Announcement: Homework 2 due tonight at 11:59PM (Gradescope)
Homework 3 released 😊
Karatsuba Multiplication

Multiply two \( n \)-bit integers \( x \) and \( y \):

- Add two \( \frac{1}{2}n \) bit integers.
- Multiply three \( \frac{1}{2}n \)-bit integers, recursively.
- Add, subtract, and shift to obtain result.

\[
T(n) = 3T(n/2) + O(n) \Rightarrow T(n) \text{ in } O(n^{1.585})
\]

Generalization: Multiply \((2k-1)\) pairs of \((n/k)\)-bit integers

\[
T(n) = (2k-1)T(n/k) + O(n) \Rightarrow T(n) \text{ in } O(n^{\log_{k}(2k-1)})
\]

\[
\lim_{k \to \infty} \log_{k}(2k - 1) = 1
\]

Matrix Multiplication

Multiply two \( n \times n \) matrices \( A \) and \( B \)

- Multiply 7 \((n/2)\times(n/2)\) matrices
- Add, Subtract and Shift to obtain result
Fast Matrix Multiplication

Key idea. multiply 2-by-2 blocks with only 7 multiplications.

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix} = 
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix} \times 
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]

- 7 multiplications.
- 18 = 8 + 10 additions and subtractions.

Example: 

\[P_1 + P_2 = A_{11}B_{12} - A_{11}B_{22} + A_{11}B_{22} + A_{12}B_{22} = A_{11}B_{12} + A_{12}B_{22} = C_{12}\]
Fast Matrix Multiplication

To multiply two $n$-by-$n$ matrices $A$ and $B$: [Strassen 1969]
- Divide: partition $A$ and $B$ into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
- Compute: $14 \frac{1}{2}n$-by-$\frac{1}{2}n$ matrices via $10$ matrix additions.
- Conquer: multiply $7$ pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
- Combine: $7$ products into $4$ terms using $8$ matrix additions.

Analysis.
- $T(n) = \#$ arithmetic operations.

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

- Apply Master Theorem ($a=7, b=2, c=2$)
  - $\left(\frac{a}{bc}\right) = \frac{7}{4} > 1 \quad \Rightarrow \quad T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 7}\right) = \Theta(n^{2.81})$
Fast Matrix Multiplication: Practice

Implementation issues.
- Sparsity.
- Caching effects.
- Numerical stability.
- Odd matrix dimensions.
- Crossover to classical algorithm around $n = 128$.

Common misperception. "Strassen is only a theoretical curiosity."
- Apple reports 8x speedup on G4 Velocity Engine when $n \approx 2,500$.
- Range of instances where it's useful is a subject of controversy.

Remark. Can "Strassenize" $Ax = b$, determinant, eigenvalues, SVD, ....
Fast Matrix Multiplication: Theory

Q. Multiply two 2-by-2 matrices with 7 scalar multiplications?
   A. Yes! [Strassen 1969] \( \Theta(n^\log_2 7) = O(n^{2.807}) \)

Q. Multiply two 2-by-2 matrices with 6 scalar multiplications?
   A. Impossible. [Hopcroft and Kerr 1971] \( \Theta(n^\log_2 6) = O(n^{2.59}) \)

Q. Two 3-by-3 matrices with 21 scalar multiplications?
   A. Also impossible. \( \Theta(n^\log_3 21) = O(n^{2.77}) \)

Begun, the decimal wars have. [Pan, Bini et al, Schönhage, ...]

- Two 20-by-20 matrices with 4,460 scalar multiplications. \( O(n^{2.805}) \)
- Two 48-by-48 matrices with 47,217 scalar multiplications. \( O(n^{2.7801}) \)
- A year later. \( O(n^{2.7799}) \)
- December, 1979. \( O(n^{2.521813}) \)
- January, 1980. \( O(n^{2.521801}) \)
Best known. \( O(n^{2.376}) \) [Coppersmith–Winograd, 1987]

Conjecture. \( O(n^{2+\varepsilon}) \) for any \( \varepsilon > 0 \).

Caveat. Theoretical improvements to Strassen are progressively less practical.
Fast Matrix Multiplication: Theory

**Best known.** $O(n^{2.373})$ [Williams, 2014]

**Conjecture.** $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

**Caveat.** Theoretical improvements to Strassen are progressively less practical.
Fast Matrix Multiplication: Theory

**Best known.** $O(n^{2.3729})$ [Le Gall, 2014]

**Conjecture.** $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

**Caveat.** Theoretical improvements to Strassen are progressively less practical.
5.6 Convolution and FFT
Polynomials: Coefficient Representation

**Polynomial.** [coefficient representation]

\[ A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \]

\[ B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1} \]

**Add:** \( O(n) \) arithmetic operations.

\[ A(x) + B(x) = (a_0 + b_0) + (a_1 + b_1)x + \cdots + (a_{n-1} + b_{n-1})x^{n-1} \]

**Evaluate:** \( O(n) \) using Horner's method.

\[ A(x) = a_0 + (x(a_1 + x(a_2 + \cdots + x(a_{n-2} + x(a_{n-1})))) \cdots) \]

**Multiply (convolve):** \( O(n^2) \) using brute force.

\[ A(x) \times B(x) = \sum_{i=0}^{2n-2} c_i x^i, \quad \text{where} \quad c_i = \sum_{j=0}^{i} a_j b_{i-j} \]
Polynomials: Point-Value Representation

**Fundamental theorem of algebra.** [Gauss, PhD thesis] A degree $n$ polynomial with complex coefficients has $n$ complex roots.

**Corollary.** A degree $n-1$ polynomial $A(x)$ is uniquely specified by its evaluation at $n$ distinct values of $x$.

**Pf:** Suppose both $A(x_i)=B(x_i)$ at $n$ points

Consider $C(x)=A(x)-B(x) \Rightarrow C(x_i)=0$

- has degree $n-1$ but $n$ roots?

$y_j = A(x_j)$

$y$ $x$
Polynomials: Point-Value Representation

**Polynomial.** [point-value representation]

\[ A(x) : (x_0, y_0), \ldots, (x_{n-1}, y_{n-1}) \]

\[ B(x) : (x_0, z_0), \ldots, (x_{n-1}, z_{n-1}) \]

**Add:** \(O(n)\) arithmetic operations.

\[ A(x) + B(x) : (x_0, y_0 + z_0), \ldots, (x_{n-1}, y_{n-1} + z_{n-1}) \]

**Multiply:** \(O(n)\), but need \(2n-1\) points.

\[ A(x) \times B(x) : (x_0, y_0 \times z_0), \ldots, (x_{2n-1}, y_{2n-1} \times z_{2n-1}) \]

**Evaluate:** \(O(n^2)\) using Lagrange's formula.

\[ A(x) = \sum_{k=0}^{n-1} y_k \prod_{j \neq k}^{n-1} \frac{(x - x_j)}{(x_k - x_j)} \]
Converting Between Two Polynomial Representations

**Tradeoff.** Fast evaluation or fast multiplication. We want both!

<table>
<thead>
<tr>
<th>Representation</th>
<th>Multiply</th>
<th>Evaluate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coefficient</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Point-value</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Goal.** Make all ops fast by efficiently converting between two representations.

coefficient representation \[a_0, a_1, \ldots, a_{n-1}\] \[\rightarrow\] point-value representation \[\{(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})\}\]
Theorem. Inverse FFT algorithm interpolates a degree $n-1$ polynomial given values at each of the $n^{th}$ roots of unity in $O(n \log n)$ steps.

assumes $n$ is a power of 2

Theorem. FFT algorithm evaluates a degree $n-1$ polynomial at each of the $n^{th}$ roots of unity in $O(n \log n)$ steps.
Polynomial Multiplication

**Theorem.** Can multiply two degree $n-1$ polynomials in $O(n \log n)$ steps.

A polynomial $A(x)$ of degree $n-1$ can be represented as:

$$A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$$

Similarly, another polynomial $B(x)$ of degree $n-1$ can be represented as:

$$B(x) = b_0 + b_1 x + \cdots + b_{n-1} x^{n-1}$$

The product $C(x)$ of these two polynomials is:

$$C(x) = c_0 + c_1 x + \cdots + c_{2n-2} x^{2n-2}$$

**Algorithm:**

1. **Coefficient Representation:** Convert $A(x)$ and $B(x)$ to their coefficient representations $\{a_0, a_1, \ldots, a_{n-1}\}$ and $\{b_0, b_1, \ldots, b_{n-1}\}$.
2. **FFT:** Use the Fast Fourier Transform (FFT) to find $A(x)$ and $B(x)$ at $2n$ points $x_0, x_1, \ldots, x_{2n-1}$.
3. **Point-Value Multiplication:** Multiply the point-values $A(x_i)B(x_i)$ for each $i$.
4. **Inverse FFT:** Use the Inverse FFT to convert the point-value multiplication back to the coefficient representation.

The total time complexity is $O(n \log n)$. For the conversion between coefficient representation and point-value representation, the time complexity is $O(n)$. The overall time complexity is $O(n \log n)$. This is the fastest known algorithm for polynomial multiplication.
Touch Tone

Button 1 signal. [exact]
\[\frac{1}{2}\sin(2\pi \times 697t) + \frac{1}{2}\sin(2\pi \times 1209t)\]

Magnitude of Fourier transform of button 1 signal.

Reference: Cleve Moler, Numerical Computing with MATLAB
Touch Tone

Button 1 signal.  [recorded, 8192 samples per second]

Magnitude of FFT.

Reference: Cleve Moler, Numerical Computing with MATLAB
Fast Fourier Transform: Applications

Applications.

- Optics, acoustics, quantum physics, telecommunications, control systems, signal processing, speech recognition, data compression, image processing.
- DVD, JPEG, MP3, MRI, CAT scan.
- Numerical solutions to Poisson’s equation.

The FFT is one of the truly great computational developments of this [20th] century. It has changed the face of science and engineering so much that it is not an exaggeration to say that life as we know it would be very different without the FFT.  -Charles van Loan
Fast Fourier Transform: Brief History

Gauss (1805, 1866). Analyzed periodic motion of asteroid Ceres.


Importance not fully realized until advent of digital computers.
Converting Between Two Polynomial Representations: Brute Force

**Coefficient to point-value.** Given a polynomial \( a_0 + a_1 x + \ldots + a_{n-1} x^{n-1} \), evaluate it at \( n \) distinct points \( x_0, \ldots, x_{n-1} \).

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

- \( O(n^2) \) for matrix-vector multiply
- \( O(n^3) \) for Gaussian elimination

Vandermonde matrix is invertible iff \( x_i \) distinct

**Point-value to coefficient.** Given \( n \) distinct points \( x_0, \ldots, x_{n-1} \) and values \( y_0, \ldots, y_{n-1} \), find unique polynomial \( a_0 + a_1 x + \ldots + a_{n-1} x^{n-1} \) that has given values at given points.
Coefficient to Point-Value Representation: Intuition

**Coefficient to point-value.** Given a polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, evaluate it at $n$ distinct points $x_0, \ldots, x_{n-1}$.

**Divide.** Break polynomial up into even and odd powers.

- $A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7$.
- $A_{\text{even}}(x) = a_0 + a_2 x + a_4 x^2 + a_6 x^3$.
- $A_{\text{odd}}(x) = a_1 + a_3 x + a_5 x^2 + a_7 x^3$.
- $A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2)$.
- $A(-x) = A_{\text{even}}(x^2) - x A_{\text{odd}}(x^2)$.

**Intuition.** Choose two points to be $\pm 1$.

- $A(1) = A_{\text{even}}(1) + 1 A_{\text{odd}}(1)$.
- $A(-1) = A_{\text{even}}(1) - 1 A_{\text{odd}}(1)$.

Can evaluate polynomial of degree $\leq n$ at 2 points by evaluating two polynomials of degree $\leq \frac{1}{2}n$ at 1 point.
Coefficient to Point-Value Representation: Intuition

Coefficient to point-value. Given a polynomial $a_0 + a_1 x + ... + a_{n-1} x^{n-1}$, evaluate it at $n$ distinct points $x_0, ..., x_{n-1}$.

Divide. Break polynomial up into even and odd powers.

- $A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7$.
- $A_{even}(x) = a_0 + a_2 x + a_4 x^2 + a_6 x^3$.
- $A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + a_7 x^3$.
- $A(x) = A_{even}(x^2) + x A_{odd}(x^2)$.
- $A(-x) = A_{even}(x^2) - x A_{odd}(x^2)$.

Intuition. Choose four points to be $\pm 1$, $\pm i$.

- $A(1) = A_{even}(1) + 1 A_{odd}(1)$.
- $A(-1) = A_{even}(1) - 1 A_{odd}(1)$.
- $A(i) = A_{even}(-1) + i A_{odd}(-1)$.
- $A(-i) = A_{even}(-1) - i A_{odd}(-1)$.

Can evaluate polynomial of degree $\leq n$ at 4 points by evaluating two polynomials of degree $\leq \frac{1}{2}n$ at 2 points.
Coefficient to Point-Value Representation: Intuition

Coefficient to point-value. Given a polynomial \( a_0 + a_1 x + \ldots + a_{n-1} x^{n-1} \), evaluate it at \( n \) distinct points \( x_0, \ldots, x_{n-1} \).

Divide. Break polynomial up into even and odd powers.

- \( A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7 \).
- \( A_{\text{even}}(x) = a_0 + a_2 x + a_4 x^2 + a_6 x^3 \).
- \( A_{\text{odd}}(x) = a_1 + a_3 x + a_5 x^2 + a_7 x^3 \).
- \( A(-x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2) \).
- \( A(-x) = A_{\text{even}}(x^2) - x A_{\text{odd}}(x^2) \).

Intuition. Choose four points to be \( \pm 1, \pm i \).

- \( A(1) = A_{\text{even}}(1) + 1 A_{\text{odd}}(1) \).
- \( A(-1) = A_{\text{even}}(1) - 1 A_{\text{odd}}(1) \).
- \( A(i) = A_{\text{even}}(-1) + i A_{\text{odd}}(-1) \).
- \( A(-i) = A_{\text{even}}(-1) - i A_{\text{odd}}(-1) \).

Goal: evaluate polynomial of degree \( \leq n \) at \( n \) points by evaluating two polynomials of degree \( \leq \frac{1}{2} n \) at \( n/2 \) points.
Discrete Fourier Transform

**Coefficient to point-value.** Given a polynomial \( a_0 + a_1 x + ... + a_{n-1} x^{n-1} \), evaluate it at \( n \) distinct points \( x_0, ... , x_{n-1} \).

**Key idea:** choose \( x_k = \omega^k \) where \( \omega \) is principal \( n^{th} \) root of unity.

\[
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
y_3 \\
\vdots \\
y_{n-1}
\end{bmatrix}
= 
\begin{bmatrix}
1 & 1 & 1 & 1 & \cdots & 1 \\
1 & \omega^1 & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\
1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(n-1)} \\
1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(n-1)} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{n-1} & \omega^{2(n-1)} & \omega^{3(n-1)} & \cdots & \omega^{(n-1)(n-1)}
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 \\
\vdots \\
a_{n-1}
\end{bmatrix}
\]

Discrete Fourier transform\quad Fourier matrix \( F_n \)
**Def.** An $n^\text{th}$ root of unity is a complex number $x$ such that $x^n = 1$.

**Fact.** The $n^\text{th}$ roots of unity are: $\omega^0, \omega^1, \ldots, \omega^{n-1}$ where $\omega = e^{\frac{2\pi i}{n}}$.

**Pf.** $(\omega^k)^n = (e^{\frac{2\pi i k}{n}})^n = (e^{\pi i})^{2k} = (-1)^{2k} = 1$.

**Fact.** The $\frac{1}{2}n^\text{th}$ roots of unity are: $\nu^0, \nu^1, \ldots, \nu^{n/2-1}$ where $\nu = e^{\frac{4\pi i}{n}}$.

**Fact.** $\omega^2 = \nu$ and $(\omega^2)^k = \nu^k$. 

![Diagram of roots of unity for n = 8](image)
Fast Fourier Transform

**Goal.** Evaluate a degree n-1 polynomial \( A(x) = a_0 + \ldots + a_{n-1} x^{n-1} \) at its \( n^{th} \) roots of unity: \( \omega^0, \omega^1, \ldots, \omega^{n-1} \).

**Divide.** Break polynomial up into even and odd powers.
- \( A_{\text{even}}(x) = a_0 + a_2x + a_4x^2 + \ldots + a_{n/2-2} x^{(n-1)/2} \).
- \( A_{\text{odd}}(x) = a_1 + a_3x + a_5x^2 + \ldots + a_{n/2-1} x^{(n-1)/2} \).
- \( A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2) \).

**Conquer.** Evaluate degree \( A_{\text{even}}(x) \) and \( A_{\text{odd}}(x) \) at the \( \frac{1}{2}n^{th} \) roots of unity: \( \gamma^0, \gamma^1, \ldots, \gamma^{n/2-1} \).

**Combine.**
- \( A(\omega^k) = A_{\text{even}}(\gamma^k) + \omega^k A_{\text{odd}}(\gamma^k), \quad 0 \leq k < n/2 \)
- \( A(\omega^{k+n/2}) = A_{\text{even}}(\gamma^{k}) - \omega^k A_{\text{odd}}(\gamma^{k}), \quad 0 \leq k < n/2 \)

\[ \omega^{k+\frac{1}{2}n} = -\omega^k \]

\( \gamma^k = (\omega^k)^2 = \omega^n (\omega^k)^2 = (\omega^{k+\frac{1}{2}n})^2 \)
fft(n, a_0,a_1,…,a_{n-1}) {
    if (n == 1) return a_0

    (e_0,e_1,…,e_{n/2-1}) \leftarrow \text{FFT}(n/2, a_0,a_2,a_4,…,a_{n-2})
    (d_0,d_1,…,d_{n/2-1}) \leftarrow \text{FFT}(n/2, a_1,a_3,a_5,…,a_{n-1})

    \text{for} \ k = 0 \ \text{to} \ n/2 - 1 \ \{ \n        \omega^k \leftarrow e^{2\pi i k/n}
        y_k \leftarrow e_k + \omega^k \ d_k
        y_{k+n/2} \leftarrow e_k - \omega^k \ d_k
    \}\n
    \text{return} \ (y_0,y_1,…,y_{n-1})
}
FFT Summary

**Theorem.** FFT algorithm evaluates a degree n-1 polynomial at each of the n\(^{th}\) roots of unity in \(O(n \log n)\) steps.

```
assumes n is a power of 2
```

**Running time.** \(T(2n) = 2T(n) + O(n) \Rightarrow T(n) = O(n \log n)\).
Recursion Tree

```
  a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7

  perfect shuffle

  a_0, a_2, a_4, a_6
  |    |    |    |
  v    v    v    v
  a_0, a_4
  |    |    |
  v    v    v
  a_0  a_4

  a_1, a_3, a_5, a_7
  |    |    |    |
  v    v    v    v
  a_1, a_5
  |    |    |
  v    v    v
  a_3  a_7

  "bit-reversed" order
```

000 100 010 110 001 101 011 111
**Goal.** Given the values \(y_0, \ldots, y_{n-1}\) of a degree \(n-1\) polynomial at the \(n\) points \(\omega^0, \omega^1, \ldots, \omega^{n-1}\), find unique polynomial \(a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}\) that has given values at given points.

$$
\begin{bmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 \\
\vdots \\
a_{n-1}
\end{bmatrix}
= \begin{bmatrix}
1 & 1 & 1 & 1 & \cdots & 1 \\
1 & \omega^1 & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\
1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(n-1)} \\
1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(n-1)} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{n-1} & \omega^{2(n-1)} & \omega^{3(n-1)} & \cdots & \omega^{(n-1)(n-1)}
\end{bmatrix}^{-1}
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
y_3 \\
\vdots \\
y_{n-1}
\end{bmatrix}
$$

**Inverse DFT**

**Fourier matrix inverse \((F_n)^{-1}\)**
Claim. Inverse of Fourier matrix is given by following formula.

\[
G_n = \frac{1}{n} \begin{bmatrix}
1 & 1 & 1 & 1 & \cdots & 1 \\
1 & \omega^{-1} & \omega^{-2} & \omega^{-3} & \cdots & \omega^{-(n-1)} \\
1 & \omega^{-2} & \omega^{-4} & \omega^{-6} & \cdots & \omega^{-2(n-1)} \\
1 & \omega^{-3} & \omega^{-6} & \omega^{-9} & \cdots & \omega^{-3(n-1)} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \omega^{-3(n-1)} & \cdots & \omega^{-(n-1)(n-1)}
\end{bmatrix}
\]

Consequence. To compute inverse FFT, apply same algorithm but use \(\omega^{-1} = e^{-2\pi i/n}\) as principal \(n^{th}\) root of unity (and divide by \(n\)).
Inverse FFT: Proof of Correctness

**Claim.** \( F_n \) and \( G_n \) are inverses.

**Pf.**

\[
(F_n G_n)_{kk'} = \frac{1}{n} \sum_{j=0}^{n-1} \omega^{kj} \omega^{-jk'} = \frac{1}{n} \sum_{j=0}^{n-1} \omega^{(k-k')j} = \begin{cases} 
1 & \text{if } k = k' \\
0 & \text{otherwise}
\end{cases}
\]

**Summation lemma.** Let \( \omega \) be a principal \( n \)th root of unity. Then

\[
\sum_{j=0}^{n-1} \omega^{kj} = \begin{cases} 
n & \text{if } k \equiv 0 \mod n \\
0 & \text{otherwise}
\end{cases}
\]

**Pf.**

- If \( k \) is a multiple of \( n \) then \( \omega^k = 1 \Rightarrow \) sums to \( n \).
- Each \( n \)th root of unity \( \omega^k \) is a root of
  \[
x^n - 1 = (x - 1) (1 + x + x^2 + \ldots + x^{n-1}).
\]
- If \( \omega^k \neq 1 \) we have: \( 1 + \omega^k + \omega^{k(2)} + \ldots + \omega^{k(n-1)} = 0 \Rightarrow \) sums to 0.
Inverse FFT: Algorithm

```plaintext
ifft(n, a_0, a_1, ..., a_{n-1}) {
    if (n == 1) return a_0

    (e_0, e_1, ..., e_{n/2-1}) ← FFT(n/2, a_0, a_2, a_4, ..., a_{n-2})
    (d_0, d_1, ..., d_{n/2-1}) ← FFT(n/2, a_1, a_3, a_5, ..., a_{n-1})

    for k = 0 to n/2 - 1 {
        \omega^k ← e^{-2\pi i k / n}
        y_k ← (e_k + \omega^k d_k) / n
        y_{k+n/2} ← (e_k - \omega^k d_k) / n
    }

    return (y_0, y_1, ..., y_{n-1})
}
```
Inverse FFT Summary

**Theorem.** Inverse FFT algorithm interpolates a degree n-1 polynomial given values at each of the n\(^{th}\) roots of unity in \(O(n \log n)\) steps.

assumes \(n\) is a power of 2
Theorem. Can multiply two degree n-1 polynomials in $O(n \log n)$ steps.
FFT in Practice?

[Image of Google search results for FFT in Java]

- FFT in Java
- FFT code in Java...Compilation: javac FFT.java * Execution: java FFT N * Dependencies: Complex.java * Compute the FFT and inverse FFT of a length N.
- YOV/408 Programming Resources - Code Spotlight - FFT Java source...
  Compilation: javac FFT.java * Execution: java FFT N * Dependencies: ...A nice implementation of the FFT algorithm in Java. Even though it can use too much...
- FFT JAVA Demo
  This is a JAVA applet demonstrating basic concept of Fast Fourier...If you want to run the program, copy it, download FFT.zip and unzip it to a directory.
  www.ing.unipmn.it-iklas/Projects/displ-8k - Cached - Similar pages
- Mathtools.net - Java/FFT
  Listing of Java FFT related links, tools, and resources.
  www.mathtools.net/Java/FFT/index.html - 19k - Cached - Similar pages
- FFT Spectrum Analyser Demo
  The following features are new in the Java 1.1 version of the FFT Spectrum Analyser applica...
  The signal is plotted in either the time domain (signal) or the...
  www.dispsector.freeuk.com/analyser/SpectrumAnalyser.html - 4k - Cached - Similar pages
- Fun with Java Understanding the Fast Fourier Transform (FFT)
  Fun with Java, Understanding the Fast Fourier Transform (FFT) Algorithm By Richard G.
  Baldwin...Java Programming, Notes # 1466. Preface; General Discussion...
  wwwdeveloper.com/java/otherarticle.php?3407251 - 11k - Cached - Similar pages
- Spectrum Analysis using Java, Sampling Frequency, Folding...
  File Dsp030.java Copyright 2004, Richarditch, w 5/16/04 Uses an FFT algorithm to compute...
  display and the magnitude of the spectrum content for up to the...
  wwwdeveloper.com/javareferencearticle.php?3360331 - 279k - Cached - Similar pages
- Bruce R. Miller's Java(TM) Demo Page
  These classes may be of use to other java programmers. Available Packages, Demos & Bug...
  Fixes; FFT, TabPanel, ObjectList, Scroll, etc.
  math.nist.gov/BMillerJava/ - 7k - Cached - Similar pages
- FFT Java Glossary
  Roedy Green's Java & Internet Glossary: FFT. ...You are here: home ← Java Glossary ← F Words ← FFT, FFT. Fast Fourier Transform...
  mindsprod.com/jgloss/fft.html - 8k - Cached - Similar pages

FFT java
Dynamic Programming
Algorithmic Paradigms

**Greedy.** Build up a solution incrementally, myopically optimizing some local criterion.

**Divide-and-conquer.** Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

**Dynamic programming.** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
Dynamic Programming History

Bellman. [1950s] Pioneered the systematic study of dynamic programming.

Etymology.
- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.

"it's impossible to use dynamic in a pejorative sense"  
"something not even a Congressman could object to"

Dynamic Programming Applications

Areas.
- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ...

Some famous dynamic programming algorithms.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.
6.1 Weighted Interval Scheduling
Weighted interval scheduling problem.

- Job $j$ starts at $s_j$, finishes at $f_j$, and has weight or value $v_j$.
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.
Unweighted Interval Scheduling (will cover in Greedy paradigms)

Previously Showed: Greedy algorithm works if all weights are 1.

- **Solution:** Sort requests by finish time (ascending order)

**Observation.** Greedy algorithm can fail spectacularly if arbitrary weights are allowed.
**Notation.** Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

**Def.** $p(j) = \text{largest index } i < j \text{ such that job } i \text{ is compatible with } j$.

**Ex:** $p(8) = 5$, $p(7) = 3$, $p(2) = 0$. 

![Diagram showing weighted interval scheduling with times and jobs labeled.](image-url)
Dynamic Programming: Binary Choice

**Notation.** \( OPT(j) = \text{value of optimal solution to the problem consisting of job requests } 1, 2, \ldots, j. \)

- **Case 1:** \( OPT \) selects job \( j \).
  - collect profit \( v_j \)
  - can't use incompatible jobs \( \{ p(j) + 1, p(j) + 2, \ldots, j - 1 \} \)
  - must include optimal solution to problem consisting of remaining compatible jobs \( 1, 2, \ldots, p(j) \)

- **Case 2:** \( OPT \) does not select job \( j \).
  - must include optimal solution to problem consisting of remaining compatible jobs \( 1, 2, \ldots, j-1 \)

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left\{ v_j + OPT(p(j)), \ OPT(j-1) \right\} & \text{otherwise}
\end{cases}
\]
Input: $n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p(1), p(2), \ldots, p(n)$

Compute-Opt$(j)$ {
    if $(j = 0)$
        return 0
    else
        return max$(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1))$
}

$T(n) = T(n-1) + T(p(n)) + O(1)$
$T(1) = 1$
Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence ($F_n > 1.6^n$).

Key Insight: Do we really need to repeat this computation?
Memoization. Store results of each sub-problem in a cache; lookup as needed.

Input: \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

Sort jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).
Compute \( p(1), p(2), \ldots, p(n) \)

\[
\text{for } j = 1 \text{ to } n \\
\quad M[j] = \text{empty} \\
\quad M[0] = 0
\]

M-Compute-Opt\((j)\) {
    if (M\([j]\) is empty)
        M\([j]\) = \max(v_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
    return M\([j]\)
}
Claim. Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$ via sorting by start time.

- $\text{M-Compute-Opt}(j)$: each invocation takes $O(1)$ time and either
  - (i) returns an existing value $M[j]$
  - (ii) fills in one new entry $M[j]$ and makes two recursive calls

- Progress measure $\Phi = \# \text{nonempty entries of } M[]$.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases $\Phi$ by 1 $\Rightarrow$ at most $2n$ recursive calls.

- Overall running time of $\text{M-Compute-Opt}(n)$ is $O(n)$.

Remark. $O(n)$ if jobs are pre-sorted by start and finish times.
Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?
A. Do some post-processing.

\[
\text{Find-Solution}(j) \begin{cases} 
\text{if } (j = 0) & \text{output nothing} \\
\text{else if } (v_j + M[p(j)] > M[j-1]) & \text{print } j \\
\text{else} & \text{Find-Solution}(p(j)) \\
\text{Find-Solution}(j-1) & \end{cases} 
\]

- \# of recursive calls \( \leq n \) \( \Rightarrow \) \( O(n) \).
Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

Input: \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

Sort jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).

Compute \( p(1), p(2), \ldots, p(n) \)

Iterative-Compute-Opt {
  \( M[0] = 0 \)
  for \( j = 1 \) to \( n \)
  \( M[j] = \max(v_j + M[p(j)], M[j-1]) \)
}