. If p and q both have degree n, then their product $p \cdot q$ has degree $2n$.

We learned simple algorithms for all three of these operations in high-school algebra. The addition and multiplication algorithms are straightforward generalizations of the standard algorithms for integer arithmetic.

```
EVALUATE(P[0..n], x):
  X ← 1    ⟨⟨X = x^j⟩⟩
  y ← 0
  for j ← 0 to n
      y ← y + P[j]·X
      X ← X·x
  return y
```

```
ADD(P[0..n], Q[0..n]):
  for j ← 0 to n
      R[j] ← P[j] + Q[j]
  return R[0..n]
```

```
MULTIPLY(P[0..n], Q[0..m]):
  for j ← 0 to n+m
      R[j] ← 0
  for j ← 0 to n
      for k ← 0 to m
          R[j+k] ← R[j+k] + P[j]·Q[k]
  return R[0..n+m]
```

EVALUATE uses $O(n)$ arithmetic operations.[1] This is the best we can hope for, but we can cut the number of multiplications in half using *Horner's rule*:

$$p(x) = a_0 + x(a_1 + x(a_2 + \ldots + xa_n)).$$

```
HORNER(P[0..n], x):
  y ← P[n]
  for i ← n−1 down to 0
      y ← x·y + P[i]
  return y
```

The addition algorithm also runs in $O(n)$ time, and this is clearly the best we can do.

[1] All time analysis in this lecture assumes that each arithmetic operation takes $O(1)$ time. This may not be true in practice; in fact, one of the most powerful applications of fast Fourier transforms is fast *integer* multiplication. The fastest algorithm currently known for multiplying two n-bit integers, published by David Harvey and Joris van der Hoeven in 2019, uses $O(n \log n)$ bit operations and is based on fast Fourier transforms.

The multiplication algorithm, however, runs in $O(n^2)$ time. In Chapter 1, we saw a divide-and-conquer algorithm (due to Karatsuba) for multiplying two n-bit integers in only $O(n^{\lg 3})$ steps; precisely the same approach can be applied here. Even cleverer divide-and-conquer strategies lead to multiplication algorithms whose running times are arbitrarily close to linear—$O(n^{1+\varepsilon})$ for your favorite value $e > 0$—but except for a few simple cases, these algorithms not worth the trouble in practice, thanks to large hidden constants.

A.2 Alternate Representations

Part of what makes multiplication so much harder than the other two operations is our input representation. Coefficient vectors are the most common representation for polynomials, but there are at least two other useful representations.

Roots

The Fundamental Theorem of Algebra states that every polynomial p of degree n has exactly n *roots* $r_1, r_2, \ldots r_n$ such that $p(r_j) = 0$ for all j. Some of these roots may be irrational; some of these roots may by complex; and some of these roots may be repeated. Despite these complications, this theorem implies a unique representation of any polynomial of the form

$$p(x) = s \prod_{j=1}^{n} (x - r_j)$$

where the r_j's are the roots and s is a scale factor. Once again, to represent a polynomial of degree n, we need a list of $n + 1$ numbers: one scale factor and n roots.

Given a polynomial in this root representation, we can clearly evaluate it in $O(n)$ time. Given two polynomials in root representation, we can multiply them in $O(n)$ time by multiplying their scale factors and concatenating the two root sequences.

Unfortunately, if we want to add two polynomials in root representation, we're out of luck. There's essentially *no* correlation between the roots of p, the roots of q, and the roots of $p + q$. We could convert the polynomials to the more familiar coefficient representation first—this takes $O(n^2)$ time using the high-school algorithms—but there's no easy way to convert the answer back. In fact, for most polynomials of degree 5 or more in coefficient form, it's *impossible* to compute roots exactly.[2]

Samples

Our third representation for polynomials comes from a different consequence of the Fundamental Theorem of Algebra. Given a list of $n + 1$ pairs $\{(x_0, y_0), (x_1, y_1), \ldots,$

[2]This is where numerical analysis comes from.

$(x_n, y_n)\}$, there is *exactly one* polynomial p of degree n such that $p(x_j) = y_j$ for all j. This is a natural generalization of the fact that any two points determine a unique line, because a line is the graph of a polynomial of degree 1. We say that the polynomial p ***interpolates*** the points (x_j, y_j). As long as we agree on the sample locations x_j in advance, we once again need exactly $n+1$ numbers to represent a polynomial of degree n.

Adding or multiplying two polynomials in this sample representation is straightforward, as long as they use the same sample locations x_j. To add the polynomials, add their sample values. To multiply two polynomials, multiply their sample values; however, if we're multiplying two polynomials of degree n, we must *start* with $2n + 1$ sample values for each polynomial, because that's how many we need to uniquely represent their product. Both algorithms run in $O(n)$ time.

Unfortunately, evaluating a polynomial in this representation is no longer straightforward. The following formula, due to Lagrange, allows us to compute the value of any polynomial of degree n at any point, given a set of $n + 1$ samples.

$$p(x) = \sum_{j=0}^{n-1} \left(\frac{y_j}{\prod_{k \neq j}(x_j - x_k)} \prod_{k \neq j} (x - x_k) \right)$$

Hopefully it's clear that formula actually describes a polynomial function of x, since each term in the sum is a scaled product of monomials. It's also not hard to verify that $p(x_j) = y_j$ for every index j; most of the terms of the sum vanish. As I mentioned earlier, the Fundamental Theorem of Algebra implies that p is *the only* polynomial that interpolates the points $\{(x_j, y_j)\}$. Lagrange's formula can be translated mechanically into an $O(n^2)$-time algorithm.

Summary

We find ourselves in the following frustrating situation. We have three representations for polynomials and three basic operations. Each representation allows us to almost trivially perform a different pair of operations in linear time, but the third takes at least quadratic time, if it can be done at all!

representation	evaluate	add	multiply
coefficients	$O(n)$	$O(n)$	$\mathbf{O(n^2)}$
roots + scale	$O(n)$	$\boldsymbol{\infty}$	$O(n)$
samples	$\mathbf{O(n^2)}$	$O(n)$	$O(n)$

A.3 Converting Between Representations

What we need are fast algorithms to convert quickly from one representation to another. Then if we need to perform an operation that's hard for our default representation, we

can switch to a different representation that makes the operation easy, perform the desired operation, and then switch back. This strategy immediately rules out the root representation, since (as I mentioned earlier) finding roots of polynomials is impossible in general, at least if we're interested in exact results.

So how do we convert from coefficients to samples and back? Clearly, once we choose our sample positions x_j, we can compute each sample value $y_j = p(x_j)$ in $O(n)$ time from the coefficients using Horner's rule. So we can convert a polynomial of degree n from coefficients to samples in $O(n^2)$ time. Lagrange's formula can be used to convert the sample representation back to the more familiar coefficient form. If we use the naïve algorithms for adding and multiplying polynomials (in coefficient form), this conversion takes $O(n^3)$ time.

We can improve the cubic running time by observing that *both* conversion problems boil down to computing the product of a matrix and a vector. The explanation will be slightly simpler if we assume the polynomial has degree $n-1$, so that n is the number of coefficients or samples. Fix a sequence $x_0, x_1, \ldots, x_{n-1}$ of sample *positions*, and let V be the $n \times n$ matrix where $v_{ij} = x_i^j$ (indexing rows and columns from 0 to $n-1$):

$$V = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix}.$$

The matrix V is called a ***Vandermonde matrix***. The vector of coefficients $\vec{a} = (a_0, a_1, \ldots, a_{n-1})$ and the vector of sample *values* $\vec{y} = (y_0, y_1, \ldots, y_{n-1})$ are related by the matrix equation

$$V\vec{a} = \vec{y},$$

or in more detail,

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}.$$

Given this formulation, we can clearly transform any coefficient vector $\vec{a}$ into the corresponding sample vector $\vec{y}$ in $O(n^2)$ time.

Conversely, if we know the sample values $\vec{y}$, we can recover the coefficients by solving a system of n linear equations in n unknowns, which can be done in $O(n^3)$ time using

Gaussian elimination.[3] But we can speed this up by implicitly hard-coding the sample positions into the algorithm, To convert from samples to coefficients, we can multiply the sample vector by the inverse of V, again in $O(n^2)$ time:

$$\vec{a} = V^{-1}\vec{y}.$$

Computing V^{-1} would take $O(n^3)$ time if we had to do it from scratch using Gaussian elimination, but because we fixed the set of sample positions in advance, the matrix V^{-1} can be hard-coded directly into the algorithm.[4]

So we can convert from coefficients to samples and back in $O(n^2)$ time. At first glance, this result seems pointless; we can already add, multiply, or evaluate directly in either representation in $O(n^2)$ time, so why bother? But there's a degree of freedom we haven't exploited yet: ***We get to choose the sample positions!*** Our conversion algorithm is slow only because we're trying to be too general. If we choose a set of sample positions with the right recursive structure, we can perform this conversion more quickly.

A.4 Divide and Conquer

Any polynomial of degree at most $n-1$ can be expressed as a combination of two polynomials of degree at most $(n/2)-1$ as follows:

$$p(x) = p_{\text{even}}(x^2) + x \cdot p_{\text{odd}}(x^2).$$

The coefficients of p_{even} are precisely the even-degree coefficients of p, and the coefficients of p_{odd} are precisely the odd-degree coefficients of p. Thus, we can evaluate $p(x)$ by recursively evaluating $p_{\text{even}}(x^2)$ and $p_{\text{odd}}(x^2)$ and performing $O(1)$ additional arithmetic operations.

Now call a set X of n values ***collapsing*** if either of the following conditions holds:

- X has one element.
- The set $X^2 = \left\{x^2 \;\middle|\; x \in X\right\}$ has exactly $n/2$ elements and is (recursively) collapsing.

The size of a collapsing set must be a power of 2. Given a polynomial p of degree $n-1$, and a collapsing set X of size n, we can compute the set $\{p(x) \mid x \in X\}$ of sample values as follows:

1. Recursively compute the sample values $\left\{p_{\text{even}}(x^2) \;\middle|\; x \in X\right\} = \left\{p_{\text{even}}(\hat{x}) \;\middle|\; \hat{x} \in X^2\right\}$.
2. Recursively compute the sample values $\left\{p_{\text{odd}}(x^2) \;\middle|\; x \in X\right\} = \left\{p_{\text{odd}}(\hat{x}) \;\middle|\; \hat{x} \in X^2\right\}$.
3. For each $x \in X$, compute the sample value $p(x) = p_{\text{even}}(x^2) + x \cdot p_{\text{odd}}(x^2)$.

[3] In fact, Lagrange's formula is a special case of Cramer's rule for solving linear systems.

[4] Actually, it is possible to invert an $n \times n$ matrix in $o(n^3)$ time, using fast matrix multiplication algorithms that closely resemble Karatsuba's sub-quadratic divide-and-conquer algorithm for integer/polynomial multiplication. On the other hand, my numerical-analysis colleagues have reasonable cause to shoot me in the face for daring to suggest, even in passing, that anyone actually invert a matrix at all, ever.

The running time of this algorithm satisfies the familiar "mergesort" recurrence $T(n) = 2T(n/2) + \Theta(n)$, which as we all know solves to $T(n) = \Theta(n \log n)$.

Great! Now all we need is a sequence of arbitrarily large collapsing sets. The simplest method to construct such sets is to invert the recursive definition: If X is a collapsible set of size n *that does not contain the number* 0, then $\sqrt{X} = \{\pm\sqrt{x} \mid x \in X\}$ is a collapsible set of size $2n$. This observation gives us an infinite sequence of collapsible sets, starting as follows:[5]

$$X_1 := \{1\}$$
$$X_2 := \{1, -1\}$$
$$X_4 := \{1, -1, i, -i\}$$
$$X_8 := \left\{1, -1, i, -i, \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i, -\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i, \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i, -\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i\right\}$$

A.5 The Discrete Fourier Transform

For any n, the elements of X_n are called the ***complex nth roots of unity***; these are the roots of the polynomial $x^n - 1 = 0$. These n complex values are spaced exactly evenly around the unit circle in the complex plane. Every nth root of unity is a power of the *primitive* nth root

$$\boxed{\omega_n = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}.}$$

A typical nth root of unity has the form

$$\omega_n^k = e^{(2\pi i/n)k} = \cos\left(\frac{2\pi}{n}k\right) + i\sin\left(\frac{2\pi}{n}k\right).$$

These complex numbers have several useful properties for any integers n and k:

- There are exactly n different nth roots of unity: $\omega_n^k = \omega_n^{k \bmod n}$.
- If n is even, then $\omega_n^{k+n/2} = -\omega_n^k$; in particular, $\omega_n^{n/2} = -\omega_n^0 = -1$.
- $1/\omega_n^k = \omega_n^{-k} = \overline{\omega_n^k} = (\overline{\omega_n})^k$, where the bar represents complex conjugation: $\overline{a + bi} = a - bi$
- $\omega_n = \omega_{kn}^k$. Thus, every nth root of unity is also a (kn)th root of unity.

These properties imply immediately that if n is a power of 2, then the set of all nth roots of unity is collapsible!

[5] In this chapter, lower case italic i always represents the square root of -1. Computer scientists are used to thinking of i as an integer index into a sequence, an array, or a for-loop, but we obviously can't do that here. The engineers' habit of using $j = \sqrt{-1}$ just delays the problem—How do engineers write quaternions?—and typographical hacks like I or $\mathbf{i}$ or $\imath$ or Mathematica's $\mathring{\imath}$ are missing the point.

If we sample a polynomial of degree $n-1$ at the nth roots of unity, the resulting list of sample values is called the ***discrete Fourier transform*** of the polynomial (or more formally, of its coefficient vector). Thus, given an array $P[0..n-1]$ of coefficients, its discrete Fourier transform is the vector $P^*[0..n-1]$ defined as follows:

$$P^*[j] := p(\omega_n^j) = \sum_{k=0}^{n-1} P[k] \cdot \omega_n^{jk}$$

As we already observed, the fact that sets of roots of unity are collapsible implies that we can compute the discrete Fourier transform in $O(n \log n)$ time. The resulting algorithm, called the ***fast Fourier transform***, was popularized by Cooley and Tukey in 1965.[6] The algorithm assumes that n is a power of two; if necessary, we can pad the coefficient vector with at most n zeros.

```
RADIX2FFT(P[0..n-1]):
  if n = 1
      return P

  for j ← 0 to n/2-1
      U[j] ← P[2j]
      V[j] ← P[2j+1]

  U* ← RADIX2FFT(U[0..n/2-1])
  V* ← RADIX2FFT(V[0..n/2-1])

  ω_n ← cos(2π/n) + i sin(2π/n)
  ω ← 1

  for j ← 0 to n/2-1
      P*[j]       ← U*[j] + ω·V*[j]
      P*[j+n/2] ← U*[j] - ω·V*[j]
      ω ← ω·ω_n

  return P*[0..n-1]
```

Figure A.1. The Cooley-Tukey radix-2 fast Fourier transform algorithm.

Variants of this divide-and-conquer algorithm were previously described by Good in 1958, by Thomas in 1948, by Danielson and Lánczos in 1942, by Stumpf in 1937, by Yates in 1932, and by Runge in 1903; some special cases were published even earlier by Everett in 1860, by Smith in 1846, and by Carlini in 1828. But the algorithm, in its full modern recursive generality, was first described *and used* by Gauss around 1805 for calculating the periodic orbits of asteroids from a finite number of observations.

[6]Tukey apparently developed this algorithm to help detect Soviet nuclear tests without actually visiting Soviet nuclear facilities, by interpolating off-shore seismic readings. Without his rediscovery, the nuclear test ban treaty might never have been ratified, and we might all be speaking Russian, or more likely, whatever language radioactive glass speaks.

In fact, Gauss's recursive algorithm predates even Fourier's introduction of harmonic analysis by two years. So, of course, the algorithm is universally called the *Cooley-Tukey* algorithm. Gauss's work built on earlier research on trigonometric interpolation by Bernoulli, Lagrange, Clairaut, and Euler; in particular, the first explicit description of the discrete "Fourier" transform was published by Clairaut in 1754, more than half a century before Fourier's work. Alas, nobody will understand you if you talk about Gauss's fast Clairaut transform algorithms. Hooray for Stigler's Law![7]

A.6 More General Factoring

The algorithm in Figure A.1 is often called the ***radix-2 fast Fourier transform*** to distinguish it from other FFT algorithms. In fact, both Gauss and (much later) Cooley and Tukey described a more general divide-and-conquer strategy that does not assume n is a power of two, but can be applied to any composite order n. Specifically, if $n = pq$, we can decompose the discrete Fourier transform of order n into simpler discrete Fourier transforms as as follows. For all indices $0 \le a < p$ and $0 \le b < q$, we have

$$\begin{aligned}
x^*_{aq+b} &= \sum_{\ell=0}^{pq-1} x_{jp+k} \, (\omega_{pq}^{jp+k})^{\ell} \\
&= \sum_{k=0}^{p-1} \sum_{j=0}^{q-1} x_{jp+k} \, \omega_{pq}^{(aq+b)(jp+k)} \\
&= \sum_{k=0}^{p-1} \sum_{j=0}^{q-1} x_{jp+k} \, \omega_q^{bj} \, \omega_{pq}^{bk} \, \omega_p^{ak} \\
&= \sum_{k=0}^{p-1} \left(\left(\sum_{j=0}^{q-1} x_{jp+k} \, (\omega_q^b)^j \right) \omega_{pq}^{bk} \right) (\omega_p^a)^k,
\end{aligned}$$

[7] Lest anyone believe that Stigler's Law has treated Gauss unfairly, remember that "Gaussian elimination" was not discovered by Gauss; the algorithm was not even given that name until the mid-20th century! Elimination became the standard method for solving systems of linear equations in Europe in the early 1700s, when it appeared in Isaac Newton's influential textbook *Arithmetica universalis*.[8] Although Newton apparently (and perhaps even correctly) believed he had invented the elimination algorithm, it actually appears in several earlier works, including the eighth chapter of the Chinese manuscript *The Nine Chapters of the Mathematical Art*. The authors and precise age of the *Nine Chapters* are unknown, but commentary written by Liu Hui in 263CE claims that the text was already several centuries old. It was almost certainly *not* invented by a Chinese emperor named Fast.

[8] *Arithmetica universalis* was compiled from Newton's lecture notes and published over Newton's strenuous objections. He refused to have his name associated with the book, and he even considered buying up every copy of the first printing to destroy them. Apparently he didn't want anyone to think it was his latest research. The first edition crediting Newton as the author did not appear until 25 years after his death.

The innermost sum in this expression is one coefficient of a discrete Fourier transform of order q, and the outermost sum is one coefficient of a discrete Fourier transform of order q. The intermediate factors ω_{pq}^{bk} are now formally known as "twiddle factors". No, seriously, that's actually what they're called. This wall of symbols implies that the discrete Fourier transform of order n can be evaluated as follows:

1. Write the input vector into a $p \times q$ array in row-major order.
2. Apply a discrete Fourier transform of order p to each column of the 2d array.
3. Multiply each entry of the 2d array by the appropriate twiddle factor.
4. Apply a discrete Fourier transform of order q to each row of the 2d array.
5. Extract the output vector from the $p \times q$ array in column-major order.

The algorithm is described in more detail in Figure A.2.

```
FACTORFFT(P[0..pq−1]):
  ⟨⟨Copy/typecast to 2d array in row-major order⟩⟩
  for j ← 0 to p−1
      for k ← 0 to q−1
          A[j,k] ← P[jp+k]

  ⟨⟨Recursively apply order-p FFTs to columns⟩⟩
  for k ← 0 to q−1
      B[·,k] ← FFT(A[·,k])

  ⟨⟨Multiply by twiddle factors⟩⟩
  for j ← 0 to p−1
      for k ← 0 to q−1
          B[·,k] ← B[·,k] · ω_pq^jk

  ⟨⟨Recursively apply order-q FFTs to rows⟩⟩
  for j ← 0 to p−1
      C[j,·] ← FFT(C[j,·])

  ⟨⟨Copy/typecast to 1d array in column-major order⟩⟩
  for j ← 0 to p−1
      for k ← 0 to q−1
          P*[j+kq] ← C[j,k]

  return P*[0..pq−1]
```

Figure A.2. The Gauss-Cooley-Tukey FFT algorithm

We can recover the original radix-2 FFT algorithm in Figure A.1 by setting $p = n/2$ and $q = 2$. The lines $P^*[j] \leftarrow U^*[j] + \omega \cdot V^*[j]$ and $P^*[j+n/2] \leftarrow U^*[j] - \omega \cdot V^*[j]$ are applying an order-2 discrete Fourier transform; in particular, the multipliers ω and $-\omega$ are the "twiddle factors".

Both Gauss[9] and Cooley and Tukey recommended applying this factorization approach recursively. Cooley and Tukey observed further that if all prime factors of n are

[9]Gauss wrote: *"Nulla iam amplius explicatione opus erit, quomodo illa partitio adhuc ulterius extendi et ad eum casum applicari possit, ubi multitudo omnium valorum propositorum numerus e tribus pluribusve*

smaller than some constant, so that the subproblems at the leaves of the recursion tree can be solved in $O(1)$ time, the entire algorithm runs in only $O(n \log n)$ time.

Using a completely different approach, Charles Rader and Leo Bluestein described FFT algorithms that run in $O(n \log n)$ time when n is an arbitrary prime number, by reducing to two FFTs whose orders have only small prime factors. Combining these two approaches yields an $O(n \log n)$-time algorithm to compute discrete Fourier transforms of any order n.

A.7 Inverting the FFT

We also need to recover the coefficients of the product from the new sample values. Recall that the transformation from coefficients to sample values is *linear*; the sample vector is the product of a Vandermonde matrix V and the coefficient vector. For the discrete Fourier transform, each entry in V is an nth root of unity; specifically,

$$\boxed{v_{jk} = \omega_n^{jk}}$$

for all integers j and k. More explicitly:

$$V = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}$$

To invert the discrete Fourier transform, converting sample values back to coefficients, it suffices to multiply the vector P^* of sample values by the inverse matrix V^{-1}. The following amazing fact implies that this is almost the same as multiplying by V itself:

Lemma A.1. $V^{-1} = \overline{V}/n$

Proof: It suffices to show that $M = V\overline{V}$ is the identity matrix scaled by a factor of n. We can compute a single entry in M as follows:

$$m_{jk} = \sum_{l=0}^{n-1} \omega_n^{jl} \cdot \overline{\omega_n}^{lk} = \sum_{l=0}^{n-1} \omega_n^{jl-lk} = \sum_{l=0}^{n-1} (\omega_n^{j-k})^l$$

factoribus compositus est, e.g. si numerus μ rursus esset compositus, in quo casu manifesto quaevis periodus μ terminorum in plures periodos minores subdividi potest." ["There is no need to explain how that division can be extended further and applied to the case where most of the proposed values are composed of three or more factors, for example, if the number μ is again composite, in which case each period of μ terms can obviously be subdivided into several smaller periods."]

If $j = k$, then $\omega_n^{j-k} = \omega_n^0 = 1$, so

$$m_{jk} = \sum_{l=0}^{n-1} 1 = n,$$

and if $j \neq k$, we have a geometric series

$$m_{jk} = \sum_{l=0}^{n-1} (\omega_n^{j-k})^l = \frac{(\omega_n^{j-k})^n - 1}{\omega_n^{j-k} - 1} = \frac{(\omega_n^n)^{j-k} - 1}{\omega_n^{j-k} - 1} = \frac{1^{j-k} - 1}{\omega_n^{j-k} - 1} = 0. \qquad \square$$

In other words, if $W = V^{-1}$ then $w_{jk} = \overline{v_{jk}}/n = \overline{\omega_n^{jk}}/n = \omega_n^{-jk}/n$. What this observation implies for us computer scientists is that any algorithm for computing the discrete Fourier transform can be trivially adapted or modified to compute the inverse transform as well; see Figure A.3.

```
INVERSEFFT(P*[0..n-1]):
  P[0..n-1] ← FFT(P*)
  for j ← 0 to n-1
      P[j] ← conj(P[j])/n
  return P[0..n-1]
```

```
INVERSERADIX2FFT(P*[0..n-1]):
  if n = 1
      return P

  for j ← 0 to n/2-1
      U*[j] ← P*[2j]
      V*[j] ← P*[2j+1]

  U ← INVERSERADIX2FFT(U*[0..n/2-1])
  V ← INVERSERADIX2FFT(V*[0..n/2-1])

  ω_n ← cos(2π/n) - i sin(2π/n)
  ω ← 1

  for j ← 0 to n/2-1
      P[j]       ← (U[j] + ω·V[j])/2
      P[j+n/2] ← (U[j] - ω·V[j])/2
      ω ← ω·ω_n

  return P[0..n-1]
```

Figure A.3. Generic and radix-2 inverse FFT algorithms.

A.8 Fast Polynomial Multiplication

Finally, given two polynomials p and q, represented by an arrays of length m and n, respectively, we can multiply them in $\Theta((m+n)\log(m+n))$ arithmetic operations as follows. First, pad the coefficient vectors with zeros to length $m+n$ (or to the next larger power of 2 if we plan to use the radix-2 FFT algorithm). Then compute the discrete Fourier transforms of each coefficient vector, and multiply the resulting sample values one by one. Finally, compute the inverse discrete Fourier transform of the resulting sample vector.

```
FFTMULTIPLY(P[0..m−1], Q[0..n−1]):
  for j ← m to m+n−1
      P[j] ← 0
  for j ← n to m+n−1
      Q[j] ← 0

  P* ← FFT(P)
  Q* ← FFT(Q)
  for j ← 0 to m+n−1
      R*[j] ← P*[j] · Q*[j]
  return INVERSEFFT(R*)
```

Figure A.4. Multiplying polynomials in $O((m+n)\log(m+n))$ time.

A.9 Inside the Radix-2 FFT

Fast Fourier transforms are often implemented in hardware as circuits; Cooley and Tukey's radix-2 algorithm unfolds into a particularly nice recursive structure, as shown in Figure A.5 for $n = 16$. On the left, the n top-level inputs and outputs are connected to the inputs and outputs of the recursive calls, represented here as gray boxes. On the left we split the input P into two recursive inputs U and V. On the right, we combine the outputs U^* and V^* to obtain the final output P^*.

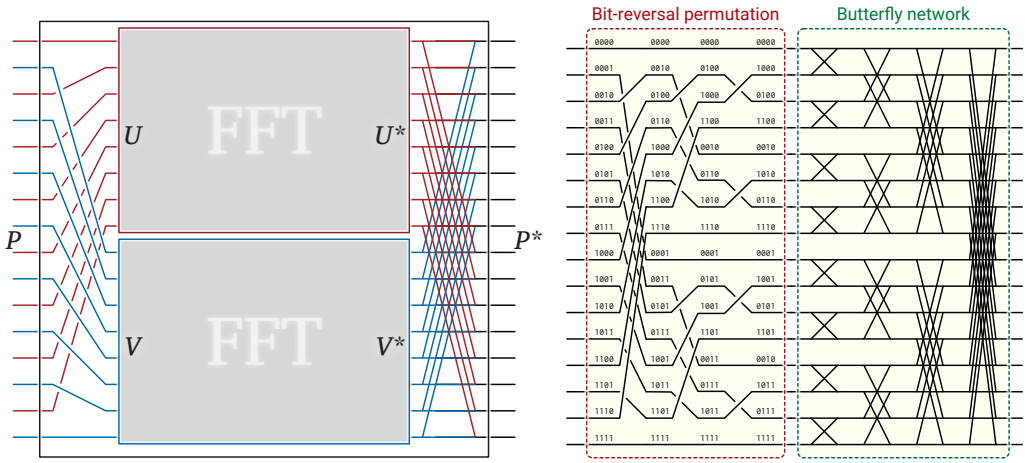


Figure A.5. The recursive structure of the radix-2 FFT algorithm.

If we expand this recursive structure completely, we see that the circuit splits naturally into two parts.

- The left half computes the *bit-reversal permutation* of the input. To find the position of $P[k]$ in this permutation, write k in binary and then read the bits backward. For example, in an 8-element bit-reversal permutation, $P[3] = P[011_2]$ ends up in position $6 = 110_2$.

- The right half of the FFT circuit is called a *butterfly network*. Butterfly networks are often used to route between processors in massively-parallel computers, because they allow any two processors to communicate in only $O(\log n)$ steps.

When n is a power of 2, recursively applying the more general FACTORFFT gives us exactly the same recursive structure, just clustered differently. For many applications of FFTs, including polynomial multiplication, the bit-reversal permutation is unnecessary and can actually be omitted.

Exercises

1. For any two sets X and Y of integers, the Minkowski sum $X + Y$ is the set of all pairwise sums $\{x + y \mid x \in X, y \in Y\}$.

   (a) Describe an analyze an algorithm to compute the number of elements in $X + Y$ in $O(n^2 \log n)$ time, where $n = |X| + |Y|$. *[Hint: The answer is* ***not*** *always $|X| \cdot |Y|$.]*

   (b) Describe and analyze an algorithm to compute the number of elements in $X + Y$ in $O(M \log M)$ time, where M is the largest absolute value of any element of $X \cup Y$. *[Hint: What's this lecture about?]*

2. Suppose we are given a bit string $B[1..n]$. A triple of distinct indices $1 \le i < j < k \le n$ is called a ***well-spaced triple*** in B if $B[i] = B[j] = B[k] = 1$ and $k - j = j - i$.

   (a) Describe a brute-force algorithm to determine whether B has a well-spaced triple in $O(n^2)$ time.

   (b) Describe an algorithm to determine whether B has a well-spaced triple in $O(n \log n)$ time. *[Hint: Hint.]*

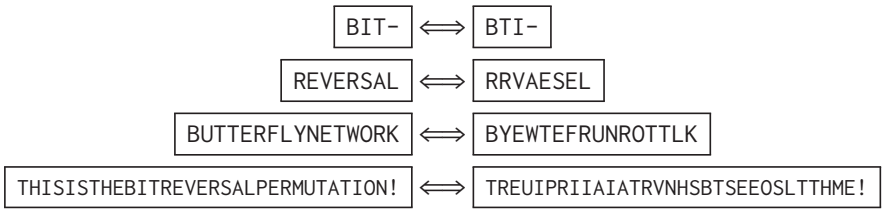
   (c) Describe an algorithm to determine the *number* of well-spaced triples in B in $O(n \log n)$ time.

3. (a) Describe an algorithm that determines whether a given set of n integers contains two elements whose sum is zero, in $O(n \log n)$ time.

   (b) Describe an algorithm that determines whether a given set of n integers contains *three* elements whose sum is zero, in $O(n^2)$ time.

   (c) Now suppose the input set X contains only integers between $-10000n$ and $10000n$. Describe an algorithm that determines whether X contains three elements whose sum is zero, in $O(n \log n)$ time. *[Hint: Hint.]*

4. Describe an algorithm that applies the bit-reversal permutation to an array $A[1..n]$ in $O(n)$ time when n is a power of 2.



♥5. Your new boss at the Dixon Ticonderoga Pencil Factory asks you to design an algorithm to solve the following problem. Suppose you are given N pencils, each with one of c different colors, and a non-negative integer k. ***How many different ways are there to choose a set of k pencils?*** Two pencil sets are considered identical if they contain the same number of pencils of each color.

For example, suppose you have two red pencils, four green pencils, and one blue pencil. Then you can form exactly five different two-pencil sets (RR, RG, RB, GG, GB), exactly six different four-pencil sets (RRGG, RRGB, RGGG, RGGB, GGGG, GGGB), and exactly three different six-pencil sets (RRGGGG, RRGGGB, RGGGGB).

Describe an algorithm to solve this problem, and analyze its running time. Your input is an array $Pencils[1..c]$ and an integer k, where $Pencils[i]$ stores the number of pencils with color i. Your output is a single non-negative integer. For example, given the input $Pencils = [2, 4, 1]$ and $k = 2$, your algorithm should return the integer 5.

For full credit, report the running time of your algorithm as a function of the parameters N (the total number of pencils), c (the number of colors), and k (the size of the target pencil sets). Assume that $k \ll c \ll N$, but do not assume that any of these parameters is a constant. *Assume for this problem that all arithmetic operations take $O(1)$ time.*

Hint:

$$\underbrace{(1+x+x^2)}_{\text{2 red pencils}} \cdot \underbrace{(1+x+x^2+x^3+x^4)}_{\text{4 green pencils}} \cdot \underbrace{(1+x)}_{\text{1 blue pencil}}$$

$$= 1 + 3x + \underbrace{5x^2}_{\text{5 2-pencil sets}} + 6x^3 + \underbrace{6x^4}_{\text{6 4-pencil sets}} + 5x^5 + \underbrace{3x^6}_{\text{3 6-pencil sets}} + 1$$

6. The FFT algorithm we described in this lecture is limited to polynomials with 2^k coefficients for some integer k. Of course, we can always pad the coefficient vector with zeros to force it into this form, but this padding artificially inflates the input size, leading to a slower algorithm than necessary.

Describe and analyze a similar DFT algorithm that works for polynomials with 3^k coefficients, by splitting the coefficient vector into three smaller vectors of length 3^{k-1}, recursively computing the DFT of each smaller vector, and correctly combining the results.

7. Fix an integer k. For any two k-bit integers i and j, let $\boldsymbol{i \wedge j}$ denote their bitwise AND, and let $\boldsymbol{\Sigma(i)}$ denote the number of 1s in the binary expansion of i. For example, when $k = 4$, we have $10 \wedge 7 = \texttt{1010} \wedge \texttt{0111} = \texttt{0010} = 2$ and $\Sigma(7) = \Sigma(\texttt{0111}) = 3$.

   The kth ***Sylvester-Hadamard matrix*** $\boldsymbol{H_k}$ is a $2^k \times 2^k$ matrix indexed by k-bit integers, each of whose entries is either $+1$ or -1, defined as follows:

$$H_k[i, j] = (-1)^{\Sigma(i \wedge j)}$$

   For example:

$$H_3 = \begin{bmatrix} +1 & +1 & +1 & +1 & +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 & +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 & +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 & +1 & -1 & -1 & +1 \\ +1 & +1 & +1 & +1 & -1 & -1 & -1 & -1 \\ +1 & -1 & +1 & -1 & -1 & +1 & -1 & +1 \\ +1 & +1 & -1 & -1 & -1 & -1 & +1 & +1 \\ +1 & -1 & -1 & +1 & -1 & +1 & +1 & -1 \end{bmatrix}$$

   (a) Prove that the matrix H_k can be decomposed into four copies of H_{k-1} as follows:

$$H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$$

   (b) Prove that $H_k \cdot H_k = 2^k \cdot I_k$, where I is the $2^k \times 2^k$ identity matrix.

   (c) For any vector $\vec{x} \in \mathbb{R}^{2^k}$, the product $H_k\vec{x}$ is called the ***Walsh-Hadamard transform*** of $\vec{x}$. Describe an algorithm to compute the Walsh-Hadamard transform in $O(n \log n)$ time, given the integer k and a vector of $n = 2^k$ integers as input.

♥8. The ***discrete Hartley transform*** of a vector $\vec{x} = (x_0, x_1, \ldots, x_{n-1})$ is another vector $\vec{X} = (X_0, X_1, \ldots, X_{n-1})$ defined as follows:

$$X_j = \sum_{i=0}^{n-1} x_i \left(\cos\left(\frac{2\pi}{n} ij\right) + \sin\left(\frac{2\pi}{n} ij\right) \right)$$

   Describe an algorithm to compute the discrete Hartley transform in $O(n \log n)$ time when n is a power of 2.

9. Suppose n is a power of 2. Prove that if we recursively apply the FACTORFFT algorithm, factoring of n into *arbitrary* smaller powers of two, the resulting algorithm still runs in $O(n \log n)$ time.

10. Let $f : \mathbb{Z} \to \mathbb{Z}$ be *any* function such that $2 \le f(n) \le \sqrt{n}$ for all $n \ge 4$. Prove that the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n \le 4 \\ f(n) \cdot T\left(\dfrac{n}{f(n)}\right) + \dfrac{n}{f(n)} \cdot T(f(n)) + O(n) & \text{otherwise} \end{cases}$$

has the solution $T(n) = O(n \log n)$. For example, setting $f(n) = 2$ for all n gives us the standard mergesort/FFT recurrence.

11. Although the radix-2 FFT algorithm and the more general recursive factoring algorithm both run in $O(n \log n)$ time, the latter algorithm is more efficient in practice (when n is large) due to caching effects.

Consider an idealized two-level memory model, which has an arbitrarily large but slow *main memory* and a significantly faster *cache* that holds the C most recently accessed memory items. The running time of an algorithm in this model is dominated by the number of *cache misses*, when the algorithm needs to access a value in memory that it not stored in the cache.

The number of cache misses seen by the radix-2 FFT algorithm obeys the following recurrence:

$$M(n) \le \begin{cases} 2M(n/2) + O(n) & \text{if } n > C \\ O(n) & \text{if } n \le C \end{cases}$$

If the input array is too large to fit in the cache, then *every* memory access in both the initial and final for-loops will cause a cache miss, and the recursive calls will cause their own cache misses. But if the input array is small enough to fit in cache, the initial for-loop loads the input into the cache, but there are no more cache misses, even inside the recursive calls.

(a) Solve the previous recurrence for $M(n)$, as a function of both n and C. To simplify the analysis, assume both n and C are powers of 2.

(b) Suppose we always recursively call FactorFFT with $p = C$ and $q = n/C$. How many cache misses does the resulting algorithm see, as a function of n and C? To simplify the analysis, assume $n = C^k$ for some integer k. (In particular, assume C is a power of 2.)

(c) Unfortunately, it is not always possible for a program (or a compiler) to determine the size of the cache. Suppose we always recursively call FactorFFT with $p = q = \sqrt{n}$. How many cache misses does the resulting algorithm see, as a function of n and C? To simplify the analysis, assume $n = C^{2^k}$ for some integer k.