Dynamic Programming

Week4

Outline

I. Overview
II. Comparison to Divide and Conquer
III. Dynamic Programming
IV. Recurrence Relations
V. Principle of Optimality
VI. Approximate String Match
VII. Time Complexity

Introduction

- Greedy algorithms
  - did not necessarily provide an optimal solution
  - did return a solution very quickly when compared to their brute force counterparts

- Dynamic Programming
  - Like divide and conquer algorithms, dynamic programming solutions are suited for problems that are inherently recursive or involve a recurrence relation.

Recurrence Relations

- A recurrence relation consists of a function with a recursive definition and initial conditions.
  - Example: Factorial
    - Recursive definition: factorial(n) = n*factorial (n-1)
    - Intitial conditions: factorial(0) = 1
    - Sequence View: 1,1,2,6,24, …
- Divide and Conquer approaches are best suited for *non-overlapping recurrence relations*
- Dynamic programming solutions are applicable to *overlapping recurrence relations.*

Overlapping Recurrence Relations

- Divide and Conquer may result in inefficiency
  - Recompute the same sub-problem multiple times

- Dynamic Programming solutions save subproblems results and recall their solutions rather than recomputing them!

Fibonacci Sequence Example

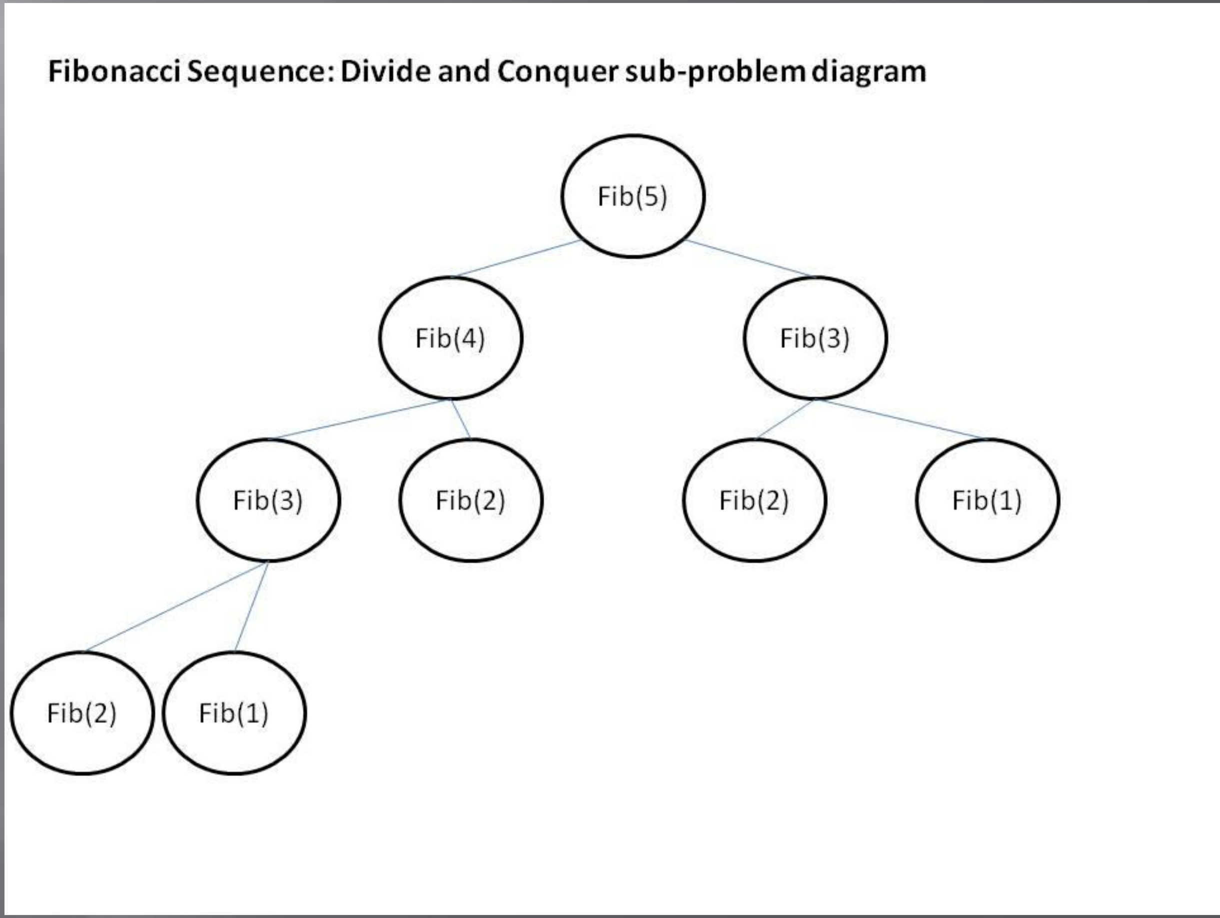
- Fibonacci Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- Recurrence
  - Function : Fib(n) = Fib(n-1) + Fib(n-2)
  - Intitial conditions: Fib(0) = 0 and Fib(1) = 1

- Divide and conquer solution

```
// Using a divide and conquer scheme, recursively computes Fib(n)
Algorithm DnCFib( n )
    if n == 1 or n == 0{
            return n;}
    else{
            return DnCFib(n-1) + DnCFib(n-2);}
} end
```

An Overlapping Recurrence Relation

- Since we have an overlapping recurrence relation, the divide and conquer solution recomputes the answer to many of the subproblems including Fib(3) twice, Fib(2) thrice, and Fib(1) twice.
- Excessive recursion



Dynamic Programming Paradigm

- Using dynamic programming we wish to store intermediate results and use them as necessary *(rather than recomputing*).

- Dynamic Programming Solution.
- *// Using a dynamic programming scheme, computes Fib(n)*
- ***Algorithm** dynFib(n)*
- *fib[0] = 0;*
- *fib[1]-= 1;*
- *for i:= 2 to n-1 do{*
- *fib[i] = fib[i-1] + fib[i-2];*
- *} end for*
- *return fib[n];*
- *} end*

Upon inspection, we can see that this very direct and intuitive solution has a time complexity that is O(*n*): An efficient solution

Recursive Implementation

- Divide and Conquer approaches lend themselves nicely to recursive implementation.
- A recursive implementations for dynamic programming solutions do exist in many cases; however the algorithm designs are not always intuitive.
  - Recursive Fib

- *// Using a dynamic programming scheme, computes Fib(n) recursively*
- ***Algorithm** dynFib_recursion(n , p_0, p_1)*
- *if n ==1 {*
- *return p_1;*
- *else{*
- *return dynFib_recursion(n-1, p_1, p_0+p_1);*
- *} end*

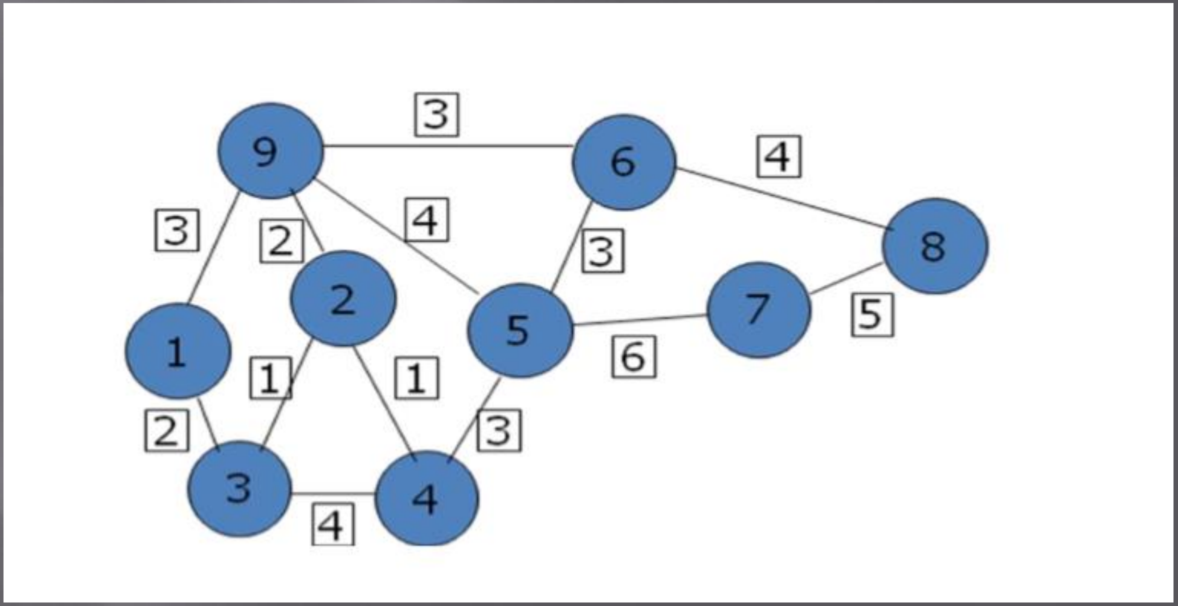
Why (and when) Does Dynamic Programming Work?

- Principle of Optimality
  - The principle of optimality states that an optimal solution consists of optimal solutions to its subproblems. It is also worth noting that each subproblem is usually solved sequentially.

- When does this principle hold?

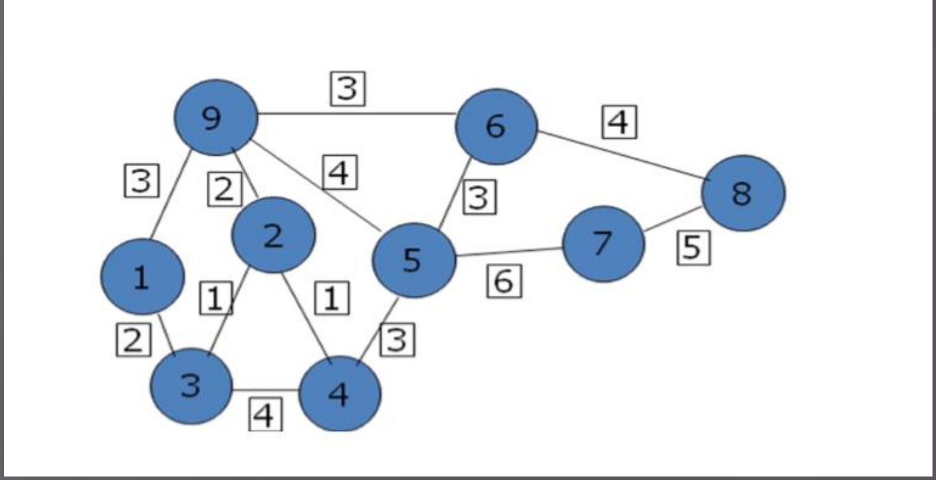
Principle of Optimality

- Example: Shortest Path Problem.
  - Does Principle of Optimality hold?
  - Applying this principle on this problem, we state that if we have an optimal (shortest) path from A to B, then all subpaths (to intermediate vertices) along this path are also optimal (shortest).
  - Proof by contradiction.



Principle of Optimality

- Example: *Longest* (non-cyclic) Path Problem.
  - Interestingly the principle of optimality *does not hold* for the longest path solution. That is, if we have an optimal (longest) path from A to B (without cycles), it is not guaranteed that all subpaths are also optimal (longest paths). Therefore if there is a node C on the longest path from A to B, then the subpath from A to C is not necessarily the longest path from A to C. [Try to find a counterexample to prove this statement]



Sequential Improvement

- As we have seen thus far, in dynamic programming, as each subproblem is sequentially solved, we get closer to the optimal solution: <u>sequential improvement.</u>
- Similar to our shortest path problem, dynamic programming solutions can be applied to many optimization problems (as long as the principle of optimality holds.)
- Many applications of dynamic programming have to do with finding a minimum or optimum number of steps, operations or units for some task.

Example: String Match or Edit Distance

- Approximate String Matching Problem.
  - This is similar to the shortest paths problem, but instead of computing distance between nodes in a graph, we are computing the "edit distance" (difference) between two strings.
  - Just like a spell check program!

- Problem.
  - "How different" is string $X = x_1,\ldots x_n$ is from string $Y = y_1,\ldots,y_m$?
  - First, we need to define the concept of difference here.

Difference in Strings

- Compare two Strings "watermelon" and "atermelon“

  w| a| t| e| r| m|e| l| o| n

  a | t | e|r |m|e | l|o|n

A naïve comparison would suggest these words are completely different!

- does not account for missed letters, typos, different lengths, …

Instead, it is common to produce a distance matrix to track the distance between each string, as we iteratively solve each subproblem.

Does dynamic programming fit?

- Step 1: Formalize the recurrence relation.
- Step 2: Does principle of Optimality hold.
- Step 3: Convert to code.

- One way to measure this "distance" is a minimum edit distance.
  - Should we compare all possible substrings?

- Step 1: We will define D[i,j] be the minimum number of differences between the substrings $x_1,\ldots x_i$ is from string $Y = y_1,\ldots,y_j$.

Define Edit Distance

- We will simply compute the difference between strings by noting number of unmatched characters.
  - Mismatch: If a character is unmatched, we will increment the difference between the strings by 1.
  - Insertion: If a character needs to be inserted we will increment the distance by 1
  - Deletion: If a character needs to be deleted we will increment the distance by 1

- Compare "Hello" and "elllo".
  - What would you say is the edit distance?
  - Lets compare substrings using a Matrix

Substring Compare

D		e	l	l	l	o
	0	1	2	3	4	5
H	1	1	2	3	4	5
e	2	1	2	3	4	5
l	3	?	?	2	3	4
l	4	3	2	1	2	3
o	5	4	3	?	?	?

Lets use entry (i,j) to count the number of edits necessary to transform $x_1, x_2, \dots, x_i$ to $y, y_2, \dots, y_j$. The first row and column will correspond to the empty string.

Step 1: Can we formalize a recurrence relation.

D		e	l	l	l	o
	0	1	2	3	4	5
H	1	1	2	3	4	5
e	2	1	2	3	4	5
l	3	?	?	2	3	4
l	4	3	2	1	2	3
o	5	4	3	?	?	?

- Here we have found a minimum edit distance of two.
- This is similar to shortest path, so lets see if we can formalize a recurrence relation.
- D[i,j] = ?

Step 1: Recurrence Relation

D	0	1	2	…	n
0	0	1	2	…	n
1	1				
2	2	D[i-1,j-1]	D[i-1,j]		
…	…	D[i,j-1]	D[i,j]		
m	m				

$$\mathrm{D[i,j]} = \min \begin{cases} D[i-1,j-1], \quad \text{if } x_i = y_j \text{ OR } D[i-1,j-1]+1, \text{if }, \text{if } x_i \neq y_j \\ D[i-1,j]+1, \quad \text{account for } x_i \text{ is not in } Y \text{ (size mismatch, compare to empty string)} \\ D[i,j-1]+1, \quad \text{account for } y_i \text{ is not in } X \text{ (size mismatch, compare to empty string)} \end{cases}$$

Step 2: Does Principle of Optimality Hold?

- Is there a shorter edit distance than the one computed using this recurrence relation?
  - Nope – proof similar to that of shortest path.

- Yes – it holds!

Step 3: Convert to Code

- Exercise:
  - Inputs: X and Y
  - Output: D or min value of last row or last column

  - Lets try to code this up as a group.

$$
D[i,j] = \min \begin{cases} D[i-1,j-1], & \text{if } x_i = y_j \text{ OR } D[i-1,j-1]+1, \text{if }, \text{if } x_i \neq y_j \\ D[i-1,j]+1, & \text{account for } x_i \text{ is not in } Y \text{ (size mismatch, compare to empty string)} \\ D[i,j-1]+1, & \text{account for } y_i \text{ is not in } X \text{ (size mismatch, compare to empty string)} \end{cases}
$$

Computational Complexity

- The crux of the algorithm is building the D matrix, which is nxm.
- Time: Thus *D* matrix will have *m* x *n* entries. The computation of each entry can be done in constant or O(1) time. Therefore the total time complexity is O(*mn*).
- Space: Since we must construct the *D* matrix we will require up to O(*mn*) space as well since we will need *m* x *n* memory locations.
  - In some instances the entire D matrix does not need to be maintained throughout the process. In such cases an improvement of space complexity is possible.