Lecture 19: Dynamic Programming I:
Memoization, Fibonacci, Crazy Eights, Guessing

Lecture Overview

- Fibonacci Warmup
- Memoization and subproblems
- Shortest Paths
- Crazy Eights
- Guessing Viewpoint

Readings

CLRS 15

Dynamic Programming (DP)

Big idea: hard yet simple

- Powerful algorithmic design technique
- Large class of seemingly exponential problems have a polynomial solution ("only") via DP
- Particularly for optimization problems (min / max) (e.g., shortest paths)

* DP ≈ "controlled brute force"
* DP ≈ recursion + re-use

Fibonacci Numbers

\[ F_1 = F_2 = 1; \quad F_n = F_{n-1} + F_{n-2} \]

Naive Algorithm

follow recursive definition

\[
\text{fib}(n): \\
\text{if } n \leq 2: \text{ return 1} \\
\text{else return fib}(n - 1) + \text{fib}(n - 2) \\
\implies T(n) = T(n - 1) + T(n - 2) + O(1) \approx \phi^n \\
\geq 2T(n - 2) + O(1) \geq 2^{n/2}
\]

EXPOENTIAL - BAD!
Simple Idea

memoize

```python
memo = {}  # Initialize memo as an empty dictionary
fib(n):
    if n in memo: return memo[n]  # If n is already computed, return its value
    else:
        if n ≤ 2 : f = 1  # Base cases
        else:
            f = fib(n − 1) + fib(n − 2)  # Recursive computation
        memo[n] = f  # Store the computed value in memo
        return f
```

**Figure 1: Naive Fibonacci Algorithm**

\[ T(n) = T(n − 1) + O(1) = O(n) \]

[Side Note: There is also an \( O(\log n) \)-time algorithm for Fibonacci, via different techniques]

* DP \( \approx \) recursion + memoization
  * remember \( \text{memoize} \) previously solved “subproblems” that make up problem
    - in Fibonacci, subproblems are \( F_0, F_1, \ldots, F_n \)
  * if subproblem already solved, re-use solution
* \( \implies \) time = \( \frac{\text{\# of subproblems}}{\text{time/subproblem}} \)
  * - in fib: \( \frac{\text{\# of subproblems}}{\text{time/subproblem}} \) is \( O(n) \) and time/subproblem is \( O(1) \) - giving us a total time of \( O(n) \).
Shortest Paths

- Recursive formulation:
  \[ \delta(s, t) = \min \{ w(s, v) + \delta(v, t) \mid (s, v) \in E \} \]

- does this work with memoization?
  no, cycles \(\implies\) infinite loops (see Figure 2).

\[ \text{Figure 2: Shortest Paths} \]

- in some sense necessary for neg-weight cycles

- works for directed acyclic graphs in \(O(V + E)\)
  (recursion effectively DFS/topological sort)

- trick for shortest paths: removing cyclic dependency.
  
  \[ \delta_k(s, t) = \text{shortest path using} \leq k \text{ edges} \]
  \[ = \min \{ \delta_{k-1}(s, t) \} \cup \{ w(s, v) + \delta_{k-1}(v, t) \mid (s, v) \in E \} \]
  
  \[ \ldots \text{except} \ \delta_k(t, t) = \phi, \ \delta_\phi(s, t) = \infty \text{ if } s \neq t \]
  
  \[ \delta(s, t) = \delta_{\phi-1}(s, t) \text{ assuming no negative cycles} \]

  \[ \implies \text{time} = O(n^3) \text{ for } s, t, k \text{-- really } O(n^2) \]

  \[ = O(V \cdot \sum_{V} \deg(V)) = O(VE) \]

* Subproblem dependency should be acyclic.
Crazy Eights Puzzle

- given a sequence of cards \( c[0], c[1], \ldots, c[n-1] \)
  e.g., \( 7\heartsuit, 6\heartsuit, 7\diamondsuit, 3\diamondsuit, 8\clubsuit, J\spadesuit \)
- find longest left-to-right “trick” (subsequence)

\[
c[i_1], c[i_2], \ldots, c[i_k] \quad (i_1 < i_2 < \cdots < i_k)
\]
where \( c[i_j] \) & \( c[i_{j+1}] \) “match” for all \( j \)
have some suit or rank or one has rank 8

- recursive formulation:

\[
\text{trick}(i) = \text{length of best trick starting at } c[i] \\
= 1 + \max(\text{trick}(j) \text{ for } j \text{ in } \text{range}(i+1, n) \text{ if } (c[i], c[j]))
\]
best \( = \max(\text{trick}(i) \text{ for } i \text{ in } \text{range}(n)) \)

- memoize: \( \text{trick}(i) \) depends only on \( \text{trick}(>i) \)

\[
\implies \text{time} = \frac{n}{O(n)} \cdot \frac{O(n)}{O(n)}
= O(n^2) \quad \text{(to find actual trick, trace through max’s)}
\]

“Guessing” Viewpoint

- what is the first card in best trick? guess!
i.e., try all possibilities & take best result
  - only \( O(n) \) choices

- what is next card in best trick from \( i \)? guess!
  - if you pretend you knew, solution becomes easy (using other subproblems)
  - actually pay factor of \( O(n) \) to try all

- * use only small \( \frac{\text{choices/guesses}}{\text{per subproblem}} \approx O(1) \)

\[
\text{poly}(n) \sim O(1)
\]