Menu

• New technique: Dynamic Programming
  ▪ Computing Fibonacci numbers – Warmup
  ▪ “Definition” of DP
  ▪ Crazy Eights Puzzle
Fibonacci Numbers

• Fibonacci sequence:
  - $F_0=0$, $F_1=1$
  - $F_n=F_{n-1}+F_{n-2}$

• How fast does $F_n$ grow?
  - $F_n = F_{n-1} + F_{n-2} \geq 2F_{n-2} \Rightarrow F_n = 2^{\Omega(n)}$

• How quickly can we compute $F_n$?
  (time measured in arithmetic operations)
\[ F_n = F_{n-1} + F_{n-2} \]

- Algorithm I: recursion
  ```python
  def naive_fibo(n):
      if n == 0: return 0
      elif n == 1: return 1
      else:
          return naive_fibo(n-1) + naive_fibo(n-2)
  ```

- Time \( O(F_n) \)
- Better algorithm?
\[ F_n = F_{n-1} + F_{n-2} \]

- **Algorithm II: memoization**
  
  \[
  \text{memo} = \{ \}
  \]

  \[
  \text{fibo}(i):
  \]
  
  if \( i \) in \( \text{memo} \): return \( \text{memo}[i] \)
  
  else if \( i = 0 \): return 0
  
  else if \( i = 1 \): return 1
  
  else:
    \( f = \text{fibo}(i-1) + \text{fibo}(i-2) \)
    \[ \text{memo}[i] = f \]
  
  return \( f \)

  return \( \text{fibo}(n) \)

- **Time? \( O(n) \)**

  - in the whole recursive execution, I will only go beyond this point, \( n \) times
    (since every time I do this, I fill in another slot in \( \text{memo[]} \))

  - hence, all other calls to \( \text{fibo}() \) act as reading an entry of an array
Dynamic Programming

• DP ≈ Recursion + Memoization

• DP works when:
  ▪ the solution can be produced by combining solutions of subproblems;  \( F_n = F_{n-1} + F_{n-2} \)
  ▪ the solution of each subproblem can be produced by combining solutions of sub-subproblems, etc;
  moreover…. \( F_{n-1} = F_{n-2} + F_{n-3} \quad F_{n-2} = F_{n-3} + F_{n-4} \)
  ▪ the total number of subproblems arising recursively is polynomial.

\[ F_1, F_2, \ldots, F_n \]
Dynamic Programming

• DP ≈ Recursion + Memoization
• DP works when:

  Optimal substructure
  The solution to a problem can be obtained by solutions to subproblems.

  \[ F_n = F_{n-1} + F_{n-2} \]

  moreover….

  Overlapping Subproblems
  A recursive solution contains a “small” number of distinct subproblems (repeated many times)

  \[ F_1, F_2, \ldots, F_n \]
Crazy 8s

- **Input:** a sequence of cards $c[0]...c[n-1]$.
- E.g., $7♣ 7♥ K♣ K♠ 8♥ ...$
- **Goal:** find the longest “trick subsequence” $c[i_1]...c[i_k]$, where $i_1 < i_2 < ... < i_k$.
- For it to be a trick subsequence, it must be that:
  - $\forall j$, $c[i_j]$ and $c[i_{j+1}]$ “match” i.e.
    - they either have the same rank,
    - or the same suit
    - or one of them is an 8
    - in this case, we write: $c[i_j] \sim c[i_{j+1}]$
- E.g., $7♣ K♣ K♠ 8♥$ is the longest such subsequence in the above example
Algorithm

• Let $\text{trick}(i)$ be the length of the longest trick subsequence that starts at card $c[i]$

• **Question:** How can I relate value of $\text{trick}(i)$ with the values of $\text{trick}(i+1), \ldots, \text{trick}(n)$?

• Recursive formula:

  $$\text{trick}(i) = 1 + \max_{j > i, c[i] \sim c[j]} \text{trick}(j)$$

• Maximum trick length:

  $$\max_{i} \text{trick}(i)$$
Implementations

Recursive

- memo = {}
- trick(i):
  - if i in memo: return memo[i]
  - else if i=n-1: return 1
  - else
    - f := 1+\max_{j>i, c[i] \sim c[j]} trick(j)
    - memo[i] := f
    - return f
- call trick(0)
- return maximum value in memo
Implementations (cont.)

Iterative

```plaintext
memo = {}
for i=n-1 downto 0
  memo[i] = 1 + \max_{j>i, c[i] \sim c[j]} \text{memo}[j]
return maximum value in memo
```

Runtime: $O(n^2)$
Dynamic Programming

• DP ≈ Recursion + Memoization
• DP works when:

  **Optimal substructure**
  An solution to a problem can be obtained by solutions to subproblems.
  \[
  \text{trick}(i) = 1 + \max_{j > i, c[i] \sim c[j]} \text{trick}(j)
  \]

 moreover….

  **Overlapping Subproblems**
  A recursive solution contains a “small” number of distinct subproblems (repeated many times)
  \[
  \text{trick}(0), \text{trick}(1), \ldots, \text{trick}(n-1)
  \]
Menu

• New technique: Dynamic Programming
  ▪ Computing Fibonacci numbers – Warmup
  ▪ “Definition” of DP
  ▪ Crazy Eights Puzzle
  ▪ Next Time: all-pairs shortest paths
All-pairs shortest paths

- **Input:** Digraph $G = (V, E)$, where $|V| = n$, with edge-weight function $w : E \to \mathbb{R}$.
- **Output:** $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.

**Assumption:** No negative-weight cycles
Dynamic Programming Approach

• Consider the $n \times n$ matrix $A = (a_{ij})$, where $a_{ij} = w(i,j)$ if $(i,j) \in E$, and define
  
  - $d_{ij}^{(m)} = \text{weight of a shortest path from } i \text{ to } j \text{ that uses at most } m \text{ edges}$

  **Claim:** We have
  
  $d_{ij}^{(0)} = 0$, if $i = j$, and $\infty$, if $i \neq j$;

  and for $m = 1, 2, \ldots, n-1$,
  
  $d_{ij}^{(m)} = \min_k \{d_{ik}^{(m-1)} + a_{kj}\}$. 

Proof of Claim

d_{ij}^{(m)} = \min_k \{d_{ik}^{(m-1)} + a_{kj}\}

for \(k \leftarrow 1\) to \(n\)

if \(d_{ij} > d_{ik} + a_{kj}\)

\(d_{ij} \leftarrow d_{ik} + a_{kj}\)

Relaxation
Dynamic Programming Approach

• Consider the $n \times n$ matrix $A = (a_{ij})$, where $a_{ij} = w(i,j)$ if $(i,j) \in E$, and define
  - $d_{ij}^{(m)} = \text{weight of a shortest path from } i \text{ to } j \text{ that uses at most } m \text{ edges}$

Claim: We have

$$d_{ij}^{(0)} = 0, \text{ if } i = j, \text{ and } \infty, \text{ if } i \neq j;$$

and for $m = 1, 2, \ldots, n-1$,

$$d_{ij}^{(m)} = \min_k \{d_{ik}^{(m-1)} + a_{kj}\}.$$

Time? $O(n^4)$ - similar to $n$ runs of Bellman-Ford