Dynamic Programming 1

Course Outline

1. Dynamic programming
2. Randomized algorithms
3. Divide and conquer
4. Graph algorithms
5. Max-Flow
6. Linear programming
7. NP-completeness

Dynamic Programming Overview

1. Fibonacci numbers
2. Longest increasing subsequence
3. Longest common subsequence
4. Knapsack
5. Chain matrix multiply
6. Shortest path algorithms

Fibonacci Numbers

1. Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
   - F0 = 0
   - F1 = 1
   - for n > 1: Fn = Fn-1 + Fn-2
2. Input: n >= 0
3. Output: nth Fibonacci number

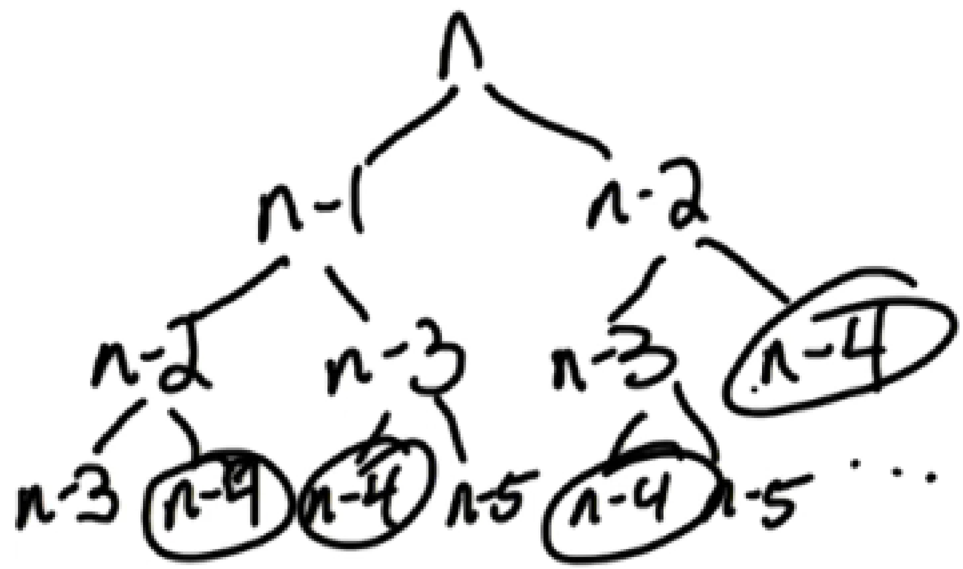
Fibonacci Numbers: Recursive Algorithm

```
Fib1(n):
    input: integer n >= 0
    output: Fn
    if n == 0:
        return 0
    if n == 1:
        return 1
    return Fib1(n-1) + Fib1(n-2)
```

1. Let T(n) = number of steps for Fib1(n)
   - T(n) <= O(1) + T(n-1) + T(n-2)
   - T(n) >= Fn
     - Grows like phi^n / sqrt(5)
       - phi = (1 + sqrt(5)) / 2 ~ 1.618
       - Golden ratio
     - Running time is exponential

Fibonacci Numbers: Exponential Running Time

1. The problem with the recursive solution is we recompute the same values many times



Recursive Fibonacci Solution

2. Instead, we compute the small subproblems first and reuse them for future computations

Fibonacci Numbers: Dynamic Programming

```
Fib2(n):
    F[0] = 0
    F[1] = 1
    for i in range(2, n):
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

1. Runtime analysis
    - Initialization is O(1)
    - For loop is O(n)
    - Amount of work done in the loop is O(1)
    - Total run time is O(n)

Fibonacci Numbers: Dynamic Programming Recap

1. No recursion in algorithm
    - Used the recursive nature of the algorithm, but solution is not recursive
2. Alternative: Memoization
    - Use hash table or some other structure to keep track of previous problems
        - Don't use this at all in this course!
        - No recursion in our algorithms
3. Advantages of dynamic programming vs memoization
    - More beautiful
    - Easier to analyze running time
    - Faster due to no overhead of recursion

Longest Increasing Subsequences

1. Input: n numbers a1, a2, . . . , an
2. Goal: Find length of LIS in a1:an
3. Example: [5, 7, 4, -3, 9, 1, 10, 4, 5, 8, 9, 3]
    - Substring: Set of consecutive elements
    - Subsequence: Subset of element in order (can skip)
    - LIS: -3, 1, 4, 5, 8, 9 (length = 6)

LIS: Subproblem Attempt 1

1. Define subproblem in words
    - F[i] = ith Fibonacci number
2. State recursive relation
    - Express F[i] in terms of F[1], . . . , F[i-1]
        - F[i] = F[i-1] + F[i-2]
3. LIS:
    - Define subproblem in words
        - Let L(i) = length of LIS on a1, a2, . . . , ai
    - State recursive relation
        - Express L(i) in terms of L(1), . . . , L(i-1)

LIS: Recurrence Attempt 1

1. A = [5, 7, 4, -3, 9, 1, 10, 4, 5, 8, 9, 3]
2. L = [1, 2, 2, 2, 3, 3, 4, 4, 4, 5, 6, 6]
3. Want to find LIS with minimum ending character
    - Gives us the most possibilities for appending in the future
4. Need to maintain suboptimal solutions
    - Require length of LIS for every ending character
    - There are i-1 possible ending characters

LIS: Subproblem Attempt 2

1. Let L(i) = length of LIS on a1, . . . , ai and includes ai
2. Example:
    - A = [5, 7, 4, -3, 9, 1, 10, 4, 5, 8, 9, 3]
    - L = [1, 2, 1, 1, 3, 2, 4, 3, 4, 5, 6, 3]

LIS: Recurrence Attempt 2

1. Define subproblem in words
    - Let L(i) = length of LIS on a1, . . . , ai and includes ai
2. State recursive relation
    - L(i) = 1 + max{L(j) : aj < ai && j < i}
    - L(i) = 1 + max(L(j)) where 1 <= j <= i-1, aj < ai

LIS: DP Algorithm

```
LIS(ai,...,an):
    for i in range(1, n):
        L(i) = 1
        for j in range(1, i-1):
            if a[j] < a[i] and L[i] < 1 + L[j]:
                L[i] = 1 + L[j]
```

```
max = 1
for i in range(2, n):
    if L[i] > L[max]
        max = i
return max
```

LIS: DP Running Time

1. Two nested for loops, both of which loop over at most n elements
    - O(n^2)
2. Single for loop for computing max
    - O(n)
3. Total complexity is O(n^2)

LIS Recap

1. First, state problem in words
2. Then, find a recurrence relation
    - Like proof by induction
        - State an inductive hypothesis
            - Usually of the same form of the statement you're trying to prove
        - Try to prove inductive hypothesis
        - If you can't, strengthen the inductive hypothesis and try to prove it again

Longest Common Subsequence

1. Input: Two strings X = x1,...,xn and Y = y1,...,yn
2. Goal: Find the length of the longest string which is a subsequence of both X and Y

LCS Example

1. X = BCDBCDA
2. Y = ABECBAB
3. LCS = BCBA (length 4)
4. Application: Unix diff

LCS: Subproblem Attempt 1

1. Define subproblem in words
    - Try same problem on prefix of input
    - For i where 0 <= i <= n, let L(i) = length of LCS in X1...Xi, Y1...Yi
2. Define recurrence
    - Express L(i) in terms of L(1),...,L(i-1)

LCS: Recurrence Attempt 1

1. For i where 0 <= i <= n, let L(i) = length of LCS in X1...Xi, Y1...Yi
2. Consider the last character:
    - Either Xi == Yi
        - L(i) = 1 + L(i-1)
    - Or Xi != Yi

LCS: Recurrence Attempt 2

1. Consider the last character:
    - Either Xi == Yi

    - L(i) = 1 + L(i-1)
  - Or Xi != Yi
    - LCS does not include Xi and/or Yi
    - If Xi or Yi is not included, we need to look up the LCS where X and Y are different lengths, which is not something we're tracking
2. For this subproblem definition, we are unable to define a valid recurrence
  - Need to account for prefixes of different lengths
  - Use i and j to track

LCS: Subproblem Attempt 2

1. Revised subproblem
   - Two indices i and j and a 2-dimensional table
   - For i,j where 0 <= i <= n and 0 <= j <= n, let L(i,j) = length of LCS in X1…Xi, Y1…Yj
2. Recurrence:
   - L(i,0) = 0
   - L(0,j) = 0

LCS: Recurrence Unequal Case

1. X = BCDBCDA
2. Y = ABECBABD
3. If Xi != Yj either Xi and/or Yj are not in optimal solution
   - If drop Xi then L(i,j) = L(i-1,j)
   - If drop Yi then L(i,j) = L(i,j-1)
   - L(i,j) = max{L(i-1,j), L(i,j-1)}

LCS: Recurrence Equal Case

1. X = BCDBCDA
2. Y = ABECBA
3. If Xi == Yj either drop Xi, drop Yj, or optimal solution ends at Xi = Yj
   - If drop Xi then L(i,j) = L(i-1,j)
   - If drop Yi then L(i,j) = L(i,j-1)
   - If optimal solution then L(i,j) = 1 + L(i-1,j-1)

LCS: Recurrence Equal Recap

1. If Xi == Yj:
   - L(i,j) = max{L(i-1,j), L(i,j-1), 1 + L(i-1,j-1)}
   - This can be simplified to:
     - L(i,j) = 1 + L(i-1,j-1)

LCS: Recurrence Summary

1. Let L(i,j) = length of LCS in X1…Xi, Y1…Yj
   - For i >= 1, j >= 1:
     - If Xi != Yj
       * L(i,j) = max{L(i-1,j), L(i,j-1)}
     - If Xi == Yj
       * L(i,j) = 1 + L(i-1,j-1)
   - Fill L row by row

LCS: DP Algorithm

```
LCS(X, Y):
    for i = 0 to n:
        L(i,0) = 0
    for j = 0 to n:
        L(0,j) = 0
    for i = 1 to n:
        for j = 1 to n:
            if Xi = Yj:
                L(i,j) = 1 + L(i-1,j-1)
            else
                L(i,j) = max(L(i-1,j), L(i,j-1)
    return L(n,n)
```

LCS: Time Complexity

1. What is the running time of the above DP algorithm for the LCS problem?
    - O(n^2)

LCS: DP Table

LCS Example

Input:

X = BCDBCDA

Y = ABECBA

Fill in the table row by row:

	j	0	1	2	3	4	5	6
i			A	B	E	C	B	A
0		0	0	0	0	0	0	0
1	B	0	0	1	1	1	1	1
2	C	0						
3	D	0						
4	B	0						
5	C	0						
6	D	0						
7	A	0						

Here is an example problem with inputs **X = BCDBCDA**, and **Y = ABECBA**. The first couple of rows of the DP table **L(i, j)** have been filled in for you. Now try to fill in the third row (for **C**) in the box below, using comma-separated values, as indicated.

```
   _, A, B, E, C, B, A
_, 0, 0, 0, 0, 0, 0, 0
B, 0, 0, 1, 1, 1, 1, 1
C, 0, 0, 1, 1, 2, 2, 2

```

DP Table

1. Next row of table: 0,1,1,2,2,2

LCS: Extract Sequence

Here is what the DP table looks like when it is complete. The Longest Common Subsequence should b
of length **4**.

Input:

X = BCDBCDA

Y = ABECBA

Fill in the table row by row:

	j	0	1	2	3	4	5	6
i			A	B	E	C	B	A
0		0	0	0	0	0	0	0
1	B	0	0	1	1	1	1	1
2	C	0	0	1	1	2	2	2
3	D	0	0	1	1	2	2	2
4	B	0	0	1	1	2	3	3
5	C	0	0	1	1	2	3	3
6	D	0	0	1	1	2	3	3
7	A	0	1	1	1	2	3	$\boxed{4}$

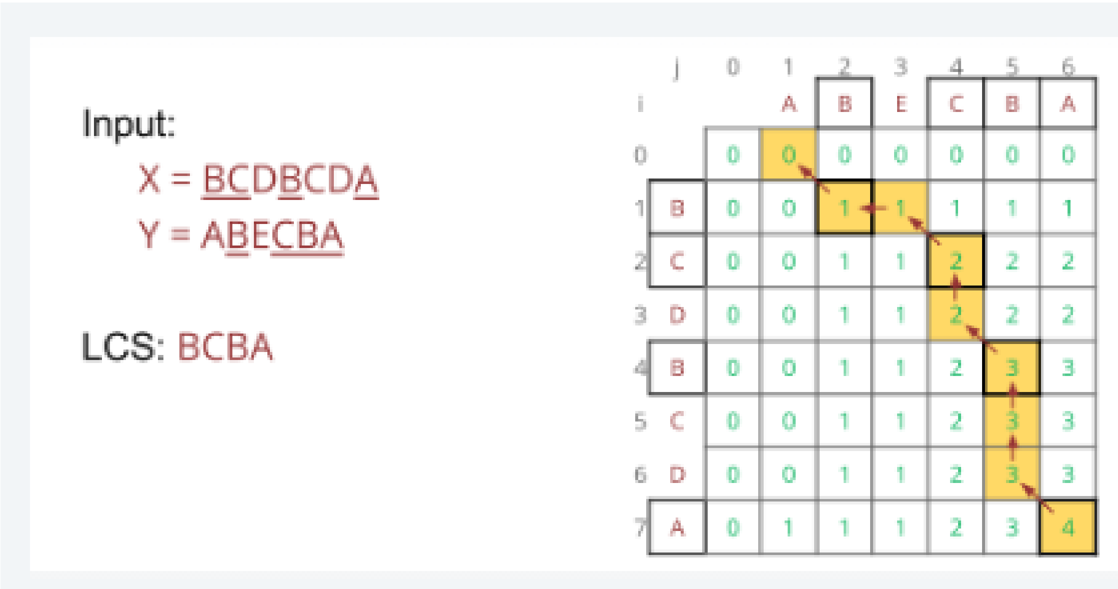
LCS: Extract Sequence

Now use this table to trace back and find the Longest Common Subsequence.

Hint: *Start with the last matching cell.*

Complete DP Table

1. Use this table to extract the LCS
    - Start with last matching cell and work backwards
    - ABCB



Solution

DP1: Practice Problems

1. Practice problems:
   - DPV 6.1 (contiguous subsequence aka substring)
   - DPV 6.2 (hotel stops)
   - DPV 6.3 (Yuckdonald's)
   - DPV 6.4 (string of words)
   - DPV 6.11 (longest common substring)
2. Approach:
   - Define subproblem in words
     - Try the same problem on prefix
     - Add constraint - include last element
   - Define recurrence relation
     - T(i) in terms of T(1),...,T(i-1)

DP1: Practice Problem 6.1

1. Input: a1,...,an
2. Goal: Substring with max sum
3. Subproblem: for 0 <= i <= n
   - Let S(i) = max sum from substring of a1,...,ai
   - S(i) in terms of S(1),...,S(i-1)
   - Need to strengthen subproblem definition to include ai-1

DP1: Practice Solution

1. Input: a1,...,an
2. Goal: Substring with max sum
3. Subproblem: for 0 <= i <= n
   - Let S(i) = max sum from substring of a1,...,ai which includes ai
   - S(0) = 0
   - S(i) = ai + max(0, a(i-1))
4. Output: max(S(i))

5. Time complexity: O(n)