Dynamic Programming

Chapter 12

Acknowledgement

- I have used some slides from
  - https://www.cis.upenn.edu/~matuszek/
  - cit594-2014/Lectures/30-dynamic-programming.ppt
  - http://web.stanford.edu/class/cs97si/

What is DP?

- Wikipedia definition: "method for solving complex problems by breaking them down into simpler subproblems"

- This definition will make sense once we see some examples
  - Actually, we'll only see problem solving examples today

Steps for Solving DP Problems

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

- Each step is very important!

Dynamic Programming

- Dynamic programming is a very powerful, general tool for solving optimization problems.
- Once understood it is relatively easy to apply, but many people have trouble understanding it.

Greedy Algorithms

- Greedy algorithms focus on making the best local choice at each decision point.
- For example, a natural way to compute a shortest path from x to y might be to walk out of x, repeatedly following the cheapest edge until we get to y. WRONG!
- In the absence of a correctness proof greedy algorithms are very likely to fail.

Problem:

Let's consider the calculation of **Fibonacci** numbers:

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

with seed values *F(1) = 1, F(2) = 1*

or *F(0) = 0, F(1) = 1*

What would a series look like:

0, 1, 1, 2, 3, 4, 5, 8, 13, 21, 34, 55, 89, 144, …

Recursive Algorithm:

```
Fib(n)
{
   if (n == 0)
      return 0;

   if (n == 1)
      return 1;

  Return Fib(n-1)+Fib(n-2)
}
```

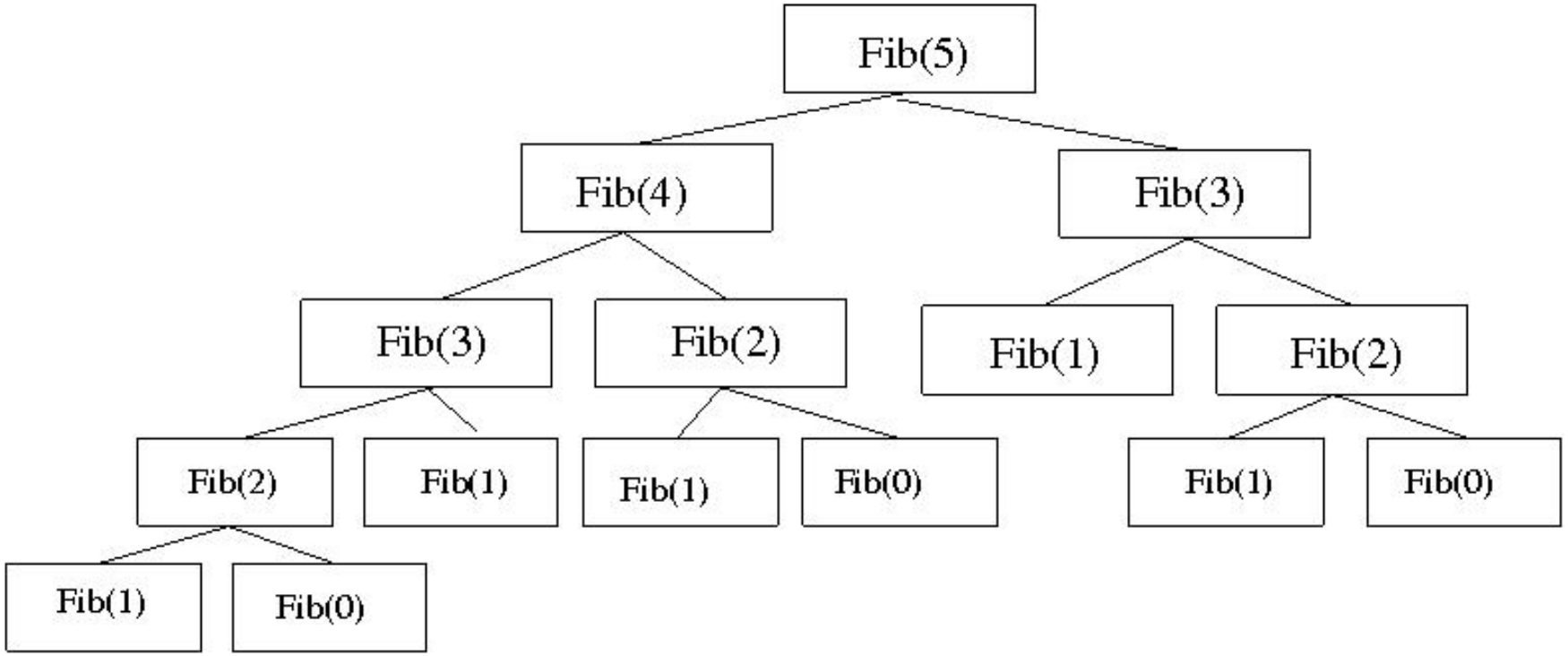
Recursive Algorithm:

```
Fib(n)
{
   if (n == 0)
      return 0;

   if (n == 1)
      return 1;

  Return Fib(n-1)+Fib(n-2)
}
```

It has a serious issue!

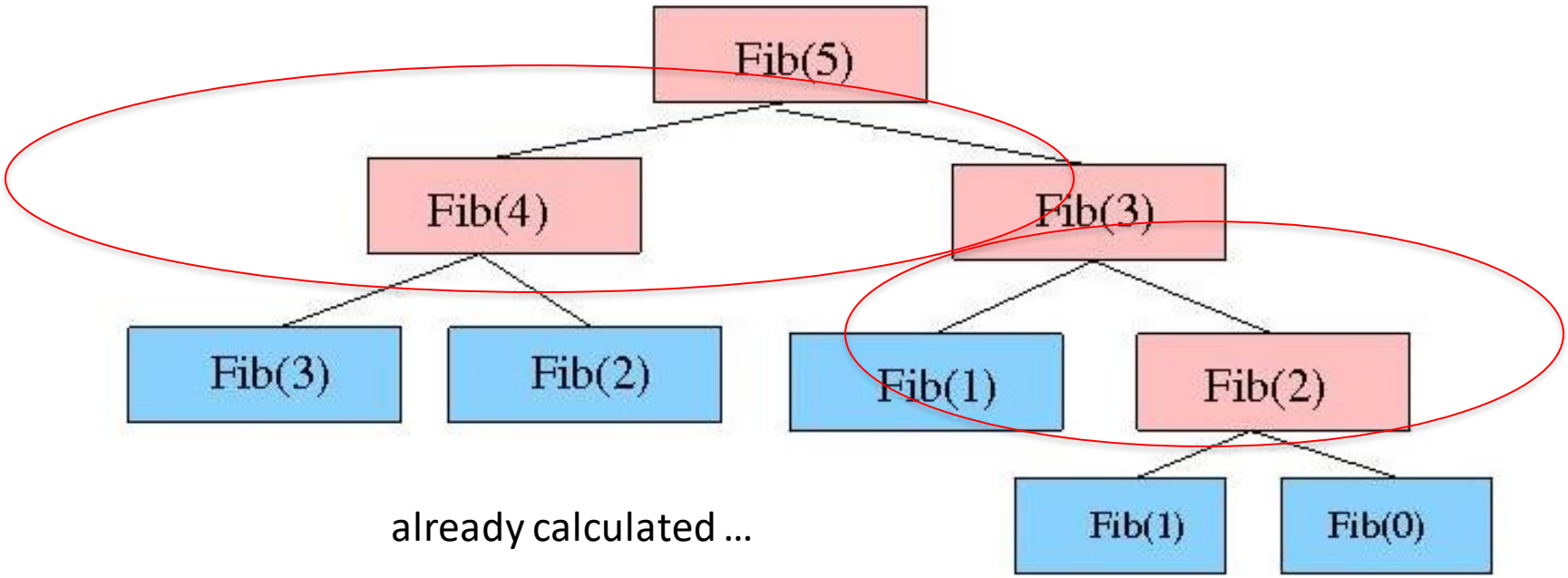
Recursion tree

What's the problem?



Memoization:

```
Fib(n)
{
   if (n == 0)
      return M[0];

   if (n == 1)
      return M[1];

   if (Fib(n-2) is not already calculated)
      call Fib(n-2);

   if(Fib(n-1) is already calculated)
      call Fib(n-1);

   //Store the ${n}^{th}$ Fibonacci no. in memory & use previous results.
   M[n] = M[n-1] + M[n-2]

   Return M[n];
}
```



already calculated …

Dynamic programming

- Main approach: recursive, holds answers to a sub problem in a table, can be used without recomputing.

- Can be formulated both via recursion and saving results in a table (*memoization*). Typically, we first formulate the recursive solution and then turn it into recursion plus dynamic programming via *memoization* or bottom-up.

-"*programming*" as in tabular not programming code

1-dimensional DP Problem

- Problem: 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

$$
\begin{aligned}
5 &= 1+1+1+1+1 \\
&= 1+1+3 \\
&= 1+3+1 \\
&= 3+1+1 \\
&= 1+4 \\
&= 4+1
\end{aligned}
$$

1-dimensional DP Problem

- Define subproblems
  - Let D_n be the number of ways to write n as the sum of 1, 3, 4
- Find the recurrence
  - Consider one possible solution $n = x_1 + x_2 + \cdots + x_m$
  - If $x_m = 1$, the rest of the terms must sum to $n - 1$
  - Thus, the number of sums that end with $x_m = 1$ is equal to D_{n-1}
  - Take other cases into account ($x_m = 3$, $x_m = 4$)

1-dimensional DP Problem

- Recurrence is then

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

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

- We're basically done!

1-dimensional DP Problem

```
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];
```

- Very short!

What happens when n is extremely large?

Extension: Solving this for huge n,
$n \approx 10^{12}$!

We have $D(n) = D(n-1) + D(n-3) + D(n-4)$, $n \geq 4$

$D(0) = D(1) = D(2) = 1$

$D(3) = 2$.

We can write

$$\begin{bmatrix} D(n) \\ D(n-1) \\ D(n-2) \\ D(n-3) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} D(n-1) \\ D(n-2) \\ D(n-3) \\ D(n-4) \end{bmatrix}$$

Let $A = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

$\therefore$ 猜 $\begin{bmatrix} D(n) \\ D(n-1) \\ D(n-2) \\ D(n-3) \end{bmatrix} = A \begin{bmatrix} D(n-1) \\ D(n-2) \\ D(n-3) \\ D(n-4) \end{bmatrix}$ —(1)

We can write again

$$\begin{bmatrix} D(n-1) \\ D(n-2) \\ D(n-3) \\ D(n-4) \end{bmatrix} = A \begin{bmatrix} D(n-2) \\ D(n-3) \\ D(n-4) \\ D(n-5 \end{bmatrix}$$

$\therefore$ (1) becomes

$$\begin{bmatrix} D(n) \\ D(n-2) \\ D(n-3) \\ D(n-4) \end{bmatrix} = A^2 \begin{bmatrix} D(n-2) \\ D(n-3) \\ D(n-4) \\ D(n-5) \end{bmatrix} = \cdots$$

$$= A^{n-3} \begin{bmatrix} D(3) \\ D(2) \\ D(1) \\ D(0) \end{bmatrix}$$

Need to evaluate A^{n-3}

Evaluate (A^k)

Step 1: if $k=1$ return A^k

Step 2: Compute $B = A^{\lfloor k/2 \rfloor}$

Step 3: If k is even return B^2

else return $B^2 . A$

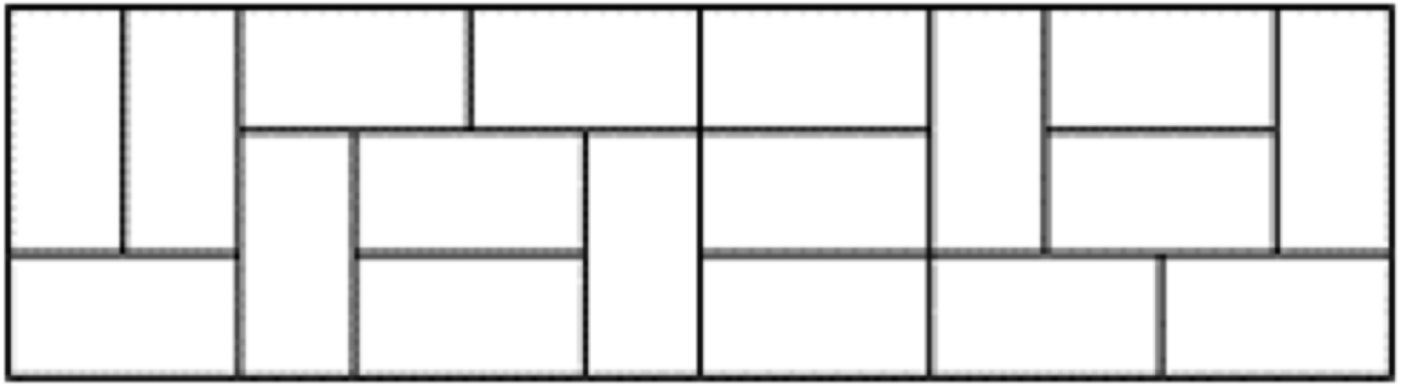
Total cost: $O(\log n)$. ~~[illegible]~~ #

Logarithmic time

Tri Tiling

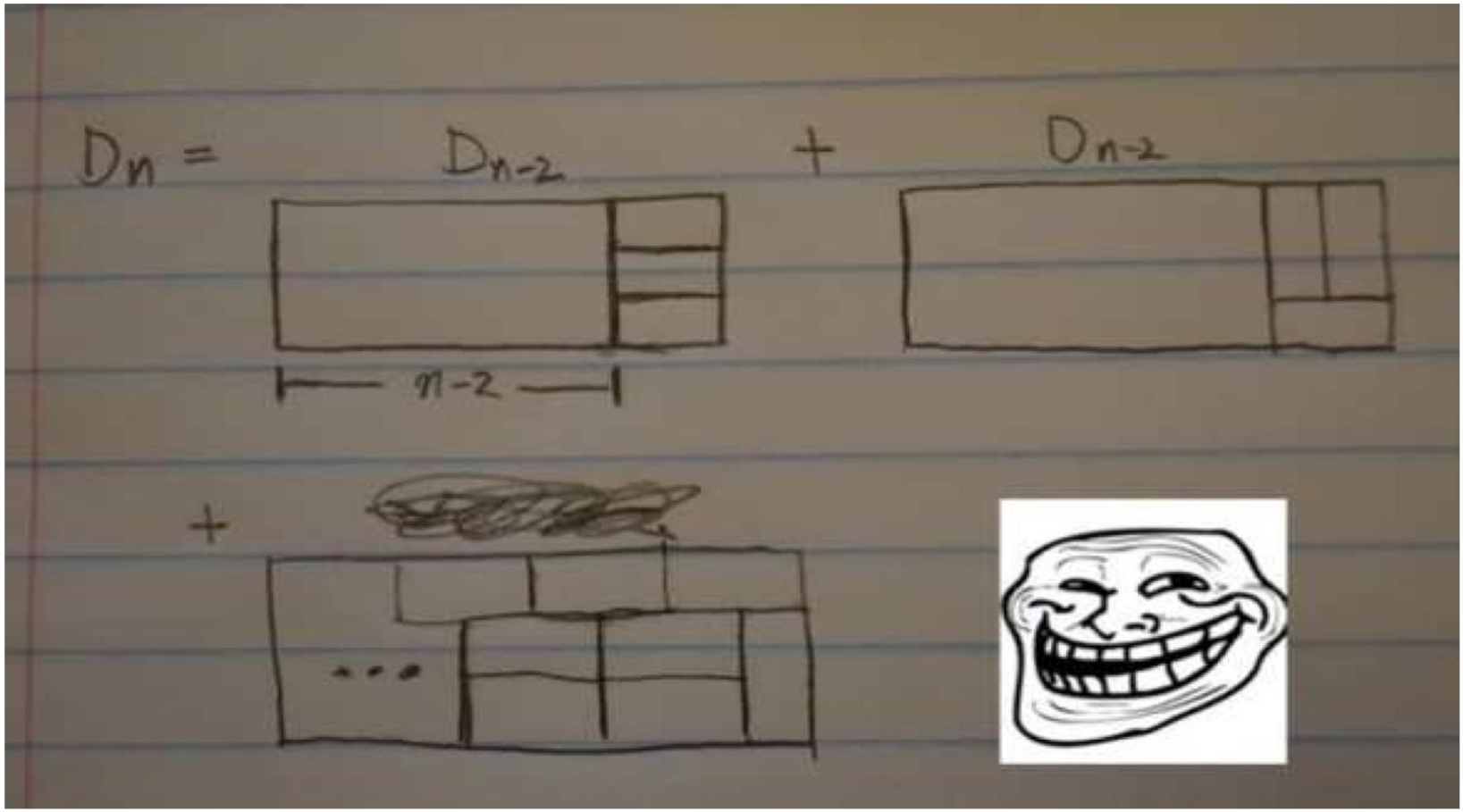
- Given n, find the number of ways to fill a $3 \times n$ board with dominoes

- Here is one possible solution for $n = 12$



Tri Tiling

- Define subproblems
  - Define D_n as the number of ways to tile a $3 \times n$ board

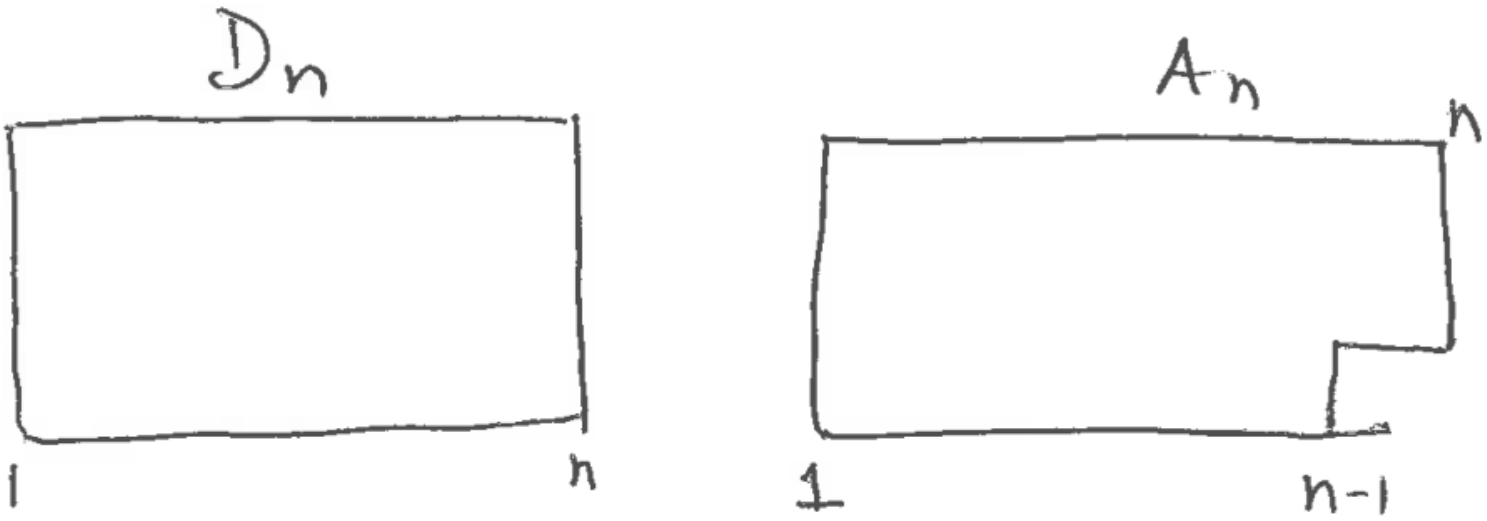


Tri Tiling

- Obviously, the previous definition didn't work very well
- D_n's don't relate in simple terms

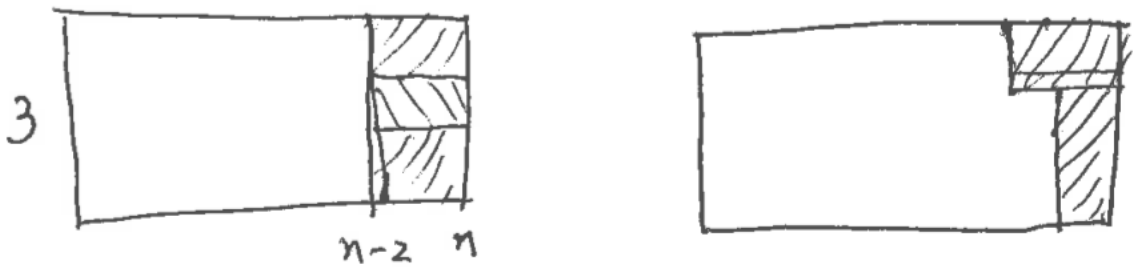
- What if we introduce more subproblems?

Defining Subproblems

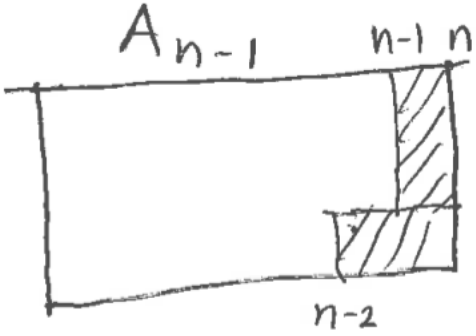


Finding Recurrences

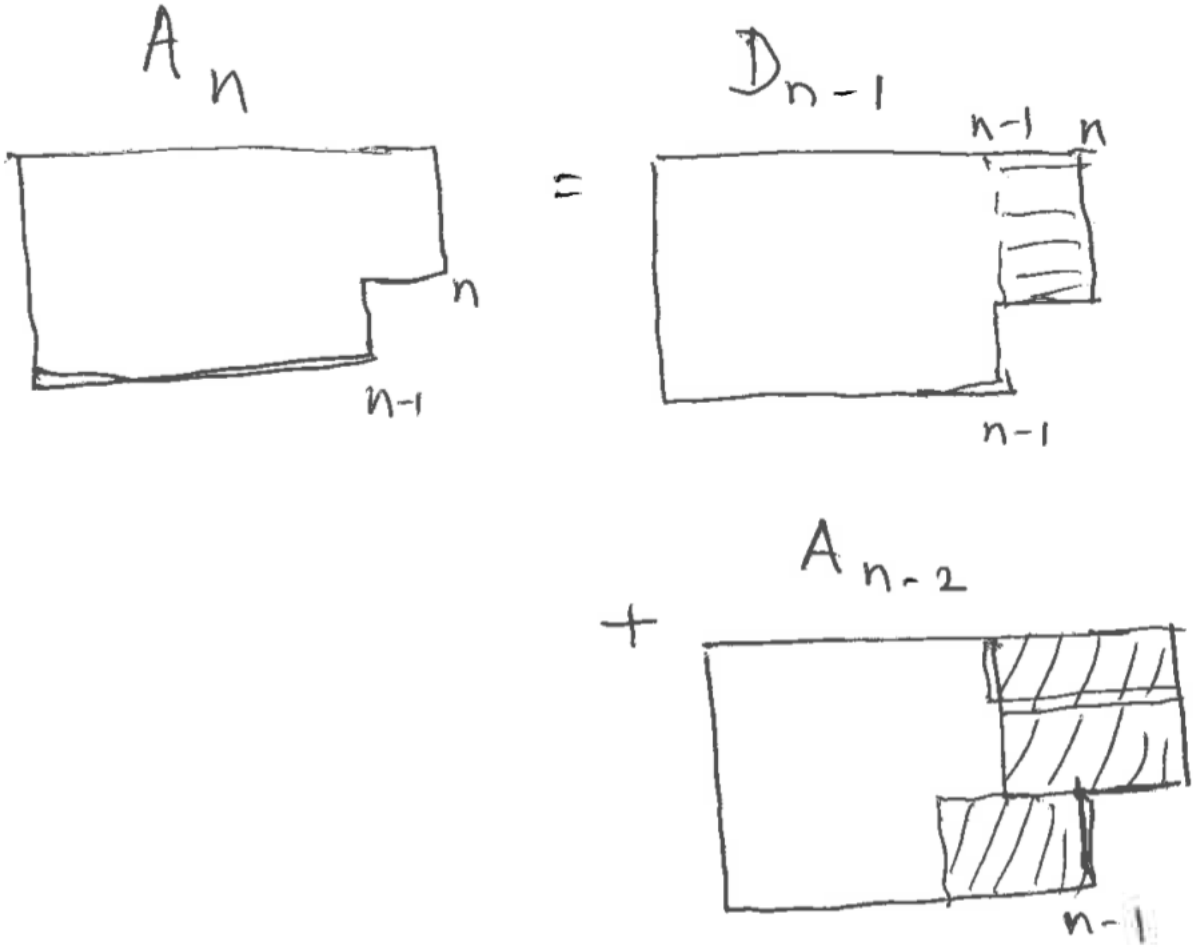
$D_n =$ D_{n-2} + A_{n-1}



+ A_{n-1}



Finding Recurrences



$D_0 = 0$; $D_1 = \phi$; $A_0 = 0$ $A_1 = 0$

Extension

- Solving the problem for n x m grids, where n is small, say $n \leq 10$.
  - How many subproblems do we consider?

Egg dropping problem

You have a building with *n* floors, and you have 1 egg.

For some floor *F*, if the egg is dropped from a height of at least *F* it will break and be unusable in the future.

Determine *F* with the minimal number of egg drops in the worst case.

Egg dropping problem

We cannot afford to have the egg break. We must search floor 1 first, then floor 2, etc until the egg breaks. When the egg breaks, we know F.

What would we do if we had 2 eggs?

Egg dropping problem

With 2 eggs we can afford to have an egg break to get us close to our goal.

A good paradigm to try is the classic binary search lens. Does it work on this problem?

Egg dropping problem

Here's another strategy: start at floor 10. If it breaks, linear search floors $1 - 9$ with the other egg. Otherwise, try floor 20. Continue in $10's$ until the egg breaks, then search by $1's$.

Worst case: egg breaks on floor 99. We try

$$10, 20, 30, 40, 50, 60, 70, 80, 90, 100$$

(10 floors), then

$$91, 92, 93, 94, 95, 96, 97, 98, 99$$

(9 more floors)

Much better than 50 floors!

Egg dropping problem

What we really want to do with 'binary search' is try some floor that gives us roughly equal 'time left to solve' no matter what the outcome of the drop is (break or not).

Let T be the number of trials required (in the worst case) to solve the problem when we drop the egg from some floor G. We want something like:

$$T(G, EGG_{BREAK}) = T(G, EGG_{INTACT})$$

Egg dropping problem

If we drop the egg at floor G to start and the egg survives, the next floor to drop an egg at is floor $2G - 1$.

Why?

After dropping the egg once (and it survives), we pick some new floor. We must guarantee that we can do the linear search component using at most $G - 1$ steps, so we add $G - 1$ floors to our current floor G.

After dropping the egg twice (it survives both times), we must guarantee that we can do the linear search component using at most $G - 2$ steps. This egg is now dropped at $3G - 3$.

Egg dropping problem

Now we apply our math lens to find a formula for this pattern:

$$G + (G-1) + (G-2) + ... + 1 = G(G+1)/2$$

So by starting at floor G, we can definitely find F if it lies in the range $[1, G(G+1)/2]$. Since this takes G moves, the optimal solution requires us to use the lowest G such that $G(G+1)/2 >= n$

This yields an O(1) solution.

Egg dropping problem(n eggs) Dynamic Programming Approach

- D[j,m] : There are j floors and m eggs. Like to find the floors with the largest value from which an egg, when dropped doesn't crack.

For the original problem, the recurrence might be:

$$DP[n, e] = \arg\min_{g}\{1 + max(DP[g-1, e-1], DP[n-g, e])\}$$

Here the egg cracked when dropped from floor g.

Egg dropping problem(n eggs) Dynamic Programming Approach

- DP[j,m] : There are j floors and m eggs. Like to find the floors with the largest value from which an egg, when dropped doesn't crack.

For the original problem, the recurrence r

Here the egg didn't crack when dropped from floor g.

$$DP[n, e] = \arg\min_{g}\{1 + max(DP[g-1, e-1], DP[n-g, e])\}$$

Here the egg cracked when dropped from floor g.

Egg dropping problem(n eggs) Dynamic Programming Approach

- DP[j,m] : There are j floors and m eggs. Like to find the floors with the largest value from which an egg, when dropped doesn't crack.

For the original problem, the recurrence might be:

$$DP[n, e] = \arg\min_{g}\{1 + max(DP[g-1, e-1], DP[n-g, e])\}$$

DP[1,e] = 0 for all e (Base case)

2-dimensional DP Example

- Problem: given two strings x and y, find the longest common subsequence (LCS) and print its length
- Example:
  - x: A**BC**BD**AB**
  - y: **B**D**CAB**C
  - "BCAB" is the longest subsequence found in both sequences, so the answer is 4

Solving the LCS Problem

- Define subproblems
  - Let D_{ij} 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
    - $D_{ij} = D_{i-1,j-1} + 1$
  - Otherwise, either x_i or y_j does not contribute to the LCS, so one can be dropped
    - $D_{ij} = \max\{D_{i-1,j}, D_{i,j-1}\}$
  - Find and solve the base cases: $D_{i0} = D_{0j} = 0$

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]);
    }
}
```

Tree DP Example

- Problem: given a tree, color nodes black as many as possible without coloring two adjacent nodes

- Subproblems:
  - First, 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
  - Answer is $\max\{B_r, W_r\}$

Tree DP Example

- Find the recurrence
  - Crucial observation: once v's color is determined, subtrees can be solved independently
  - If v is colored, its children must not be colored

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

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

$$W_v = 1 + \sum_{u \in \text{children}(v)} \max\{B_u, W_u\}$$

- Base cases: leaf nodes

Subset DP Example

- Problem: given a weighted graph with n nodes, find the shortest path that visits every node exactly once (Traveling Salesman Problem)

- Wait, isn't this an NP-hard problem?
  - Yes, but we can solve it in $O(n^2 2^n)$ time
  - Note: brute force algorithm takes $O(n!)$ time

Subset DP Example

- Define subproblems
  - $D_{S,v}$: the length of the optimal path that visits every node in the set S exactly once and ends at v
  - There are approximately $n2^n$ subproblems
  - Answer is $\min_{v \in V} D_{V,v}$, where V is the given set of nodes

- Let's solve the base cases first
  - For each node v, $D_{\{v\},v} = 0$

Subset DP Example

- Find the recurrence
  - Consider a path that visits all nodes in S exactly once and ends at v
  - Right before arriving v, the path comes from some u in $S - \{v\}$
  - And that subpath has to be the optimal one that covers $S - \{v\}$, ending at u
  - We just try all possible candidates for u

$$D_{S,v} = \min_{u \in S - \{v\}} \Big(D_{S-\{v\},u} + \text{cost}(u, v) \Big)$$

Working with Subsets

- When working with subsets, it's good to have a nice representation of sets
- Idea: Use an integer to represent a set
  - Concise representation of subsets of small integers $\{0, 1, \ldots\}$
  - If the ith (least significant) digit is 1, i is in the set
  - If the ith digit is 0, i is not in the set
  - *e.g.*, $19 = 010011_{(2)}$ in binary represent a set $\{0, 1, 4\}$

Coin-change Problem

- To find the minimum number of Canadian coins to make any amount, the greedy method always works.
  - At each step select the largest denomination not going over the desired amount.

Coin-change Problem

- The greedy method doesn't work if we didn't have 5¢ coin.
  - For 31¢, the greedy solution is 25 +1+1+1+1+1+1
  - But we can do it with 10+10+10+1
- The greedy method also wouldn't work if we had a 21¢ coin
  - For 63¢, the greedy solution is 25+25+10+1+1+1
  - But we can do it with 21+21+21

Coin set for examples

- For the following examples, we will assume coins in the following denominations:

  1¢ 5¢ 10¢ 21¢ 25¢

- We'll use 63¢ as our goal

A solution

- We can reduce the problem recursively by choosing the first coin, and solving for the amount that is left
- For 63¢:
  - One 1¢ coin plus the best solution for 62¢
  - One 5¢ coin plus the best solution for 58¢
  - One 10¢ coin plus the best solution for 53¢
  - One 21¢ coin plus the best solution for 42¢
  - One 25¢ coin plus the best solution for 38¢
- Choose the best solution from among the 5 given above
- We solve 5 recursive problems.
- This is a very expensive algorithm

A dynamic programming solution

- Idea: Solve first for one cent, then two cents, then three cents, etc., up to the desired amount
  - *Save each answer in an array !*
- For each new amount N, combine a selected pairs of previous answers which sum to N
  - For example, to find the solution for 13¢,
    - First, solve for all of 1¢, 2¢, 3¢, ..., 12¢
    - Next, choose the best solution among:
      - Solution for 1¢ + solution for 12¢
      - Solution for 5¢ + solution for 8¢
      - Solution for 10¢ + solution for 3¢

A dynamic programming solution

- Let T(n) be the number of coins taken to dispense n¢.
- The recurrence relation
  - $T(n) = \min \{T(n-1), T(n-5), T(n-10), T(n-25)\} + 1, n \geq 26$
  - $T(c)$ is known for $n \leq 25$
- It is exponential if we are not careful.
- The bottom-up approach is the best.
- Memoization idea also can be used.

A dynamic programming solution

- The dynamic programming algorithm is O(N*K) where N is the desired amount and K is the number of different kind of coins.

Comparison with divide-and-conquer

- Divide-and-conquer algorithms split a problem into separate subproblems, solve the subproblems, and combine the results for a solution to the original problem
  - Example: Quicksort
  - Example: Mergesort
  - Example: Binary search
- Divide-and-conquer algorithms can be thought of as top-down algorithms

Comparison with divide-and-conquer

- In contrast, a dynamic programming algorithm proceeds by solving small problems, remembering the results, then combining them to find the solution to larger problems
- Dynamic programming can be thought of as bottom-up

The principle of optimality, I

- Dynamic programming is a technique for finding an *optimal* solution
- The principle of optimality applies if the optimal solution to a problem can be obtained by combining the optimal solutions to all subproblems.

The principle of optimality, I

- Example: Consider the problem of making N¢ with the fewest number of coins
  - Either there is an N¢ coin, or
  - The set of coins making up an optimal solution for N¢ can be divided into two nonempty subsets, n_1¢ and n_2¢
    - If either subset, n_1¢ or n_2¢, can be made with fewer coins, then clearly N¢ can be made with fewer coins, hence solution was *not* optimal

The principle of optimality, II

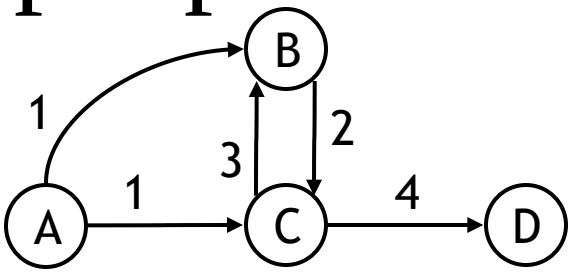
- The principle of optimality holds if
  - Every optimal solution to a problem contains...
  - ...optimal solutions to all subproblems
- The principle of optimality does *not* say
  - If you have optimal solutions to all subproblems...
  - ...then you can combine them to get an optimal solution

The principle of optimality, II

- Example: In coin problem,
  - The optimal solution to 7¢ is 5¢ + 1¢ + 1¢, *and*
  - The optimal solution to 6¢ is 5¢ + 1¢, *but*
  - The optimal solution to 13¢ is *not* 5¢ + 1¢ + 1¢ + 5¢ + 1¢
- But there is *some* way of dividing up 13¢ into subsets with optimal solutions that will give an optimal solution for 13¢
  - Hence, the principle of optimality holds for this problem

Longest simple path

- Consider the following graph:



- The longest simple path (path not containing a cycle) from A to D is A B C D
- However, the subpath A B is not the longest simple path from A to B (A C B is longer)
- The principle of optimality is not satisfied for this problem
- Hence, the longest simple path problem cannot be solved by a dynamic programming approach

- Example: In coin problem,
  - The optimal solution to 7¢ is 5¢ + 1¢ + 1¢, *and*
  - The optimal solution to 6¢ is 5¢ + 1¢, *but*
  - The optimal solution to 13¢ is *not* 5¢ + 1¢ + 1¢ + 5¢ + 1¢
- But there is *some* way of dividing up 13¢ into subsets with optimal solutions that will give an optimal solution for 13¢
  - Hence, the principle of optimality holds for this problem

The 0-1 knapsack problem

- A thief breaks into a house, carrying a knapsack...
  - He can carry up to 25 pounds of loot
  - He has to choose which of N items to steal
    - Each item has some weight and some value
    - "0-1" because each item is stolen (1) or not stolen (0)
  - He has to select the items to steal in order to maximize the value of his loot, but cannot exceed 25 pounds

The 0-1 knapsack problem

- A greedy algorithm does not find an optimal solution
- A dynamic programming algorithm works well.

The 0-1 knapsack problem

- This is similar to, but not identical to, the coins problem
  - In the coins problem, we had to make an *exact* amount of change
  - In the 0-1 knapsack problem, we can't *exceed* the weight limit, but the optimal solution may be *less* than the weight limit
  - The dynamic programming solution is similar to that of the coins problem

Steps for Solving DP Problems

- Define subproblems
- Write down the recurrence that relates subproblems
- Recognize and solve the base cases

- Each step is very important.

Comments

- Dynamic programming relies on working "from the bottom up" and saving the results of solving simpler problems
  - These solutions to simpler problems are then used to compute the solution to more complex problems
- Dynamic programming solutions can often be quite complex and tricky

Comments

- Dynamic programming is used for optimization problems, especially ones that would otherwise take exponential time
  - Only problems that satisfy the principle of optimality are suitable for dynamic programming solutions
- Since exponential time is unacceptable for all but the smallest problems, dynamic programming is sometimes essential.