Lecture 28
Backtracking and Dynamic Programming
• Fibonacci
• Weighted Interval Scheduling

Adam Smith
Advanced recursion: Backtracking

• So far: divide-and-conquer

• Not all recursive algorithms have as nice as structure as “divide and conquer”

• Coming up:
  – Backtracking
  – Dynamic programming
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]
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: \( p(8) = 5, p(7) = 3, p(2) = 0 \).
Combinatorial Optimization

WIS is an example of “combinatorial optimization”

- **Optimization problem:**
  - Set of feasible solutions
    - WIS: nonoverlapping subsets of jobs
  - Cost function
    - WIS: sum of weights

- **Goal of the problem:** find a feasible solution that minimizes (or maximizes) the cost function
  - WIS: Find a nonoverlapping subset of intervals of maximum weight
Recursive Formulation: 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 $1, 2, ..., j-1$

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

Backtracking 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 \( f_1 \leq f_2 \leq \ldots \leq f_n \).

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: Backtracking

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems ⇒ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.

Notice that the same subproblem occurs many times in this tree!
Alternative Backtracking Approach

One can approach the WIS problem in a different way

• Write a program that enumerates all possible subset of intervals

• For each candidate subset, check if it has overlap. If not, compute its value and compare the to best value seen so far.

• For more nice examples of backtracking algorithms, see Jeff Erickson’s lecture notes:
  • http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/
  • (See Chapter 3, “Backtracking”)
Dynamic Programming

• Some recursive algorithms can be sped up dramatically by taking advantage of overlap among subproblems

• Two versions:
  – “memoize”, or remember recursive calls made so far
  – Build a table “bottom-up” of answers for the subproblems
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 $n$.

Idea: $T(n) > 2T(n-2) + 1$.

Thus, $T(n) > 2^{n/2}$ (Prove this using recursion tree)
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
  - “Bottom-up” algorithm: compute the entries of the table in logical order, from the base case to the top

Bottom-up 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 a+b
```

Linear time (actually, more like quadratic since integer lengths grow)
In fact, an algorithm that uses log(n)-multiplications (total time O(nlog n)) algorithm exists, based on matrix exponentiation
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.
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 $p(1), p(2), \ldots, p(n)$

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

$M[0] = 0$

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

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

Bottom-up dynamic programming. Unwind recursion.

Input: n, s₁,...,sₙ, f₁,...,fₙ, v₁,...,vₙ

Sort jobs by finish times so that f₁ ≤ f₂ ≤ ... ≤ fₙ.

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

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