Fast Matrix Multiplication

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

\[
\begin{bmatrix}
C_0 & C_1 \\
C_0 & C_1
\end{bmatrix}
= \begin{bmatrix}
A_0 & A_1 \\
A_0 & A_1
\end{bmatrix}
\begin{bmatrix}
B_0 & B_1 \\
B_0 & B_1
\end{bmatrix}
\]

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

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 fix speedup on 64 Velocity Engine when \( n = 2560 \).
- Range of instances where it’s useful is a subject of controversy.

Remark. Can “Strassenize” \( Ax = b \), determinant, eigenvalues, SVD, …

Fast Matrix Multiplication: Theory

- \( \mathcal{O}(n \log_2 7) \) operations.
- \( \mathcal{O}(n^{2.807}) \) for \( n \geq 2 \times 22 \).
- \( \mathcal{O}(n^{2.779}) \) for \( n \geq 2 \times 21 \).
- \( \mathcal{O}(n^{2.78}) \) for \( n \geq 2 \times 20 \).
- \( \mathcal{O}(n^{2.779}) \) for \( n \geq 2 \times 19 \).
- \( \mathcal{O}(n^{2.78}) \) for \( n \geq 2 \times 18 \).

- \( \mathcal{O}(n^{2.779}) \) for \( n \geq 2 \times 17 \).
- \( \mathcal{O}(n^{2.78}) \) for \( n \geq 2 \times 16 \).
- \( \mathcal{O}(n^{2.779}) \) for \( n \geq 2 \times 15 \).
- \( \mathcal{O}(n^{2.78}) \) for \( n \geq 2 \times 14 \).

- \( \mathcal{O}(n^{2.779}) \) for \( n \geq 2 \times 13 \).
- \( \mathcal{O}(n^{2.78}) \) for \( n \geq 2 \times 12 \).

- \( \mathcal{O}(n^{2.779}) \) for \( n \geq 2 \times 11 \).
- \( \mathcal{O}(n^{2.78}) \) for \( n \geq 2 \times 10 \).

- \( \mathcal{O}(n^{2.779}) \) for \( n \geq 2 \times 9 \).
- \( \mathcal{O}(n^{2.78}) \) for \( n \geq 2 \times 8 \).

- \( \mathcal{O}(n^{2.779}) \) for \( n \geq 2 \times 7 \).
- \( \mathcal{O}(n^{2.78}) \) for \( n \geq 2 \times 6 \).

- \( \mathcal{O}(n^{2.779}) \) for \( n \geq 2 \times 5 \).
- \( \mathcal{O}(n^{2.78}) \) for \( n \geq 2 \times 4 \).

- \( \mathcal{O}(n^{2.779}) \) for \( n \geq 2 \times 3 \).
- \( \mathcal{O}(n^{2.78}) \) for \( n \geq 2 \times 2 \).

- \( \mathcal{O}(n^{2.779}) \) for \( n \geq 2 \times 1 \).
- \( \mathcal{O}(n^{2.78}) \) for \( n = 2 \).

- \( \mathcal{O}(n^{2.779}) \) for \( n = 1 \).

Begun, the decimal wars have. [Pan, Bini et al, Schönhage, …]
- Two 20-by-20 matrices with 4,440 scalar operations. \( \mathcal{O}(n^{2.779}) \)
- Two 48-by-48 matrices with 7,217 scalar operations. \( \mathcal{O}(n^{2.779}) \)
- A year later. \( \mathcal{O}(n^{2.779}) \)
- December, 1979. \( \mathcal{O}(n^{2.779}) \)
- January, 1980. \( \mathcal{O}(n^{2.779}) \)

Fast Matrix Multiplication: Practice

To multiply two \( n \times n \) matrices \( A \) and \( B \):
- Divide: partition \( A \) and \( B \) into \( 3 \times 3 \) blocks.
- Compute: \( 14 \) \( 3 \times 3 \) matrices via 10 matrix additions.
- Conquer: multiply 7 pairs of \( 3 \times 3 \) matrices, recursively.
- Combine: 7 products into 4 terms using 3 matrix additions.

Analysis.\[
T(n) = 7T(\lceil n/2 \rceil) + 16n^2 \Rightarrow T(n) = \mathcal{O}(n^{2.807})
\]

Applying Master Theorem (\( a = 7, b = 2, c = 2 \))
- \( T(n) = (2k-1)T(n/k) + O(n) \)
- \( T(n) = 3T(n/2) + O(n) \)
- \( T(n) = \Theta(n^2) \)

Generalization: Multiply \( (2k-1) \) pairs of \( (n/k) \times (n/k) \) bits integers.
- \( T(n) = (2k-1)T(n/k) + O(n) \Rightarrow T(n) = \mathcal{O}(n^{\log_2 (2k-1)}) \)

Announcement: Homework 2 due tonight at 11:59PM (Gradescope)
2/4/2019

Polynomials: Coefficient Representation

Polynomial: [Coefficient representation]

\[ A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \]

\[ B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_n x^n \]

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

\[ A(x) + B(x) = (a_0 + b_0) + (a_1 + b_1) x + \cdots + (a_n + b_n) x^n \]

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

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

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

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

5.6 Convolution and FFT

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 \( x \).
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.

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

Converting Between Two Polynomial Representations

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

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

<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>

(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.

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

\[ a_0, a_1, \ldots, a_{n-1} \rightarrow (\omega^0, y_0), \ldots, (\omega^{n-1}, y_{n-1}) \]

Touch Tone

**Button 1 signal** (exact)

Magnitude of Fourier transform of button 1 signal.

Reference: Cleve Moler, Numerical Computing with MATLAB
Fast Fourier Transform: 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 \( a_0 + a_1 x + \ldots + a_n x^n \), evaluate it at \( n \) distinct points \( x_0, \ldots, x_{n-1} \).

\[
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
y_3 \\
y_{n-1}
\end{bmatrix} =
\begin{bmatrix}
x_0^n & x_0^{n-1} & \ldots & x_0 & 1 \\
x_1^n & x_1^{n-1} & \ldots & x_1 & 1 \\
x_2^n & x_2^{n-1} & \ldots & x_2 & 1 \\
x_3^n & x_3^{n-1} & \ldots & x_3 & 1 \\
x_{n-1}^n & x_{n-1}^{n-1} & \ldots & x_{n-1} & 1
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 \\
a_{n-1}
\end{bmatrix}
\]

Order \( n^2 \) for matrix-vector multiply

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 \( a_0 + a_1 x + \ldots + a_n x^n \) that has given values at given points.

Coefficient to Point-Value Representation: Intuition

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

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

Intuition. Choose four points to be \( \pm 1, \pm i \).
- \( A(-i) = A_{\text{even}}(-1) - i A_{\text{odd}}(-1) \).
- \( A(-i) = A_{\text{even}}(-1) + i A_{\text{odd}}(-1) \).

Goal: evaluate polynomial of degree \( n \) at \( n/2 \) points by evaluating two polynomials of degree \( \lceil n/2 \rceil \) at \( 2 \) points.
Discrete Fourier Transform

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 $x_0, \ldots, x_{n-1}$.

Key idea: choose $x_k = \omega^k$ where $\omega$ is principal $n$th root of unity.

Roots of Unity

Def. An $n$th root of unity is a complex number $x$ such that $x^n = 1$.

Fact. The $n$th roots of unity are: $\omega_0, \omega_1, \ldots, \omega_{n-1}$ where $\omega_k = e^{2\pi i k / n}$.

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

Fact. The $\frac{n}{2}$th roots of unity are: $\omega_0, \omega_1, \ldots, \omega_{n/2-1}$ where $\omega_k = e^{4\pi i k / n}$.

Fact. $\omega_2 = \omega$ and $(\omega_2)^k = \omega_k$.

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.

- $A_{\mathrm{even}}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n-2}$
- $A_{\mathrm{odd}}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n-1}$
- $A(x) = A_{\mathrm{even}}(x^2) + x A_{\mathrm{odd}}(x^2)$

Conquer. Evaluate degree $A_{\mathrm{even}}(x)$ and $A_{\mathrm{odd}}(x)$ at the $\frac{n}{2}$th roots of unity: $\omega_0, \omega_1, \ldots, \omega_{n/2-1}$.

Combine.

- $A(\omega^k) = A_{\mathrm{even}}(\omega^k) + \omega^k A_{\mathrm{odd}}(\omega^k)$, $0 \leq k < n/2$
- $A(\omega^k) = A_{\mathrm{even}}(\omega^k) - \omega^k A_{\mathrm{odd}}(\omega^k)$, $0 \leq k < n/2$

FFT Algorithm

```java
fft(n, a0, a1, ..., an-1) {
    if (n == 1) return a0
    (e0, e1, ..., en/2-1) ⊲ FFT(n/2, a0, a2, a4, ..., an-2)
    (d0, d1, ..., dn/2-1) ⊲ FFT(n/2, a1, a3, a5, ..., an-1)
    for k = 0 to n/2 - 1 {
        ωk = e2πik/n
        yk+n/2 = ek + ωk dk
        yk+n/2 = ek - ωk dk
    }
    return (y0, y1, ..., yn-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.

Running time. $T(2n) = 2T(n) + O(n) \Rightarrow T(n) = O(n \log n)$.

Recursion Tree

```
000 100 010 110 001 101 011 111
```

"bit-reversed" order

perfect shuffle

Coefficient representation
point-value representation
Point-Value to Coefficient Representation: Inverse DFT

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.

Claim. \( F_{n-1} \) and \( G_{n-1} \) are inverses.

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

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

Consequence. To compute inverse FFT, apply same algorithm but use \( e^{-\frac{2\pi ik}{n}} \) as principal \( n \)th root of unity (and divide by \( n \)).

Inverse FFT: Proof of Correctness

Claim. \( F_{n-1} \) and \( G_{n-1} \) are inverses.

Proof. 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 \( \omega^k = 1 \) we have:

\[ 1 + \omega^k + \omega^{2k} + \ldots + \omega^{(n-1)k} = 0 \]

\[ \Rightarrow \text{sums to } 0. \]

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.

Assume \( n \) is a power of 2.

Polynomial Multiplication

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

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.

Weighted Interval Scheduling

Notation. Label jobs by finishing time: \( f_1 \leq f_2 \leq \ldots \leq f_n \).
Def. \( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).
Ex: \( p(8) = 5, p(7) = 3, p(2) = 0 \).

Dynamic Programming: Binary Choice

Notation. \( OPT(j) = \) value of optimal solution to the problem consisting of job requests 1, 2, ..., \( j \).
- 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, ..., \( p(j) \)
- Case 2: \( OPT \) does not select job \( j \).
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., \( j-1 \)

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \{ v_j + OPT(p(j)), OPT(j-1) \} & \text{otherwise}
\end{cases}
\]

Weighted Interval Scheduling: Brute Force

Brute force algorithm.

Input: \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)
Sort jobs by finish times so that \( s_1 \leq s_2 \leq \ldots \leq s_n \).
Compute \( p(1), p(2), \ldots, p(n) \).
Compute-Opt(j) {
  if (j = 0)
    return 0
  else
    return max \( v_j + \) Compute-Opt(p(j)), Compute-Opt(j-1)
}

\[
T(n) = T(n-1) + T(p(n)) + O(1)
\]

\( T(1) = 1 \)

Key Insight: Do we really need to repeat this computation?
Weighted Interval Scheduling: Memoization

**Memoization.** Store results of each sub-problem in a cache; lookup as needed.

**Input:** $n, s_1, ..., s_n, f_1, ..., f_n, v_1, ..., v_n$.

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

Compute $p(1), p(2), ..., p(n)$

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

  $M$-Compute-Opt($j$) {
    if ($M[j]$ is empty)
      $M[j] = \max(v_j + M$-Compute-Opt($p(j))$, $M$-Compute-Opt($j-1))$
    return $M[j]$
  }

**Global array**

**Memoization.** Store results of each sub-problem in a cache; lookup as needed.

Weighted Interval Scheduling: Running Time

**Claim.** Memoized version of algorithm takes $O(n \log n)$ time.
- Sort by finish time: $O(n \log n)$.
- Computing $p(j)$: $O(n \log n)$ via sorting by start time.
- $M$-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 $\phi \leq n$.
  - (iii) increases $\phi$ by 1, at most $2n$ recursive calls.
- Overall running time of $M$-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.

**Run** $M$-Compute-Opt($n$)

**Run** Find-Solution($n$)

**Find-Solution($j$) {}**

- if ($j = 0$)
  - output nothing
- else if ($v_j + M[p(j)] > M[j-1]$)
  - print $j$
    - Find-Solution($p(j)$)
  - else
    - Find-Solution($j-1$)

- # of recursive calls $\leq n \implies O(n)$.

Weighted Interval Scheduling: Bottom-Up

**Bottom-up dynamic programming.** Unwind recursion.

**Input:** $n, s_1, ..., s_n, f_1, ..., f_n, v_1, ..., v_n$.

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

Compute $p(1), p(2), ..., p(n)$

**Iterative-Compute-Opt {}**

- $M[0] = 0$
- for $j = 1$ to $n$
  - $M[j] = \max(v_j + M[p(j)], M[j-1]$

Copyright 2000, Kevin Wayne