Dynamic Programming 2

Knapsack Problem

1. Input: n objects with integer weights w1,...,wn and integer values v1,...,vn
    - Total capacity B
2. Goal: Find subset S of objects that:
    - Fit in backpack with total weight <= B
        - Sum of weights in subset <= B
    - Maximizes total value
        - Sum of values is maximal

Knapsack Problem Variants

1. Two versions:
    - One copy of each object - without repetition
    - Unlimited supply - with repetition
        - Subset is a multiset

Greedy Algorithm

1. Consider the following problem with a total capacity B = 22:

Object	Value	Weight
1	15	15
2	10	12
3	8	10
4	1	5

2. Optimal weight is 22 with a value of 18 using objects 2 and 3
3. Greedy: Sort objects by ri = vi / wi = value per unit of weight
    - r1 > r2 > r3 > r4
    - Greedy approach would use objects 1 and 4 for a total weight of 20 with a value of 16

Knapsack: No repetition

1. Define subproblem:
    - K(i) = max value achievable using a subset of objects 1, ..., i
2. Express K(i) in terms of K(1), ..., K(i-1)

Knapsack: Recurrence 1

1. Example:
    - B = 22
    - Values = [15, 10, 8, 1]
    - Weights = [15, 12, 10, 5]
    - K = [15, 15, 18, 18]
        - We can't obtain K = 3 with our current subproblem because we're only tracking the optimal solution
            * Need to limit capacity available
        - We need to take a suboptimal solution when K = 2
            * Take optimal solution where capacity <= B - wi

Knapsack: Subproblem 2

1. To solve K(i): Couldn't use K(i-1)
   - Needed K(i-1) with the additional restriction total weight <= B - wi
2. Two parameters: i and b
   - i specifies the prefix of the object to consider
   - b specifies the total weight available
3. Subproblem definition:
   - For i and b where 0 <= i <= n and 0 <= b <= B:
     - Let K(i,b) = max value achievable using a subset of objects 1,…,i and total weight <= b
   - Our goal: Compute k(n,B)

Knapsack: Recurrence 2

1. If wi <= b:
   - then K(i,b) = max{vi + K(i-1,b-wi), K(i-1,b)}
   - else K(i,b) = K(i-1,b)
2. Base cases: K(0,b) = 0 and K(i,0) = 0
3. 2D table filled row by row

Knapsack: DP Pseudocode

```
KnapsackNoRepeat(w1,...,wn, v1,...,vn, B):
    for b = 0 -> B:
        K(0,b) = 0
    for i = 1 -> n:
        K(i,0) = 0
    for i = 1 -> n:
        for b = 1 -> B:
            if wi <= b:
                K(i,b) = max(vi + K(i-1,b-wi), K(i-1,b))
            else:
                K(i,b) = K(i-1,b)
    return K(n,B)
```

1. Time complexity:
   - O(nB)

Knapsack in Poly-time?

1. Is the algorithm efficient:
   - Efficient: Is the running time polynomial in the input size?
   - Answer: No
     - To represent the **number** B takes space O(log(B))
     - Goal: Running time poly(n, log(B))
   - Knapsack problem is NP-complete
     - If we design a polynomial-time algorithm for this problem, then every problem in NP will have a polynomial time algorithm

Knapsack Repetition

1. Unlimited supply: Can use an object as many times as we'd like
2. Define subproblem:
   - K(i,b) = max value attainable from a multiset of objects {1,…,i} with weight <= b

Knapsack2: Recurrence

1. K(i,b) = max{K(i-1,b), vi + K(i,b-wi)}
    - Two scenarios:
        - Include no more copies of object i
        - Include another copy of object i
    - We don't use i-1 in vi + K(i,b-wi) to because we can reuse the object
2. This is a valid recurrence because the necessary previous solutions will already be computed if we go row by row
3. Time complexity: O(nB)

Knapsack2: Recap

1. When we get a solution with a 2D or 3D table, it's useful check if the solution can be simplified
    - Faster, less space, simpler, ...
2. Point of parameter i in the original problem is to track which objects are included or not
    - We don't need to track this for the repeated knapsack problem

Knapsack2: Simpler Subproblem

1. Define subproblem:
    - For b where 0 <= b <= B:
        - K(b) = max value attainable using weight <= b
2. Write recurrence:
    - Try all possibilities for last object to add
    - K(b) = max{vi + K(b-wi): 1 <= i <= n, wi <= b}

Knapsack2: Pseudocode

```
KnapsackRepeat(w1,...,wn, v1,...,vn, B):
    for b = 0 -> B:
        K(B) = 0
        for i = 1 -> n:
            if wi <= b and K(b) < vi + K(b-wi):
                K(b) = vi + K(b-wi)
    return K(B)
```

Knapsack2: Running Time

1. Time complexity: O(nB)
    - Same runtime as original solution, but space is lesser and solution is simpler

Knapsack2: Traceback

1. What if we want to output the multiset?
    - Make a separate array S and initialize it to 0
    - When we update the solution, set S(b) = i
    - Can use S to backtrack and get the multiset
        - Backtrack is similar to LCS

```
KnapsackRepeat(w1,...,wn, v1,...,vn, B):
    for b = 0 -> B:
        K(B) = 0
        S(b) = 0
        for i = 1 -> n:
            if wi <= b and K(b) < vi + K(b-wi):
```

```
            K(b) = vi + K(b-wi)
            S(b) = i
return K(B)
```

Chain Matrix Multiply

1. Goal: Compute A * B * C * D where A, B, C, D, are matrices most efficiently
    - A is 50 x 20
    - B is 20 x 1
    - C is 1 x 10
    - D is 10 x 100

Order of Operation

1. Which parenthesization?
    - ((A x B) x C) x D
    - (A x B) x (C x D)
    - (A x (B x C) x D)
    - A x (B x (C x D))

Cost for Matrix Multiply

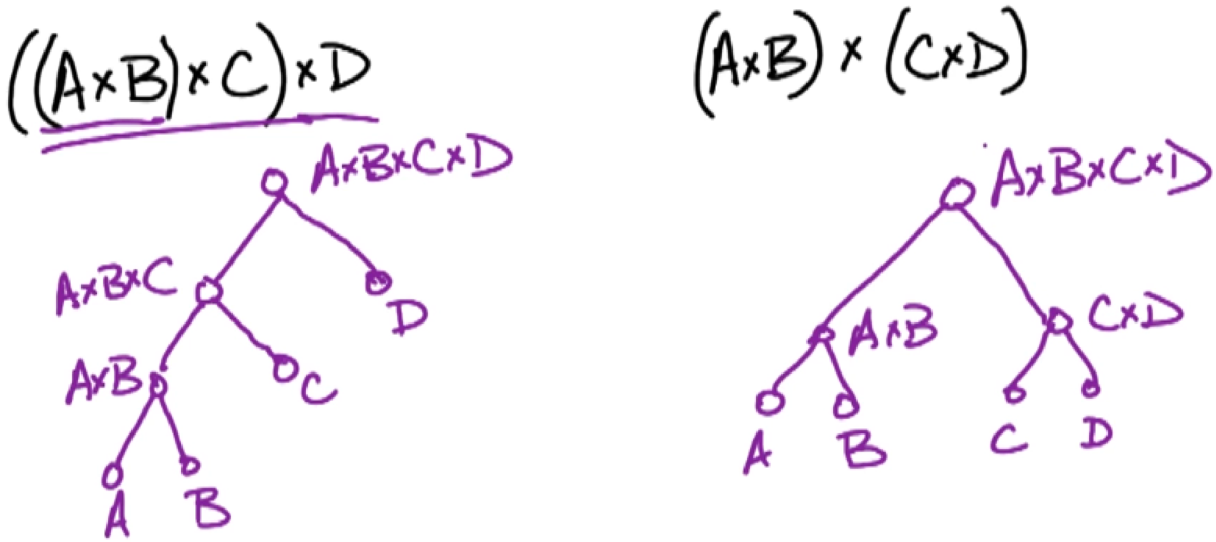
1. Take W of size a x b and Y of size b x c
    - Z = W x Y is of size a x c
        - acb multiplications
        - ac(b-1) additions
    - Cost is abc

General Problem

1. For n matrics A_1, A_2, …, A_n where A_i is m_{i-1} x m_i
2. Goal: What is the minimum cost for computing A_1 x A_2 x … x A_n
3. Input: m_0, m_1, …, m_n
4. Goal: Find minimum cost for computing A_1 x A_2 x … x A_n

Graphical View

1. Represent as binary tree



Binary Tree Representation

Chain Multiply: Prefixes

1. Define subproblem:
    - Let C(i) = min cost for computing A_1 x A_2 x ... x A_i
    - This will cause us to look at suffixes instead of prefixes
        - This means intermediate computations correspond to substrings

Chain Multiply: Substrings

1. For i and j where 1 <= i <= j <= n
    - Let C(i,j) = min cost for computing A_1 x A_2 x ... x A_i
2. Recurrence for C(i,j):
    - C(i,i) = 0
    - Only computing where j >= i which is the upper diagonal of the matrix

Chain Multiply: Recurrence

1. To compute C(i,j):
    - Total cost for split at index l:
        - Root: m_{i-1} * m_l * m_j
        - Left subtree: C(i,l)
        - Right subtree: C(l+1,j)

Chain Multiply: Summary

1. To compute C(i,j):
    - C(i,j) = min{C(i,l) + C(l+1,j) + m_{i-1} * m_l * m_j : i <= l <= j-1}

Filling the Table

1. C(i,i+1) uses C(i,i) and C(i+1,i+1)
    - C(i,i) is the base case (0)
    - Width S = j - i
    - S = 0 to n-1

Chain Multiply: DP Pseudocode

```
ChainMultiply(m0, m1, ..., mn):
    for i = 1 to n:
        C(i,i) = 0
    for s = 1 to n-1:
        for i = 1 to n-s:
            j = i + s
            C(i,j) = Inf
            for l = i to j-1:
                cur = mi-1 * ml * mj + C(i,l) + C(l+1,j)
                if C(i,j) > cur:
                    C(i,j) = cur
    return C(1,n)
```

1. Time complexity: O(n^3)
    - Instead of using prefixes we had to use substrings
        - Have to start at diagonal and work our way up instead of going row by row

DP2: Practice Problems

1. Practice Problems:
    - DPV 6.17 (change making)
    - DPV 6.18 (change making)
    - DPV 6.19 (change making)
    - DPV 6.20 (optimal BST)
    - DPV 6.7 (palindrome subsequence)
        - Also try variant of palindrome substring
2. Try prefixes first, then substrings
    - If you use substrings, go back and look at whether using prefixes is possible