Algorithm Design and Analysis

LECTURE 16
Dynamic Programming
• (plus FFT Recap)

Adam Smith
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.

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

Discrete Fourier transform \hspace{1cm} Fourier matrix $F_n$
Fast Fourier Transform

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

Divide. Break polynomial up into even and odd powers.
- \( A_{\text{even}}(x) = a_0 + a_2 x + a_4 x^2 + ... + a_{n/2-2} x^{(n-1)/2} \).
- \( A_{\text{odd}}(x) = a_1 + a_3 x + a_5 x^2 + ... + a_{n/2-1} x^{(n-1)/2} \).
- \( A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2) \).

Conquer. Evaluate degree \( A_{\text{even}}(x) \) and \( A_{\text{odd}}(x) \) at set of all squares of \( n^{\text{th}} \) roots of unity
- \( v^0, v^1, ..., v^{n/2-1} \)

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

When \( n \) is even, there are only \( n/2 \) possible values of \( \omega^{2k} \)
fft(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
        y_{k+n/2} ← e_k - \omega^k d_k
    }

    return (y_0, y_1, ..., y_{n-1})
}
Theorem. FFT algorithm evaluates a degree n-1 polynomial at each of the \( n^{th} \) roots of unity in \( O(n \log n) \) steps. \( n \) is a power of 2

Running time. \( T(2n) = 2T(n) + O(n) \Rightarrow T(n) = O(n \log n) \).
**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 $a_0 + a_1 \times + \ldots + a_{n-1} \times^{n-1}$ that has given values at given points.

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

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 \( n \)).
Polynomial Multiplication

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

Integer multiplication. Given two n bit integers \( a = a_{n-1} \ldots a_1a_0 \) and \( b = b_{n-1} \ldots b_1b_0 \), compute their product \( c = a \times b \).

Convolution algorithm.
- Form two polynomials. \( A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} \)
  \( B(x) = b_0 + b_1x + b_2x^2 + \cdots + b_{n-1}x^{n-1} \)
- Note: \( a = A(2) \), \( b = B(2) \).
- Compute \( C(x) = A(x) \times B(x) \).
- Evaluate \( C(2) = a \times b \).
- Running time: \( O(n \log n) \) complex arithmetic steps.

Theory. [Schönhage-Strassen 1971] \( O(n \log n \log \log n) \) bit operations.
  [Martin Fuerer (Penn State) 2007] \( O(n \log n \ 2^{\log^* n}) \) bit operations.

Practice. [GNU Multiple Precision Arithmetic Library] GMP proclaims to be "the fastest bignum library on the planet." It uses brute force, Karatsuba, and FFT, depending on the size of \( n \).
Wrap-up Divide and Conquer

- Look for recursive structure

- Steps:
  - Divide: Break into smaller subproblems
  - Conquer: Solve subproblems
  - Combine: compute final solution

- Often helps to compute extra information in subproblems
  - e.g. # inversions: recursive call also sorts each piece

- If pieces independent, get parallelization
Chapter 5
Dynamic Programming
Design Techniques So Far

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

• **Divide-and-conquer.** Break up a problem into subproblems, solve subproblems, and combine solutions.

• **Dynamic programming.** Break problem into *overlapping* subproblems, 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 / decoding convolutional codes
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.
Fibonacci Sequence

Sequence defined by

- $a_1 = 1$
- $a_2 = 1$
- $a_n = a_{n-1} + a_{n-2}$

1, 1, 3, 5, 8, 13, 21, 34, ...

How should you compute the Fibonacci sequence?

Recursive algorithm:

Fib(n)
1. If $n = 1$ or $n = 2$, then
2. return 1
3. Else
4. a = Fib(n-1)
5. b = Fib(n-2)
6. return a+b

Running Time?
Review Question

Prove that the solution to the recurrence $T(n)=T(n-1)+T(n-2) + \Theta(1)$ is exponential in $n$. 
Computing Fibonacci Sequence Faster

Observation: Lots of redundancy! The recursive algorithm only solves n-1 different sub-problems

"Memoization": Store the values returned by recursive calls in a sub-table

Resulting Algorithm:

Fib(n)
1. If n =1 or n=2, then
2. return 1
3. Else
5. For i=3 to n
6. f[i] ← f[i-1]+f[i-2]
7. return a+b

Linear time!
(In fact, a log(n)-time algorithm exists.)
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.

![Diagram of weighted interval scheduling with jobs a, b, c, d, e, f, g, h at different times.](image-url)
Recall. *Greedy algorithm works if all weights are 1.*

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

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

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

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

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

Weighted Interval Scheduling: Brute Force

Brute force algorithm.
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.

\[ p(1) = 0, \quad p(j) = j-2 \]
weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a table; 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 \( p(1), p(2), \ldots, p(n) \)

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

\[
\text{M-Compute-Opt}(j) \{ \\
\quad \text{if (M}[j] \text{ is empty}) \\
\quad \quad \text{M}[j] = \max(w_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1)) \\
\quad \text{return M}[j] \\
\}
\]
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 $p(\cdot)$: $O(n \log n)$ via sorting by start time.

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

- **Overall running time of $M\text{-Compute-Opt}(n)$ 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 (vj + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

- # of recursive calls ≤ n ⇒ 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 $n$
        $M[j] = \max(v_j + M[p(j)], M[j-1])$
}