CSEN 703 - Analysis and Design of Algorithms

Lecture 6 - Dynamic Programming I

Dr. Nourhan Ehab
nourhan.ehab@guc.edu.eg

Department of Computer Science and Engineering
Faculty of Media Engineering and Technology

Outline

Optimization Problems

- The most challenging problems involve optimization.

Optimization Problems

- The most challenging problems involve optimization.
- Greedy algorithms can handle optimization problems, but are only suboptimal.

Optimization Problems

- The most challenging problems involve optimization.
- Greedy algorithms can handle optimization problems, but are only suboptimal.
- Exhaustive search are guaranteed to be optimal, but come at a hindering cost.

Optimization Problems

- The most challenging problems involve optimization.
- Greedy algorithms can handle optimization problems, but are only suboptimal.
- Exhaustive search are guaranteed to be optimal, but come at a hindering cost.
- We seek to combine the best of both worlds with Dynamic Programming (DP).

Origins

Figure: Dr. Richard Ernest Bellman (1953)

Idea

Simplify a complicated problem by breaking it down into simpler sub-problems in a recursive manner, then find the solution of the bigger problem by remembering the optimal solutions of the sub-problems.

The Fibonacci Sequence

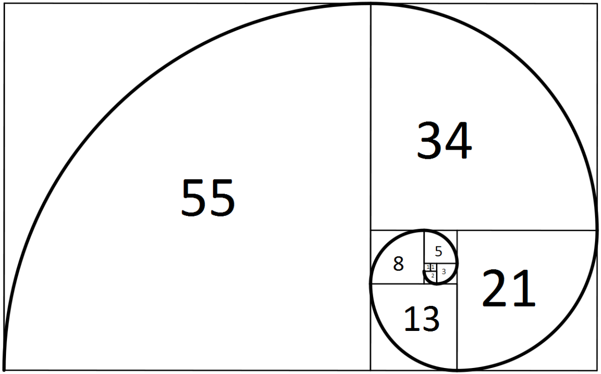
Example

The Fibonacci numbers were defined by the Italian mathematician Fibonacci in the thirteenth century to model the growth of rabbit populations. Rabbits breed, well, like rabbits. Fibonacci surmised that the number of pairs of rabbits born in a given month is equal to the number of pairs of rabbits born in each of the two previous months, starting from one pair of rabbits at the start. Thus,

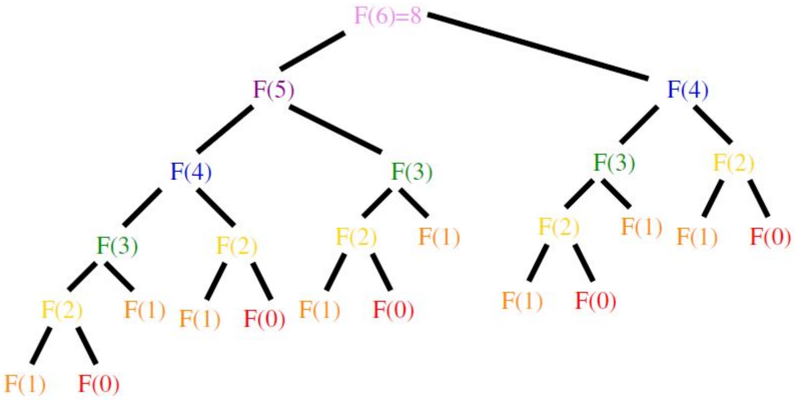
$$F(n) = F(n-1) + F(n-2)$$

with $F(0) = 0$ and $F(1) = 1$.

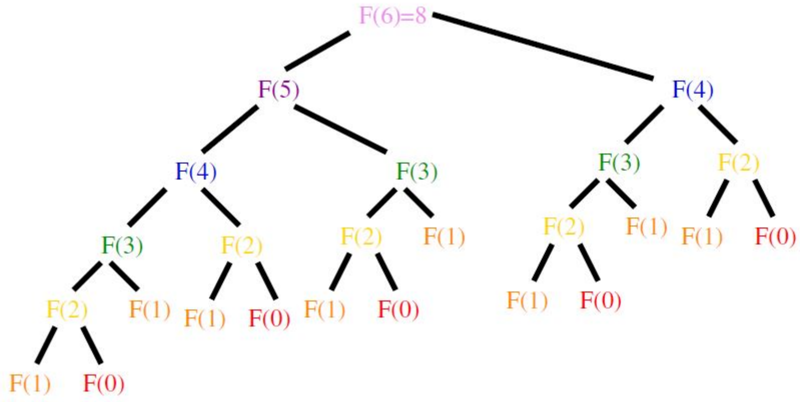
The Fibonacci Sequence



The Fibonacci Sequence

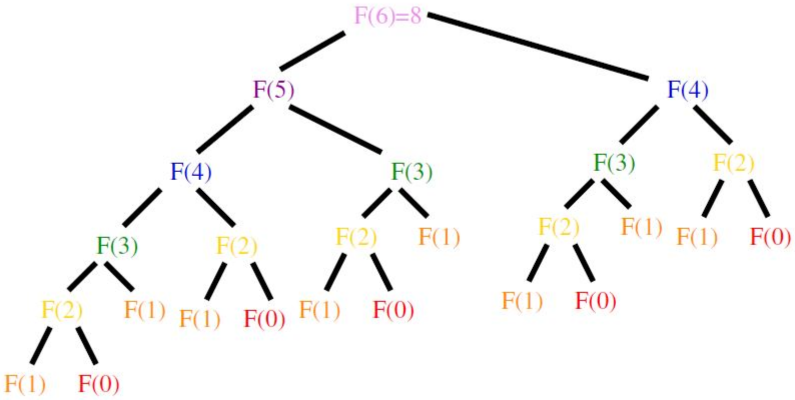


The Fibonacci Sequence



Complexity is $O(2^n)$.

The Fibonacci Sequence



Complexity is $O(2^n)$. But a lot of subproblems are repeating!

Divide and Conquer Vs. Dynamic Programming

- D&C partitions the problem into disjoint subproblems, solve them recursively, then combine their solutions to solve the original problem.

Divide and Conquer Vs. Dynamic Programming

GUC German University in Cairo

- D&C partitions the problem into disjoint subproblems, solve them recursively, then combine their solutions to solve the original problem.
- When the problems overlap, D&C does more work than necessary.

Divide and Conquer Vs. Dynamic Programming

- D&C partitions the problem into disjoint subproblems, solve them recursively, then combine their solutions to solve the original problem.
- When the problems overlap, D&C does more work than necessary.
- Dynamic Programming is all about learning from your past. We save the intermediate results for future reference.

Divide and Conquer Vs. Dynamic Programming

GUC German University in Cairo

- D&C partitions the problem into disjoint subproblems, solve them recursively, then combine their solutions to solve the original problem.
- When the problems overlap, D&C does more work than necessary.
- Dynamic Programming is all about learning from your past. We save the intermediate results for future reference.
- In this way, DP is just D&C+Caching.

DP = D&C+Caching

Einstein: Never memorize something you can look up
Person who invented Dynamic Programming:

Elements of DP

DP is typically used when a problem exhibits the following two properties.

1. Optimal substructure: the optimal solutions to the subproblems can be used to construct the optimal solution of the bigger problem.
2. Overlapping subproblems: the bigger solutions involves solving a lot of repeating subproblems recursively.

Elements of DP

DP is typically used when a problem exhibits the following two properties.

1. Optimal substructure: the optimal solutions to the subproblems can be used to construct the optimal solution of the bigger problem.
2. Overlapping subproblems: the bigger solutions involves solving a lot of repeating subproblems recursively.

Is DP always better than D&C?

Two Approaches of DP

1. Top-Down Approach: write a regular recursive function, but modify it to remember the values of what it already computed. $\Rightarrow$ Memoization

Two Approaches of DP

1. Top-Down Approach: write a regular recursive function, but modify it to remember the values of what it already computed. $\Rightarrow$ Memoization
2. Bottom-Up Approach: solve the smaller subproblems first and store their results in a table so that when solving bigger problems we are sure we solved all the prerequisites (which can be acquired from the table). $\Rightarrow$ Tabulation

Fibonacci D&C Solution

```
Fib(n)
if n == 0 then
    return 0;
else
    if n ≤ 2 then
        return 1;
    else
        return Fib(n − 1) + Fib(n − 2);
    end
end
```

Fibonacci DP Solution - Top-Down

```
a = empty array of integers of size n + 1 initialized with zeros;
Fib-DP-Mem(n)
if n ≤ 2 and n! = 0 then
    a[n] = 1;
else
    if a[n] == 0 and n! = 0 then
        a[n] = Fib-DP-Mem(n − 1) + Fib-DP-Mem(n − 2);
    end
end
return a[n];
```

Fibonacci DP Solution - Bottom-Up

```
Fib-DP-Tab(n)
a=empty array of integers of size n + 1 initialized with zeros;
a[1] = 1;
a[2] = 1;
for i = 3 to n do
    a[i] = a[i − 1] + a[i − 2] ;
end
return a[n];
```

Complexity reduces to just $O(n)$!

Big Picture of DP

Key Takeaway

To design a DP solution to a problem, you need to follow 4 steps.

Big Picture of DP

Key Takeaway

To design a DP solution to a problem, you need to follow 4 steps.

1. Characterize optimal substructure.

Big Picture of DP

Key Takeaway

To design a DP solution to a problem, you need to follow 4 steps.

1. Characterize optimal substructure.
2. Break the problem into smaller sub-problems.

Big Picture of DP

GUC German University in Cairo

Key Takeaway

To design a DP solution to a problem, you need to follow 4 steps.

1. Characterize optimal substructure.
2. Break the problem into smaller sub-problems.
3. Solve these sub-problems optimally and store their results.

Big Picture of DP

GUC German University in Cairo

Key Takeaway

To design a DP solution to a problem, you need to follow 4 steps.

1. Characterize optimal substructure.
2. Break the problem into smaller sub-problems.
3. Solve these sub-problems optimally and store their results.
4. Use the solutions to the subproblems to construct an optimal solution for the original problem.

Outline

GUC German University in Cairo

The Coins Change Problem

Example

Given a value of money V, we want to make a change for V and we have an infinite supply of each of the coin denominations d. That is, if $d = \{5, 10, 20, 25\}$ valued coins. What is the minimum number of coins needed to make the change? For example, if $V = 40$, the minimum number of coins needed is $2\ (20 + 20)$.

Coins Change: Optimal Substructure

$$c[i] = \begin{cases} 0 & \text{if } i = 0 \\ 1 + min_{\forall d[j] \leq i} \ \{c[i - d[j]]\} & otherwise \end{cases}$$

Coins Change: Memoization Solution

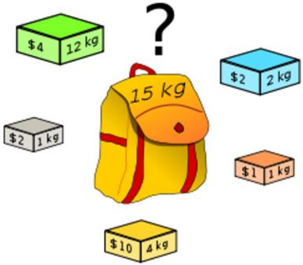
```
change = empty array of size C+1;
coins = empty array of size C+1;
coins-DP(int C, int[ ]d)
if C == 0 then
|   return 0;
else
|   if change[C] == 0 then
|   |   min =MAX_INT;    coin = 0;
|   |   for j = 0 to d.length do
|   |   |   if C − d[j] >= 0 then
|   |   |   |   temp = 1+coins-DP(C − d[j]) ;
|   |   |   |   if temp < min then
|   |   |   |   |   min = temp;    coin = d[j];
|   |   |   |   end
|   |   |   end
|   |   end
|   |   change[C] = min;
|   |   coins[C] = coin;
|   end
|   return change[C]
end
reconstruct-solution(C, coins);
```

Coins Change: Reconstructing Solution

```
reconstruct-solution(int C, int[] coins)
if C == 0 then
    return
else
    print coins[C] ;
    reconstruct-solution(C − coins[C], coins)
end
```

Outline

Problem Statement



Example

We are given a set S of n items, such that each item i has a positive value v_i and a positive weight w_i. We wish to find the subset with the maximum total value that does not exceed a given weight W. We can not take a fraction of any item.

0/1 Knapsack DP Table

Suppose $W = 5$, $w = [2, 5, 3]$, and $v = [30, 20, 40]$.

0/1 Knapsack DP Table

Suppose $W = 5$, $w = [2, 5, 3]$, and $v = [30, 20, 40]$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0 (0)	30 (1)	30 (1)	30 (1)	30 (1)
2	0	0 (0)	30 (0)	30 (0)	30 (0)	30 (0)
3	0	0 (0)	30 (0)	40 (1)	40 (1)	70 (1)

0/1 Knapsack: Optimal Substructure

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w[k] > w \\ max \begin{cases} B[k-1,w] \\ B[k-1, w-w[k]] + v[k] \end{cases} & otherwise \end{cases}$$

0/1 Knapsack: Tabulation Solution

```
0/1-Knapsack(int[ ]w, int[ ]v, int W)
n = w.length; B = empty 2D array of size [n + 1][W + 1];
store = empty 2D array of size [n + 1][W + 1];
for i = 0 to n do
    for j = 0 to W do
        if i == 0 or j == 0 then
            B[i][j] = 0;
        else
            if  w[i] > j then
                B[i][j] = B[i − 1][j];      store[i][j] = 0 ;
            else
                if B[i − 1][j] < B[i − 1][j − w[i]] + v[i] then
                    B[i][j] = B[i − 1][j − w[i]] + v[i];      store[i][j] = 1 ;
                else
                    B[i][j] = B[i − 1][j];      store[i][j] = 0 ;
                end
            end
        end
    end
end
reconstruct-solution(n, W,store);
```

0/1 Knapsack: Reconstructing Solution

```
reconstruct-solution(int n, int W, int[][]store)
for i = n down to 1 do
    if store[i][W] == 1 then
        print i;
        W = W - w[i] ;
    end
end
```

Outline

1. Dynamic Programming
2. Coins Change
3. 0/1 Knapsack
4. Recap

Conclusion: DP vs Greedy Algorithms

DP	Greedy
Makes a choice at each step after solving subproblems.	Makes a choice at each step before solving subproblems.
Considers all possibilities.	Considers only one branch.
Always Optimal.	Suboptimal.

Points to Take Home

1. The Gist of Dynamic Programming.
2. Memoization vs Tabulation approaches.
3. Fibonacci Sequence Problem.
4. Coins Change Problem.
5. 0/1 Knapsack Problem.
6. Reading Material:
    - The Algorithm Design Manual. Chapter 10, Section 10.1.
    - Algorithm Design and Applications by Goodrich and Tamasia. Chapter 12 Section 12.6.

Next Lecture: More on DP!

Due Credits

The presented material is based on:

1. Previous editions of the course at the GUC due to Dr. Wael Aboulsaadat, Dr. Haythem Ismail, Dr. Amr Desouky, and Dr. Carmen Gervet.
2. Stony Brook University's Analysis of Algorithms Course.
3. MIT's Introduction to Algorithms Course.
4. Stanford's Design and Analysis of Algorithms Course.