Dynamic Programming

CMPUT 275 - Winter 2018

University of Alberta

Outline

- Longest Common Substring
- Maximum Contiguous Sum
- Edit Distance
- Dice Throw

5 Steps to Solve Dynamic Programming Problems

1. Define subproblems
2. Guess part of the solution
3. Related subproblem solutions
4. Recurse and memoize (top-down approach) or
   create a table (bottom-up approach)
5. Solve the original problem by combining solutions of the
   subproblems

5 Steps to Solve Dynamic Programming Problems

1. Define subproblems
2. Guess part of the solution
3. Related subproblem solutions
4. Recurse and memoize (top-down approach) or create a table (bottom-up approach)
5. Solve the original problem by combining solutions of the subproblems

Running Time Analysis Template
Running time $=$ (# of distinct subproblems we need to solve) $\times$ (computation required to solve each subproblem) $+$ (computation required to combine solutions to solve the original problem, this is outside the function that is called recursively)

Longest Common Substring

We are given two strings x and y. We want to find the length of the longest common substring of these two strings.

For example, if $x=$"joyful" and $y=$"enjoyed", then the longest common substring of x and y is 3.

Longest Common Substring

We are given two strings x and y. We want to find the length of the longest common substring of these two strings.

For example, if x= "joyful" and y= "enjoyed", then the longest common substring of x and y is 3.

We first solve a different problem which is to determine the longest common suffix of given strings

Longest Common Substring

Subproblem: $LCSuffix(i, j)$ longest common suffix of $x[: i]$ and $y[: j]$
Guess: Nothing

Recursive Relation:

$$LCSuffix(i,j) = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ 1 + LCSuffix(i-1, j-1), & \text{if } x[i-1] = y[j-1] \\ 0, & \text{otherwise}\,. \end{cases}$$

Running Time Analysis: Number of distinct subproblems: $O(n \cdot m)$ where m is $len(x)$ and n is $len(y)$.
Time per subproblem: $O(1)$
Total running time: $O(n \cdot m)$

Original Problem: $\max_{1 \le i \le m, 1 \le j \le n} LCSuffix(i, j)$

Top-Down Approach

```
def LCSuffix(s1, s2, i, j, memo=None):
    if memo is None:
        memo = {}
    if not (i, j) in memo:
        if i == 0 or j == 0:
            longest_suffix = 0
        elif s1[i-1] == s2[j-1]:
            longest_suffix = 1 + LCSuffix(s1, s2, i-1, j-1, memo)
        else:
            longest_suffix = 0
        memo[(i, j)] = longest_suffix
    return memo[(i, j)]


def LCSubstr(s1, s2):
    memo = {}
    longest_substr = 0
    for i in range(1, len(s1)+1):
        for j in range(1, len(s2)+1):
            length = LCSuffix(s1, s2, i, j, memo)
            longest_substr = max(longest_substr, length)
    return longest_substr
```

Bottom-Up Approach

```
def LCSBottomUp(s1, s2, m, n):
    table = [[0 for k in range(n+1)] for l in range(m+1)]
    longest_substr = 0
        for i in range(m + 1):
        for j in range(n + 1):
            if (i == 0 or j == 0):
                table[i][j] = 0
            elif (s1[i-1] == s2[j-1]):
                table[i][j] = table[i-1][j-1] + 1
                longest_substr = max(longest_substr, table[i][j])
            else:
                table[i][j] = 0
    return longest_substr
```

Maximum Contiguous Sum

We are given a list of N numbers, denoted by *seq*. We want to find the contiguous subsequence with the largest sum and return this value.

For example, if our sequence is "1, -2, 3, 2", then the contiguous subsequence with the largest sum is "3, 2" and hence the maximum sum is 5. Note that "1, 3, 2" is not a contiguous subsequence of our sequence.

Maximum Contiguous Sum

Input: A list *seq* of length N

Goal: Find the largest sum of a contiguous subsequence of *seq*

Maximum Contiguous Sum

Input: A list *seq* of length N

Goal: Find the largest sum of a contiguous subsequence of *seq*

Subproblem: $MCS(i)$ the maximum sum of the contiguous subsequences of *seq* that end with $seq[i]$

Guess: 2 possibilities: append the last element to the list which currently has the maximum contiguous sum or start a new list that includes the last item only

Maximum Contiguous Sum

Recursive Relation:

$$MCS(i) = \begin{cases} \max(0, seq[0]), & \text{if } i = 0 \\ \max(seq[i], MCS(i-1) + seq[i]), & \text{otherwise}. \end{cases}$$

Running Time Analysis: Number of distinct subproblems: $O(N)$
Time per subproblem: $O(1)$
Total running time: $O(N)$

Original Problem: $\max_{0 \leq i \leq N-1} MCS(i)$

Top-Down Approach

```
def max_contig_sum(seq):
    memo = {}
    max_value = 0
    for ending_index in range(0, len(seq)):
        value = mcs(seq[:ending_index+1], memo)
        if(value > max_value):
            max_value = value
    return max_value


def mcs(seq, memo=None):
    if memo is None:
        memo = {}
    if not len(seq) in memo:
        if not seq:
            memo[len(seq)] = 0
        else:
            memo[len(seq)] = max(seq[-1], seq[-1] + mcs(seq[:-1], memo))
    return memo[len(seq)]
```

Bottom-Up Approach

```
def bottom_up_mcs(seq):
    best_value = max(0, seq[0])
    prev_value = seq[0]

    for j in range(1,len(seq)):
        prev_value = max(seq[j], seq[j] + prev_value)
        if(prev_value > best_value):
            best_value = prev_value

    return best_value
```

Examples: Edit Distance

What is the cheapest way to convert string x to string y, given the costs of performing insert, delete, and replace operations?

For example, if $x=$"apple" and $y=$"apt", then we can either delete 'p', 'l', and 'e' letters and insert a 't' letter, or delete two of these letters and replace the third one with a 't' letter. These operations have different costs which must be compared to find the minimum.

Examples: Edit Distance

What is the cheapest way to convert string x to string y, given the costs of performing insert, delete, and replace operations?

For example, if $x=$"apple" and $y=$"apt", then we can either delete 'p', 'l', and 'e' letters and insert a 't' letter, or delete two of these letters and replace the third one with a 't' letter. These operations have different costs which must be compared to find the minimum.

This is useful for correcting spelling errors or finding similarity between DNA strings

Edit Distance

Input: two strings x and y, and costs of insert, delete, and replace operations, denoted c_i, c_d, c_r.

Goal: Find the edit distance between x and y, that is the minimum total cost to turn x into y

Edit Distance

Input: two strings x and y, and costs of insert, delete, and replace operations, denoted c_i, c_d, c_r.

Goal: Find the edit distance between x and y, that is the minimum total cost to turn x into y

Subproblem: $EditDistance(i, j)$ edit distance between $x[i:]$ and $y[j:]$
Number of subproblems: $O(|x| \cdot |y|)$

Guess: 3 possibilities: replace $x[i]$ with $y[j]$, insert $y[j]$ in the beginning of $x[i]$, or delete $x[i]$

Edit Distance

Recursive Relation:

$$EditDistance(i,j) = \begin{cases} (|y| - j) * c_i, & \text{if } i = |x| \\ (|x| - i) * c_d, & \text{if } j = |y| \\ \min(EditDistance(i{+}1, j{+}1) + c_r, & \\ \qquad EditDistance(i, j+1) + c_i, \quad , & \text{otherwise}. \\ \qquad EditDistance(i+1, j) + c_d \ ) & \end{cases}$$

here c_r is assumed to be zero when $x[i] = y[j]$.

Running Time Analysis: Time per subproblem: $O(1)$
Total running time: $O(|x|.|y|)$

Original Problem: $EditDistance(0,0)$

Top-Down Approach

```
def EditDistance(i, j, memo=None):
    if memo is None:
        memo = {}
    if not (i, j) in memo:
        if i==len(x):
            memo[(i, j)] = (len(y)-j)*ci
        elif j==len(y):
            memo[(i, j)] = (len(x)-i)*cd
        else:
            cost_of_insert = ci + EditDistance(i, j+1, memo)
            cost_of_delete = cd + EditDistance(i+1, j, memo)
            if x[i] == y[j]:
                cost_of_replace = EditDistance(i+1, j+1, memo)
            else:
                cost_of_replace = cr + EditDistance(i+1, j+1, memo)
            memo[(i, j)] = min(cost_of_insert, cost_of_delete, cost_of_replace)
    return memo[(i, j)]
```

Bottom-Up Approach

```
def EditDistanceBottomUp(x, y):
    table = [[0 for i in range(len(y))] for j in range(len(x))]
    for i in range(len(x)):
        for j in range(len(y)):
            if i == 0 or j ==0:
                if i > j:
                    table[i][j] = i*cd + (0 if x[i]==y[j] else cr)
                else:
                    table[i][j] = j*ci + (0 if x[i]==y[j] else cr)
            else:
                cost_of_insert = ci + table[i][j-1]
                cost_of_delete = cd + table[i-1][j]
                if x[i] == y[j]:
                    cost_of_replace = table[i-1][j-1]
                else:
                    cost_of_replace = cr + table[i-1][j-1]
                table[i][j] = min(cost_of_insert, cost_of_delete, cost_of_replace)
    return table[-1][-1]
```

Example: Dice Throw

Given n dice each with m faces, numbered from 1 to m, find the number of ways to get sum S by adding values on the faces when all the dice are thrown.
For example, if we have 2 dice each with 2 faces, there are only two ways to get 3.

Dice Throw

Subproblem: $Dice(i, m, X)$ the total number of ways to get the sum $0 \leq X \leq S$ using $1 \leq i \leq n$ dice each with m faces
Number of subproblems: $O(n \cdot S)$

Guess: the value on the face of the last dice (m possibilities)

Dice Throw

Recursive Relation:

$$Dice(n, m, S) = \begin{cases} 0, & \text{if } n.m < S \text{ or } S < 0 \\ 1, & \text{if } n = 1 \text{ and } 1 \leq S \leq m \\ \sum_{i=1}^{m} Dice(n-1, m, S-i), & \text{otherwise} \,. \end{cases}$$

Dice Throw

Recursive Relation:

$$Dice(n,m,S) = \begin{cases} 0, & \text{if } n.m < S \text{ or } S < 0 \\ 1, & \text{if } n = 1 \text{ and } 1 \leq S \leq m \\ \sum_{i=1}^{m} Dice(n-1, m, S-i), & \text{otherwise}\,. \end{cases}$$

Running Time Analysis: Number of subproblems: $O(n \cdot S)$
Computation time per subproblem: $O(m)$
Total running time: $O(n \cdot m \cdot S)$

Original Problem: $Dice(n,m,S)$

Top-Down Approach

```
def Dice(n, m, S, memo=None):
    if memo is None:
        memo = {}
    if (n, S) not in memo:
        if S > n*m or S<0:
            ret = 0
        elif n==1 and S >= 1 and S <= m:
            ret = 1
        else:
            ret = sum([Dice(n-1, m, S-i, memo)
                       for i in range(1, m+1)])
        memo[(n, S)] = ret
    return memo[(n, S)]
```

Bottom-Up Approach

Do it at home!

Summary

The main difficulty with solving a problem via dynamic programming is finding the right recurrence (i.e., designing the subproblems).

The top-down approach may be simpler in that we do not have to worry about the order in which we iterate through the subproblems, but we quickly run into problems with recursion depth. The bottom-up approach avoids that altogether.