Dynamic Programming

15-451

Ananda Gunawardena
("guna")

September 29, 2010

In this lecture..

- Algorithmic Techniques
- Dynamic Programming
- Applications
  - Fibonacci series
  - Coin Change Problem
  - Least Common Subsequence problem
  - Knapsack problem

Algorithmic Techniques

Many Algorithmic Techniques

- Recursive algorithms
- Iterative Algorithms
- Brute Force Algorithms
- Divide and Conquer Algorithms
- Backtracking Algorithms
- Randomized Algorithms
- Greedy Algorithms
- Approximation Algorithms
- Dynamic Programming

Types of Algorithms

- Recursive
  - Tower of Hanoi
- Iterative
  - Iterate over all possible pairs
- Brute Force Algorithm
  - Consider all possibilities and find a one that works
- Randomized or probabilistic Algorithms
  - Randomized the data set to improve performance
- Divide and Conquer Algorithms
  - Divide: Smaller sub-problems solved recursively, Conquer – solution to the original using solution to sub-problems.
    - Eg: MergeSort, quicksort
- Backtracking Algorithms
  - Use a stack to backtrack
- Greedy Algorithms
  - Take the current "best" solution. In other words the greedy choice.
- DYNAMIC PROGRAMMING

Dynamic Programming(DP)

- Not much to do with "dynamic" or "programming"
  - idea from control theory
- **Programming** really refers to the use of a table
- **Dynamic** refers to something that changes (eg: table gets updated)
- DP Algorithmic Technique can be used to reduce **exponential time** algorithms to **polynomial time** algorithms

Motivation with Fibonacci

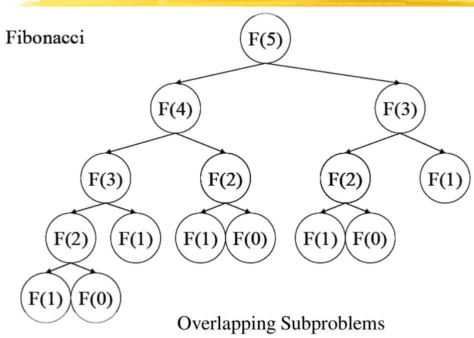
- The Fibonacci sequence
  - f(0) = 1
  - f(1) = 1
  - f(n) = f(n-1) + f(n-2) if n ≥ 2

Fibonacci – the recursive program

```
public static long fib(int n) {
     if (n ≤ 1) return n;
     else
       return (fib(n-1)+fib(n-2))
}
```

- What is the complexity of this algorithm?

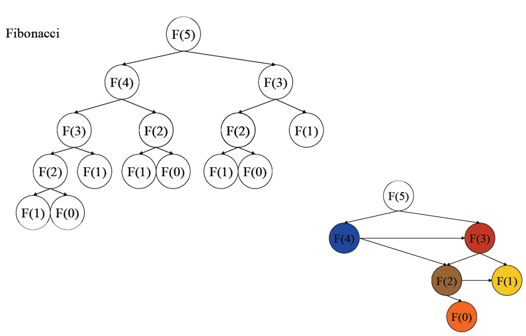
Exponential number of calls



What if we can avoid all the calls

- We note many overlapping sub problems
- What if we avoid re-computing the sub-problems
- Perhaps reduce complexity from exponential to linear …

Reduce the calculations



Dynamic programming is about reducing **exponential time** algorithms to **polynomial time** algorithms

Memoizing

- There are overlapping sub problems
- Number of sub problems is small
  - Eg: Some polynomial of n
- Store solutions to sub problems already solved
  - Increase space complexity
  - But decrease time complexity
- This idea is called "Memoizing"

Fibonacci – with Memoizing

```
public static long fib(int n) {
    if (memo[n] != -1) return memo[n];
    if (n<=1) return n;
    long u = fib(n-1);
    long v = fib(n-2);
    memo[n] = u + v;
    return u + v;
}
```

```
memo = new long[n+1];
for(int i=0; i<=n; i++)
    memo[i] = -1;
```

- This is called top-down DP

Memoization Concept

- Idea is to Remember previous results and reuse them
  - When computing the value of a function, return the **saved result** rather than calculating it again
- The "function" must be a function!
  - No side effects
  - Returns the same value each time
- Saving the values
  - Array
  - Hashtable
- Trade-offs
  - Time to retrieve vs. time to compute
  - Storage space vs. time to compute

Memoization

- Name coined by Donald Michie, Univ of Edinburgh (1960s)
- Fibonacci is the Perfect Example
- Useful for
  - game searches
  - evaluation functions
  - web caching

Fibonacci – Bottom-Up Memoizing

```
public static long fib(int n) {
    if (n<=1) return n;

    long last = 1;
    long prev = 0;
    long t = -1;

    for (int i = 2; i<=n; i++) {
        t = last + prev;
        prev = last;
        last = t;
    }
    return t;
}
```

- What is the complexity of this program?

For dynamic programming

- Key ingredients:
  - *Simple sub problems.*
    - Problem can be broken into sub problems, typically with solutions that are easy to store in a table/array.
  - *Sub problem optimization.*
    - Optimal solution is composed of optimal sub problem solutions.
  - *Sub problem overlap.*
    - Optimal solutions to separate sub problems can have sub problems in common.

Recall

- Dynamic programming in some sense involves making a table
- The table must be easy to build
- A problem that could take O(n!) could perhaps be reduced to O(polynomial)

Application of DP

Coin Change Problem

Coin Change Problem

- Problem: What is the minimum number of coins required to change 63 cents?
  - Assume US coin denominations of 25-cent, 10-cent, 5-cent and 1-cent
- Solution: (Grocery Clark technique)
  - Two quarters
  - One Dime
  - Three pennies
- This is a "Greedy-Algorithm"

Coin Change Problem

- Does a "greedy algorithm" always works?
- NO!
- Suppose we have a 21-cent coin
  - Grocery Clark algorithm does not work
- How do we find a solution?
  - Answer: Dynamic Programming

Coin Change Problem

- Suppose we have n denominations of coins, $1=d[1] < d[2] < \ldots < d[n]$
- Suppose C[i][j] denote the minimum number of coins required to make change for amount j, if only the coins 1,2…i are allowed
- If we are looking for minimum number of coins for amount A using all coins, then we are looking to find
  - C[n][A]

Coin Change Problem

- Example: d[1]=1, d[2]=6, d[3]=10

You want change for 12 cents.
What is the greedy solution here?
or

j

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12
1	0	1	2	3	4	5	6	7	8	9	10	11	12
2	0	1	2	3	4	5	1	2	3	4	5	6	2
3	0	1	2	3	4	5	1	2	3	4	1	2	2

C[i][j] = minimum # of coins to change amount j using coins 1..i

Coin Change Problem

- Question: How can we obtain row i from previously computed values?
- Recursive Solution
  - C[i][j] = C[i-1][j] if j < d[i]
  - = min(1+C[i][j-d[i]], C[i-1][j]) if d[i] ≤ j

j

i	0	1	2	3	4	5	6	7	8	9	10	11	12
1													
2													
3													

d[1]=1, d[2]=6, d[3]=10

Application of DP

Knapsack Problem

Knapsack Problem

- Imagine a homework problem with seven different parts, A thru G

	A	B	C	D	E	F	G
value:	7	10	5	12	14	6	12
time:	3	4	2	6	7	3	5

- You have 15 hours
- Which parts should you complete in order to get "maximum" credit?
  - Greedy algorithm (E,G,A) = 33 pts
  - But is there a better total?
- How about a Brute-Force Algorithm?

Knapsack Problems

- There are two kinds of knapsack problems
  - Binary Knapsack problem (BKP)
    - Must take the "whole" item
  - Fractional Knapsack Problem (FKP)
    - Can take "fractions" of items

Fractional knapsack problem (FKP)

- You rob a store: find ***n*** kinds of items
  - Gold dust. Wheat. Root Beer.
- The total inventory for the ***i*** th kind of item:
  - Weight: w_i pounds
  - Value: v_i dollars
- Knapsack can hold a maximum of ***W*** pounds.
- **Q: how much of each kind of item should you take?**

(Can take fractional weight)

A Greedy Algorithm?

- Try taking the most expensive (per unit cost) item first.
- And the next expensive item, and the next expensive item etc..
- Greedy solution:
  - Fill knapsack with "most valuable" item until all is taken.
    - Most valuable = max(v_i / w_i , i=1…n)
  - Then next "most valuable" item, etc.
  - Until knapsack is full.

Optimal Solution must include Greedy Choice

For item ***i*** let
- w_i be the total inventory,
- v_i be the total value,
- k_i be the weight in knapsack.

Assume total optimal value:

$$V = \sum_{i=1}^{n} k_i \left(\frac{v_i}{w_i} \right)$$

Let item ***h*** be the item with highest $/lb.
If $k_h < w_h$, and $k_j > 0$ for some $j \neq h$, then replace ***j*** with an equal weight of ***h***. Let new total value = ***V'***.

Difference in total value:

$$V' - V = k_j \left(\frac{v_h}{w_h} \right) - k_j \left(\frac{v_j}{w_j} \right) \geq 0$$

but, by definition of ***h***,

$$\frac{v_j}{w_j} \leq \frac{v_h}{w_h}$$

Therefore all of item *h* should be taken. ∎

Binary Knapsack Problem

- Must take the whole item
- How can we solve this problem?
  - Would a "greedy" algorithm work? (T=15 hrs)

	A	B	C	D	E	F	G
value:	7	10	5	12	14	6	12
time:	3	4	2	6	7	3	5

  - Operations Research Approach
    - Prioritize tasks to maximize the outcome
  - Use DP to find the "task priority"

Knapsack Problem

- Imagine a homework problem with seven different parts, A thru G

	A	B	C	D	E	F	G
value:	7	10	5	12	14	6	12
time:	3	4	2	6	7	3	5

- You have 15 hours
- Which parts should you complete in order to get "maximum" credit?

Solving Knapsack

- Consider a general problem with N parts, 1, 2, …., N and let time[i] and value[i] denote the time and value of Part i.
- Let T be the total time
- **DP Idea**
  - Create a table A where A[i][t] denotes max value we get if we use items from 1,2…i and allow t time.

Knapsack problem ctd..

	A	B	C	D	E	F	G
value:	7	10	5	12	14	6	12
time:	3	4	2	6	7	3	5

A has size (N+1) by (T+1).

time

# items	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
1																
2																
3																
4																

The big question?

- How to figure out the next row from the previous ones?

A[i][t] =

MAX(A[i-1][t], *// don't use item i*

A[i-1][t - time[i]] + value[i]) *//use item i.*

Memoizing – Bottom-Up

```
for(t=0; t <= T; ++t)  A[0][t] = 0;
for(i=1; i <= N; ++i) {
        for(t=0; t < time[i]; ++t)
           A[i][t] = A[i-1][t];

        for(t=time[i]; t <= T;  ++t) {
           A[i][t] = MAX( A[i-1][t],
               A[i-1][t - time[i]] + value[i]);
  }
}
```

Homework

- Complete the table for i=7, T=15.

```
A[i][t] = MAX( A[i-1][t],  // don't use item i
          A[i-1][t - time[i]] + value[i]) //use item i.
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4																
5																
6																
7																

- What is the run time of this algorithm?

	A	B	C	D	E	F	G
value:	7	10	5	12	14	6	12
time:	3	4	2	6	7	3	5

Recursive Code

```
ComputeValue(N,T)    // T = time left, N = # items still to choose from
   {
      if (T <= 0 || N = 0) return 0;
      if (time[N] > T) return ComputeValue(N-1,T);
                    // can't use Nth item
     return max(value[N] + ComputeValue(N-1, T - Time[N]),
                ComputeValue(N-1, T));
      }
```

- What is the runtime of this code?

Memoizing Top-Down

- As we calculate values, make a memo of those

```
ComputeValue(N,T)     // T = time left, N = # items still to choose from
{
       if (T <= 0 || N = 0) return 0;
       if (arr[N][T] != unknown) return arr[N][T];

    // otherwise, we haven't computed it yet.  Compute and store.

    if (time[N] > T)
      arr[N][T] = ComputeValue(N-1,T);
    else arr[N][T] = max(value[N] + ComputeValue(N-1, T - Time[N]),
                       ComputeValue(N-1, T));
    return arr[N][T];
}
```

- **What is the runtime of this code?**

One Last Example

Longest common Sub sequence

$$S = \texttt{ABAZDC}$$

$$T = \texttt{BACBAD}$$

What is the longest common Subsequence for S and T?

How do we think about this?

- Lets develop some terminology
  - L[i,j] = length of the longest common sub sequence if we use the prefixes of S and T
    - That is, we use: S[1..i] and T[1…j]
  - So if S is length n and T is length m, then we are looking for L[n,m]
  - So if we calculate all of L[i,j], we are making a table in O(mn) time.
  - But….

We must be able to use the previously computed values

- Find a relation between then
  - L[i, j], L[i-1,j], and L[i, j-1]
- Two cases
  - S[i] ≠ T[j]
    - L[i,j] = max(L[i-1,j], L[i,j-1]))
  - S[i] = T[j]
    - L[i,j] = 1 + L[i-1,j-1]

So if we fill the table

$S =$ ABAZDC

$T =$ BACBAD

	B	A	C	B	A	D
A	0	1	1	1	1	1
B	1	1	1	2	2	2
A	1	2	2	2	3	3
Z	1	2	2	2	3	3
D	1	2	2	2	3	4
C	1	2	3	3	3	4

Class work

- Find the longest common subsequence
  - XMJYAUZ" and "MZJAWXU

Class work

- Knapsack problem

	A	B	C	D	E	F	G
pts	10	5	10	8	7	3	7
time	3	1	2	3	2	2	3