Dynamic Programming and Greedy: Review

Examples in lectures and labs

Dynamic programming:

- Playing a board game
- Rod cutting
- Knapsack
- Pharmacist
- Fibonacci
- Longest TRUE interval
- LCS (longest common subsequence)
- Optional: Robbing a house
- Optional: Playing a game
- This week: Longest increasing subsequence
- This week: Unbounded knapsack
- Optional Skis and skiers

Greedy:

- Activity selection
- Guarding a museum
- A different pharmacist problem (all bottles have same cost)
- Optional: Matching points on a line
- Optional: Greedy skis and skiers

1 Rod cutting

- The problem: Given a rod of length n and a table of prices $p[i]$ for $i = 1, 2, 3, ..., n$, determine the maximal revenue obtainable by cutting up the rod and selling the pieces.
- Notation and choice of subproblem: We denote by $maxrev(x)$ the maximal revenue obtainable by cutting up a rod of length x. To solve our problem we call $maxrev(n)$.
- Recursive definition of $maxrev(n)$:

  maxrev(x)

  if $(x \leq 0)$: return 0

  For i = 1 to n: compute $p[i] + \texttt{maxrev}(x - i)$ and keep track of max

  RETURN this max

- Correctness: see notes.
- Dynamic programming solution, top-down with memoization:

  We create a table of size $[0..n]$, where $table[i]$ will store the result of $maxrev(i)$. We initialize all entries in the table as 0. To solve the problem, we call $maxrevDP(n)$.

  maxrevDP(x)

  if $(x \leq 0)$: return 0

  IF $table[x] \neq 0$: RETURN $table[x]$

  For i = 1 to n: compute $p[i] + \texttt{maxrevDP}(x - i)$ and keep track of max

  $table[x] = \max$

  RETURN $table[x]$

- Dynamic programming, bottom-up:

  maxrevDP_iterative(x)

  create $table[0..n]$ and initialize $table[i] = 0$ for all i

  for $(k = 1; k \leq n; k++)$

  for $(i = 1; i \leq k; i++)$

  set $table[k] = \texttt{max}\{table[k], p[i] + table[k - i]\}$

  RETURN $table[n]$

- Analysis: $O(n^2)$
- Computing full solution:

2 $0-1$ Knapsack

- The problem: We are given a knapsack of capacity W and a set of n items; an each item i, with $1 \leq i \leq n$, is worth $v[i]$ and has weight $w[i]$ pounds. Assume that weights $w[i]$ and the total weight W are integers. The goal is to fill the knapsack so that the value of all items in the knapsack is maximized.

- Notation and choice of subproblem: Denote by $optknapsack(k, w)$ the maximal value obtainable when filling a knapsack of capacity w using items among items 1 through k. To solve our problem we call $optknapsack(n, W)$.

- Recursive definition of $optknapsack(k, w)$:

```
optknapsack(k, w)

    if (w ≤ 0) or (k ≤ 0) : return 0 //basecase

    IF (weight[k] ≤ w): with = value[k] + optknapsack(k − 1, w − weight[k])

    ELSE: with = 0

    without = optknapsack(k − 1, w)

    RETURN max { with, without }
```

- Correctness: see notes.

- Dynamic programming solution, top-down with memoization: We create a table $table[1..n][1..W]$, where $table[i][w]$ will store the result of optknapsack(i, w). We initialize all entries in the table as 0. To solve the problem, we call $optknapsackDP(n, W)$.

```
optknapsackDP(k, w)

    if (w ≤ 0) or (k ≤ 0):: return 0

    IF (table[k][w] ≠ 0): RETURN table[k][w]

    IF (w[k] ≤ w): with = v[k] + optknapsackDP(k − 1, w − w[k])

    ELSE: with = 0

    without = optknapsackDP(k − 1, w)

    table[k][w] = max { with, without }

    RETURN table[k][w]
```

- Dynamic programming, bottom-up:

optknapsackDP_iterative

create table[0..n][0..W] and initialize all entries to 0

for $(k = 1; k < n; k++)$

for $(w = 1; w < W; w++)$

$with = v[k] + table[k-1][w - w[k]]$

$without = table[k-1][w]$

$table[k][w] = \max$ { with, without }

RETURN $table[n][W]$

- Analysis: $O(n \cdot W)$
- Computing full solution:

3 Pharmacist

- The problem: A pharmacist has W pills and n empty bottles. Bottle i can hold $p[i]$ pills and has an associated cost $c[i]$. Given W, $p[1..n]$ and $c[1..n]$, find the minimum cost for storing the pills using the bottles.

- Notation and choice of subproblem: Denote by $MinPill(i, j)$ the minimum cost obtainable when storing j pills using bottles among 1 through i. To solve our problem we call $minPill(n, W)$.

- Recursive definition of $minPill(i, j)$:

  minPill(i, j)

  if $(j \leq 0)$: return 0 //no pills left

  IF $(i == 0$ and $j > 0)$: return ∞ //have pills, but no bottles, sol not possible

  with $= c[i] + \texttt{minPill}(i-1, j-p[i])$

  without $=\texttt{minPill}(i-1, j)$

  RETURN min { with, without }

- Correctness:

- Dynamic programming solution, top-down with memoization: We create $table[1..n][1..W]$, where $table[i][j]$ will store the result of $minPill(i, j)$. We initialize all entries in the table as 0. To solve the problem, we call $minPillDP(n, W)$.

  minPillDP(i, j)

  if $(j \leq 0)$: return 0 //no pills left

  IF $(i == 0$ and $j > 0)$: return ∞ //have pills, but no bottles, sol not possible

  IF $(table[i][j] \neq 0)$: RETURN $table[i][j]$

  with $= c[i] + \texttt{minPillDP}(i-1, j-p[i])$

  without $=\texttt{minPillDP}(i-1, j)$

  $table[i][j] =$ min { with, without }

  RETURN $table[i]j]$

- Dynamic programming, bottom-up:

minPill_iterative

create table[0..n][0..W] and initialize all entries to 0

for $(i = 1; i < n; i++)$

for $(j = 1; j < W; j++)$

with = $c[i] + table[i-1][j-p[i]]$

without = $table[i-1][j]$

$table[i][j]$ = min { with, without }

RETURN $table[n][W]$

- Analysis: $O(n \cdot W)$
- Computing full solution:

4 Longest True interval

- The problem: Suppose we are given an array $A[1..n]$ of booleans. We want to find the longest interval $A[i..j]$ such that every element in the interval is true – in other words, $A[i], A[i+1], .., A[j]$ are all true.

- Notation and choice of subproblem: Denote by $G(x)$ to be the length of the longest suffix[1] of $A[1..x]$ that is all true. In other words, $G(x)$ is the largest integer l such that $A[x-l+1], A[x-l+2], .., A[x]$ are all true, or 0 if $A[x]$ is false.

- Recursive definition of $G(x)$:

```
G(x)
    IF (x == 1): return A[1]
    else
        IF A[x] == False: return 0 else return 1 + G(x − 1)
```

- Correctness:

- Dynamic programming solution, top-down with memoization: We create $table[0..n]$, where $table[i]$ will store the result of $G(i)$. We initialize all entries in the table as 0. To solve the problem, we call $G\_DP(0), G\_DP(1), G\_DP(2), ...$ to fill the table and then return the max element in this table.

```
G_DP(x)(x)
    IF (x == 1): return A[1]
    else
        IF (table[x] ≠ 0): RETURN table[x]
        IF A[x] == False: answer= 0 else answer= 1 + G_DP(x − 1)
        table[x] = answer
        return answer
```

- Dynamic programming, bottom-up:

- Analysis: $O(n)$

- Computing full solution:

[1] An array $B[1..m]$ is a suffix of an array $A[1..n]$ if $A[n-k] = B[m-k]$ for $0 \le k < m$

5 LCS

- The problem: Given two arrays $X[1..n]$ and $Y[1..m]$, find their longest common subsequence.
- Notation and choice of subproblem: Denote by $c(i, j)$ the length of the LCS of X_i and Y_j, where X_i is the array consisting of the first i elements of X, and Y_j is the array consisting of the first j elements of Y. To solve the problem, we call $c(n, m)$
- Recursive definition of $c(i, j)$:

```
c(i, j)
    IF (i == 0 or j == 0): return 0
    else
        IF X[i] == Y[j]: return 1 + c(i − 1, j − 1)
        Else: return max{c(i − 1, j), c(i, j − 1)}
```

- Correctness:
- Dynamic programming solution, top-down with memoization: We create $table[0..n][0..m]$, where $table[i][j]$ will store the result of $c(i, j)$. We initialize all entries in the table as 0 and call $c\_DP(n, m)$.

```
c_DP(i, j)
    IF (i == 0 or j == 0): return 0
    else
        IF (table[i][j] ≠ 0): RETURN table[i][j]
        IF X[i] == Y[j]: answer 1 + c_DP(i − 1, j − 1)
        Else: answer= max{c(i − 1, j), c_DP(i, j − 1)}
        table[x] = answer
        return answer
```

- Dynamic programming, bottom-up:
- Analysis: $O(m \cdot n)$
- Computing full solution: