Dynamic Programming

League of Programmers

ACA, IIT Kanpur

October 22, 2012

Outline

Dynamic Programming

What is DP?

DP is another technique for problems with optimal substructure: An optimal solution to a problem contains optimal solutions to subproblems. This doesn't necessarily mean that every optimal solution to a subproblem will contribute to the main solution.

Dynamic Programming

What is DP?

DP is another technique for problems with optimal substructure: An optimal solution to a problem contains optimal solutions to subproblems. This doesn't necessarily mean that every optimal solution to a subproblem will contribute to the main solution.

- For divide and conquer (top down), the subproblems are independent so we can solve them in any order.

Dynamic Programming

What is DP?

DP is another technique for problems with optimal substructure: An optimal solution to a problem contains optimal solutions to subproblems. This doesn't necessarily mean that every optimal solution to a subproblem will contribute to the main solution.

- For divide and conquer (top down), the subproblems are independent so we can solve them in any order.
- For greedy algorithms (bottom up), we can always choose the "right" subproblem by a greedy choice.

Dynamic Programming

What is DP?

DP is another technique for problems with optimal substructure: An optimal solution to a problem contains optimal solutions to subproblems. This doesn't necessarily mean that every optimal solution to a subproblem will contribute to the main solution.

- For divide and conquer (top down), the subproblems are independent so we can solve them in any order.
- For greedy algorithms (bottom up), we can always choose the "right" subproblem by a greedy choice.
- In dynamic programming, we solve many subproblems and store the results: not all of them will contribute to solving the larger problem. Because of optimal substructure, we can be sure that at least some of the subproblems will be useful

Dynamic Programming

Steps to solve a DP problem

Dynamic Programming

Steps to solve a DP problem

1. Define subproblems

Dynamic Programming

Steps to solve a DP problem

1. Define subproblems
2. Write down the recurrence that relates subproblems

Dynamic Programming

Steps to solve a DP problem

1. Define subproblems
2. Write down the recurrence that relates subproblems
3. Recognize and solve the base cases

Fibonacci Sequence

Naive Recursive Function

```
int Fib(int n){
  if(n==1 || n==2)
    return 1;
  return Fib(n-1)+Fib(n-2)
}
```

Fibonacci Sequence

DP Solution O(n)

```
Fib[1] = Fib[2] = 1;
for(i=3;i<N;i++)
  Fib[i] = Fib[i-1]+Fib[i-2]
```

Problem

Problem 2

Given n, find the number of different ways to write n as the sum of 1, 3, 4

Example: for $n = 5$, the answer is 6

$5 = 1+1+1+1+1$

$= 1+1+3$

$= 1+3+1$

$= 3+1+1$

$= 1+4$

Problem

Define Subproblems

Problem

Define Subproblems

- D_n be the number of ways to write n as the sum of 1, 3, 4

Problem

Define Subproblems

- D_n be the number of ways to write n as the sum of 1, 3, 4
- Find the recurrence

$$D_n = D_{n-1} + D_{n-3} + D_{n-4}$$

Problem

Define Subproblems

- D_n be the number of ways to write n as the sum of 1, 3, 4
- Find the recurrence

$$D_n = D_{n-1} + D_{n-3} + D_{n-4}$$

- Solve the base cases
  $D_0 = 1$
  $D_n = 0$ for all negative n

Problem

Define Subproblems

- D_n be the number of ways to write n as the sum of 1, 3, 4
- Find the recurrence

$$D_n = D_{n-1} + D_{n-3} + D_{n-4}$$

- Solve the base cases
  $D_0 = 1$
  $D_n = 0$ for all negative n
- Alternatively, can set: $D_0 = D_1 = D_2 = 1$, $D_3 = 2$

Problem

Implementation

```
D[0]=D[1]=D[2]=1; D[3]=2;
for(i=4;i<=n;i++)
  D[i]=D[i-1]+D[i-3]+D[i-4];
```

LCS

Problem 3

Given two strings x and y, find the longest common subsequence (LCS) and print its length.
Example:
x: ABCBDAB
y: BDCABC
"BCAB" is the longest subsequence found in both sequences, so the answer is 4

LCS

Analysis

LCS

Analysis

- There are 2^m subsequences of X.
  Testing a subsequence (length k) takes time O(k + n).
  So brute force algorithm is $O(n * 2^m)$.

LCS

Analysis

- There are 2^m subsequences of X.
  Testing a subsequence (length k) takes time O(k + n).
  So brute force algorithm is $O(n * 2^m)$.
- Divide and conquer or Greedy algorithm?

LCS

Analysis

- There are 2^m subsequences of X.
  Testing a subsequence (length k) takes time O(k + n).
  So brute force algorithm is $O(n * 2^m)$.
- Divide and conquer or Greedy algorithm?
  - No, can't tell what initial division or greedy choice to make.

LCS

Analysis

- There are 2^m subsequences of X.
  Testing a subsequence (length k) takes time O(k + n).
  So brute force algorithm is $O(n * 2^m)$.
- Divide and conquer or Greedy algorithm?
  - No, can't tell what initial division or greedy choice to make.

Thus, none of the approaches we have learned so far work here!!!

LCS

Analysis

- There are 2^m subsequences of X.
  Testing a subsequence (length k) takes time O(k + n).
  So brute force algorithm is $O(n * 2^m)$.
- Divide and conquer or Greedy algorithm?
  - No, can't tell what initial division or greedy choice to make.

Thus, none of the approaches we have learned so far work here!!!

Intuition

A LCS of two sequences has as a prefix a LCS of prefixes of the sequences. So, We concentrate on LCS for smaller problems, i.e simply removes the last (common) element.

LCS

Define Subproblems

LCS

Define Subproblems

- Let $D_{i,j}$ be the length of the LCS of $x_{1\dots i}$ and $y_{1\dots j}$

LCS

Define Subproblems

- Let $D_{i,j}$ be the length of the LCS of $x_{1\dots i}$ and $y_{1\dots j}$
- Find the recurrence

LCS

Define Subproblems

- Let $D_{i,j}$ be the length of the LCS of $x_{1\ldots i}$ and $y_{1\ldots j}$
- Find the recurrence
  - If $x_i = y_j$, they both contribute to the LCS. In this case,
$$D_{i,j} = D_{i-1,j-1} + 1$$

LCS

Define Subproblems

- Let $D_{i,j}$ be the length of the LCS of $x_{1...i}$ and $y_{1...j}$
- Find the recurrence
  - If $x_i = y_j$, they both contribute to the LCS. In this case,
    $$D_{i,j} = D_{i-1,j-1} + 1$$
  - Either x_i or y_j does not contribute to the LCS, so one can be dropped. Otherwise,
    $$D_{i,j} = max\{D_{i-1,j}, D_{i,j-1}\}$$

LCS

Define Subproblems

- Let $D_{i,j}$ be the length of the LCS of $x_{1 \ldots i}$ and $y_{1 \ldots j}$
- Find the recurrence
  - If $x_i = y_j$, they both contribute to the LCS. In this case,
  $$D_{i,j} = D_{i-1,j-1} + 1$$
  - Either x_i or y_j does not contribute to the LCS, so one can be dropped. Otherwise,
  $$D_{i,j} = max\{D_{i-1,j}, D_{i,j-1}\}$$
- Find and solve the base cases: $D_{i,0} = D_{0,j} = 0$

LCS

Implementation

```
for(i=0;i<=n;i++) D[i][0]=0;
for(j=0;j<=m;j++) D[0][j]=0;
for(i=1;i<=n;i++) {
  for(j=1;j<=m;j++) {
    if(x[i]==y[j])
      D[i][j]=D[i-1][j-1]+1;
    else
      D[i][j]=max(D[i-1][j],D[i][j-1]);
  }
}
```

LCS

Recovering the LCS

Modify the algorithm to also build a matrix $D[1 \ldots n; 1 \ldots m]$, recording how the solutions to subproblems were arrived at.

Longest Non Decreasing Subsequence

Problem 4

Given an array [1 , 2 , 5, 2, 8, 6, 3, 6, 9, 7]. Find a subsequence which is non decreasing and of maximum length.

1-5-8-9 forms a non decreasing subsequence
So does 1-2-2-6-6-7 but it is longer

LNDS

Subproblem

Length of LNDS ending at i^{th} location

LNDS

Subproblem

Length of LNDS ending at i^{th} location

Implementation

```
for(i=0;i<100;i++) {
  max=0;
  for(j=0;j<i;j++) {
    if(A[i]>=A[j] && L[j]>max)
      max = L[j];
  }
  L[i] = max+1;
}
```

Problem

Problem 5

Given a tree, color nodes black as many as possible without coloring two adjacent nodes.

Problem

Define Subproblems

Problem

Define Subproblems

- we arbitrarily decide the root node r

Problem

Define Subproblems

- we arbitrarily decide the root node r
- B_v : the optimal solution for a subtree having v as the root, where we color v black

Problem

Define Subproblems

- we arbitrarily decide the root node r
- B_v : the optimal solution for a subtree having v as the root, where we color v black
- W_v : the optimal solution for a subtree having v as the root, where we don't color v

Problem

Define Subproblems

- we arbitrarily decide the root node r
- B_v : the optimal solution for a subtree having v as the root, where we color v black
- W_v : the optimal solution for a subtree having v as the root, where we don't color v
- The answer is $\max\{B_r, W_r\}$

Problem

Find The Recurrence

Observation

Problem

Find The Recurrence

Observation

- Once v's color is determined, its subtrees can be solved independently

Problem

Find The Recurrence

Observation

- Once v's color is determined, its subtrees can be solved independently
  - If v is colored, its children must not be colored

$$B_v = 1 + \sum_{u \in child(v)} W_u$$

Problem

Find The Recurrence

Observation

- Once v's color is determined, its subtrees can be solved independently
  - If v is colored, its children must not be colored

$$B_v = 1 + \sum_{u \in child(v)} W_u$$

  - If v is not colored, its children can have any color

$$W_v = 1 + \sum_{u \in child(v)} B_u$$

Problem

Find The Recurrence

Observation

- Once v's color is determined, its subtrees can be solved independently
  - If v is colored, its children must not be colored

$$B_v = 1 + \sum_{u \in child(v)} W_u$$

  - If v is not colored, its children can have any color

$$W_v = 1 + \sum_{u \in child(v)} B_u$$

- Base cases: leaf nodes

Outline

Problems

Added on the contest on VOC http://ahmed-aly.com/voc/
Contest ID: 2616
Name: ACA, IITK LOP 03
Author: pnkjjindal
Links:

1. http://spoj.pl/problems/ACTIV
2. http://www.spoj.pl/problems/COINS
3. http://spoj.pl/problems/IOIPALIN
4. http://spoj.pl/problems/ADFRUITS
5. http://www.spoj.pl/problems/RENT
6. http://www.spoj.pl/problems/M3TILE
7. http://www.spoj.pl/problems/IOPC1203
8. http://www.spoj.pl/problems/NGON