Analysis of Algorithms

V. Adamchik

CSCI 570

Lecture 7

University of Southern California

Dynamic Programming

Reading: chapter 6

Exam - I

Date: Thursday March 11

Time: starts at 5pm Pacific time

Locations: online, DEN Quiz

Practice: posted

TA Review: March 9 and March 10

Open book and notes
Use scratch paper
No internet searching
No talking to each other (chat, phone, messenger)

REVIEW QUESTIONS

1. (**T/F**) For a divide-and-conquer algorithm, it is possible that the dividing step takes asymptotically longer time than the combining step.
2. (**T/F**) A divide-and-conquer algorithm acting on an input size of n can have a lower bound less than $\Theta(n \log n)$.
3. (**T/F**) There exist some problems that can be efficiently solved by a divide-and-conquer algorithm but cannot be solved by a greedy algorithm.

4. (**T/F**) It is possible for a divide-and-conquer algorithm to have an exponential runtime.

5. (**T/F**) A divide-and-conquer algorithm is always recursive.

6. (**T/F**) The master theorem can be applied to the following recurrence: $T(n) = 1.2\,T(n/2) + n$.
7. (**T/F**) The master theorem can be applied to the following recurrence: $T(n) = 9\,T(n/3) - n^2 \log n + n$.
8. (**T/F**) Karatsuba's algorithm reduces the number of multiplications from four to three.
9. (**T/F**) The runtime complexity of mergesort can be asymptotically improved by recursively splitting an array into three parts (rather than into two parts).

10. (**T/F**) Two $n \times n$ matrices of integers are multiplied in $\Theta(n^2)$ time.

11. (**Fill in the blank**) Let A, B be two 2×2 matrices that are multiplied using the standard multiplication method and Strassen's method.

    a. Number of multiplications in the standard method: __________

    b. Number of additions in the standard method: __________

    c. Number of multiplications using Strassen's method: __________

    d. Number of additions using Strassen's method: __________

12. (**Fill in the blank**) The space complexity of Strassen's algorithm is: __________.

Fibonacci Numbers

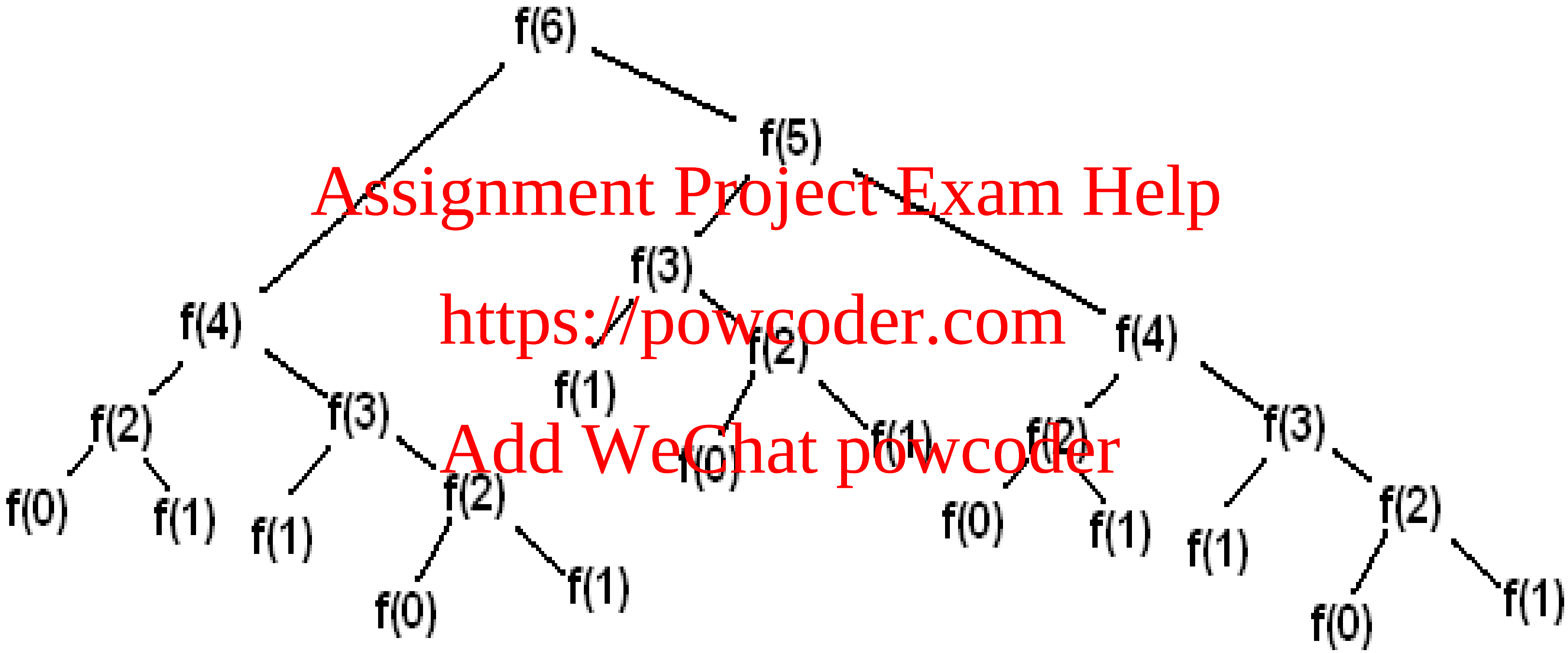
Fibonacci number F_n is defined as the sum of two previous Fibonacci numbers:

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = F_1 = 1$$

Design a divide & conquer algorithm to compute Fibonacci numbers. What is its runtime complexity?

Overlapping Subproblems



Fibonacci Numbers: $F_n = F_{n-1} + F_{n-2}$

Memoization

```
int  table [50];    //initialize to zero
table[0] = table[1] = 1;

int  fib(int  n)
{
        if  (table[n] != 0) return  table[n];
        else
        table[n] = fib(n-1) + fib(n-2);

        return  table[n];
}
```


Runtime complexity?

Tabulation

```
int  table [n];

void fib(int n)
{
        table[0] = table[1] = 1;
        for(int k = 2; k < n; k++)
            table[k] = table[k-1] + table[k-2];

        return;
}
```

Two Approaches

```
int  table [n];
table[0] = table[1] = 1;

int  fib(int  n)
{
   if  (table[n] != 0)
      return  table[n];
   else
      table[n] = fib(n-1) + fib(n-2);

   return  table[n];
}
```

Memoization:
a top-down approach.

```
int  table [n];

int[]  fib(int n)
{
   table[0] = table[1] = 1;
   for(int k = 2; k < n; k++)
      table[k]=table[k-1]+table[k-2];

   return  table;
}
```

Tabulation:
a bottom-up approach.

Dynamic Programming

General approach: in order to solve a larger problem, we solve smaller subproblems and store their values in a table.

DP is applicable when the subproblems are greatly overlap. Compare with Mergesort.

DP is not greedy either. DP tries <u>every</u> choice before solving the problem. It is much more expensive than greedy.

DP can be implemented by means of memoization or tabulation.

Dynamic Programming

Optimal substructure means that the solution can be obtained by the combination of optimal solutions to its subproblems. Such optimal substructures are usually described recursively.

Overlapping subproblems means that the space of subproblems must be small, so an algorithm solving the problem should solve the same subproblems over and over again.

Dynamic Programming

The term dynamic programming was originally used in the 1950s by Richard Bellman.

The term computer (dated from 1613) meant a person performing mathematical calculations.

In the 30-50s those early computers were mostly *women* who used painstaking calculations on paper and later punch cards.

The earliest human computers

Who put a man to the moon?

0-1 Knapsack Problem

Given a set of n unbreakable *unique* items, each with a weight w_k and a value v_k, determine the subset of items such that the total weight is less or equal to a given knapsack capacity W and the total value is as large as possible.

Decision Tree

$$x_k = \begin{cases} 1, \text{item } k \text{ selected} \\ 0, \text{item } k \text{ not selected} \end{cases}$$

Subproblems

Recurrence Formula

Tracing the Algorithm

n = 4, W = 5

(weight, value) = (2,3), (3,4), (4,5), (5,6)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
1,2	0					
1,2,3	0					
1,2,3,4	0					

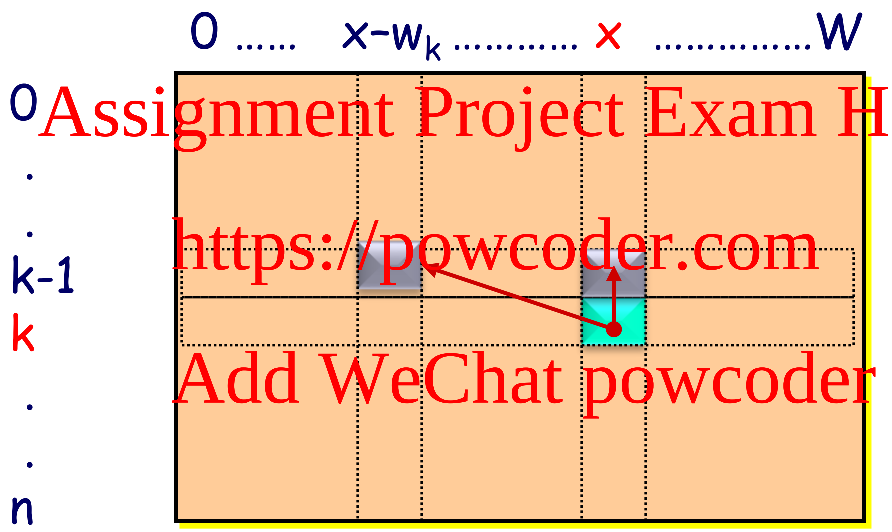
knapsack

items

Pseudo-code

```
int knapsack(int W, int w[], int v[], int n) {
   int Opt[n+1][W+1];
   for (k = 0; k <= n; k++) {
       for (x = 0; x <= W; x++) {
           if (k==0 || x==0) Opt[k][x] = 0;
           if (w[x] > x) Opt[k][x] = Opt[k-1][x];
           else
            Opt[k][x] = max( v[k] + Opt[k-1][x - w[k-1]],
                                            Opt[k-1][x] );
       }
   }
   return Opt[n][W];
}
```

Complexity



Runtime Complexity?

Space Complexity?

Pseudo-Polynomial Runtime

Definition. A numeric algorithm runs in pseudo-polynomial time if its running time is polynomial in the numeric value of the input, but is exponential in the length of the input.

0-1 Knapsack is pseudo-polynomial algorithm, $T(n) = \Theta(n \cdot W)$

Discussion Problem 2

The table built in the algorithm does not show the optimal items, but only the maximum value. How do we find which items give us that optimal value?

$n = 4$, $W = 5$

(weight, value) = (2, 3), (3, 4), (4, 5), (5, 6)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

DP Approach

solve using the following four steps:

1. Define (in plain English) subproblems to be solved.

2. Write the recurrence relation for subproblems.

3. Write pseudo-code to compute the optimal value.
4. Compute the runtime of the above DP algorithm in terms of the input size.

Discussion Problem 3

You are to compute the minimum number of coins needed to make change for a given amount m. Assume that we have an unlimited supply of coins. All denominations d_k are sorted in ascending order:

$$1 = d_1 < d_2 < \dots < d_n$$

Longest Common Subsequence

We are given string S_1 of length n, and string S_2 of length m.

Our goal is to produce their longest common subsequence.

A *subsequence* is a subset of elements in the sequence taken in order (with strictly increasing indexes.) Or you may think as removing some characters from one string to get another.

Intuition

A B A Z D C

B A C B A D

Subproblems

Recurrence

Example

S = ABAZDC
T = BACBAD

		B	A	C	B	A	D
	0	0	0	0	0	0	0
A	0						
B	0						
A	0						
Z	0						
D	0						
C	0						

Pseudo-code

```
int LCS(char[] S1, int n, char[] S2, int m)
{
int table[n+1, m+1];
table[0…n, 0] = table[0, 0…m] = 0; //init
for(i = 1; i <= n; i++)
  for(j = 1; j <= m; j++)
     if (S1[i] == S2[j])  table[i, j] = 1 + table[i-1, j-1]
     else
     table[i, j] = max(table[i, j-1], table[i-1, j]);

return table[n, m];
}
```

How much space do we need?

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

How do we find the common sequence?

	0	0	0	0	0	0	0
	0	0	1	1	1	1	1
	0	1	1	1	2	2	2
	0	1	2	2	2	3	3
	0	1	2	2	2	3	3
	0	1	2	2	2	3	4
	0	1	2	3	3	3	4

Discussion Problem 4

A subsequence is palindromic if it reads the same left and right. Devise a DP algorithm that takes a string and returns the length of the longest palindromic subsequence (not necessarily contiguous)
.

For example, the string

QRAECCETCAURP

has several palindromic subsequences, RACECAR is one of them.

Discussion Problem 5

You are to plan the spring 2022 schedule of classes. Suppose that you can sign up for as many classes as you want, and you'll have infinite amount of energy to handle all the classes, but you cannot have any time conflict between the lectures. Also assume that the problem reduces to planning your schedule of *one particular day*. Thus, consider one particular day of the week and all the classes happening on that day: $c_1, .., c_n$. Associated with each class c_i is a start time s_i and a finish time f_i and you also assign a score v_i to that class based on your interests and your program requirement. Assume $s_i < f_i$. You would like to choose a set of courses C for that day to *maximize* the total score. Devise an algorithm for planning your schedule.