LECTURE 13
Dynamic Programming
• Fibonacci
• Weighted Interval Scheduling
Design Techniques So Far

- **Greedy.** Build up a solution incrementally, myopically optimizing some local criterion.
- **Recursion / divide & 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. \textbf{If} \ n = 1 \textbf{ or } n = 2, \textbf{ then}  
2. \hspace{1cm} \textbf{return} \ 1  
3. \hspace{1cm} \textbf{Else}  
4. \hspace{1cm} a = \text{Fib}(n-1)  
5. \hspace{1cm} b = \text{Fib}(n-2)  
6. \hspace{1cm} \textbf{return} \ a + b

Running Time?
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:**

```plaintext
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 f[n]
```

Linear time, if integer operations take constant time
Computing Fibonacci Sequence Faster

Observation: Fibonacci recurrence is linear

\[
\begin{pmatrix}
a_n \\
a_{n-1}
\end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_{n-1} \\
a_{n-2} \end{pmatrix} = \cdots = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} \begin{pmatrix} a_2 \\
a_1 \end{pmatrix}
\]

Can compute \( A^n \) using only \( O(\log n) \) matrix multiplications; each one takes \( O(1) \) integer multiplications and additions.

Total running time? \( O(\log n) \) integer operations. Exponential improvement!

Exercise: how big an improvement if we count bit operations?

Multiplying \( k \)-bit numbers takes \( O(k \log k) \) time.

How many bits needed to write down \( a_n \)?
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.
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 with weights.
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 \( j \).

= largest \( i \) such that \( f_i \leq s_j \)

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 \( \{ p(j) + 1, p(j) + 2, ..., j - 1 \} \)
  - 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

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

Brute force algorithm.

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

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

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))
}

**Worst case:**
\( T(n) = T(n-1) + T(n-2) + (\ldots) \)
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.

\[
p(1) = 0, \ p(j) = j-2
\]
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 \( p(1), p(2), \ldots, p(n) \)

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

\[
\text{M-Compute-Opt}(j) \{ \\
\quad \text{if (M[j] is empty)} \\
\quad \quad \text{M}[j] = \max(w_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1)) \\
\quad \text{return M}[j]
\}
\]
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 repeated binary search.

- $M\text{-Compute-Opt}(j)$: 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

- Case (ii) occurs at most $n$ times $\Rightarrow$ at most $2n$ recursive calls overall

- 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.
**Equivalent algorithm: 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 \( p(1), p(2), \ldots, p(n) \) how fast?

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

Total time = (sorting) + \( O(n) = O(n \log(n)) \)
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 ≤ n ⇒ O(n).