Dynamic programming

Dynamic programming is an algorithm design technique for solving problems that have:

- an optimal substructure property (recursion)
- overlapping subproblems

Idea: Do not repeatedly solve the same subproblems, instead solve them only once and store the solutions in a **dynamic programming table**.

Example: Fibonacci numbers 0, 1, 1, 2, 3, 5, 8, 13, 21, ..., n

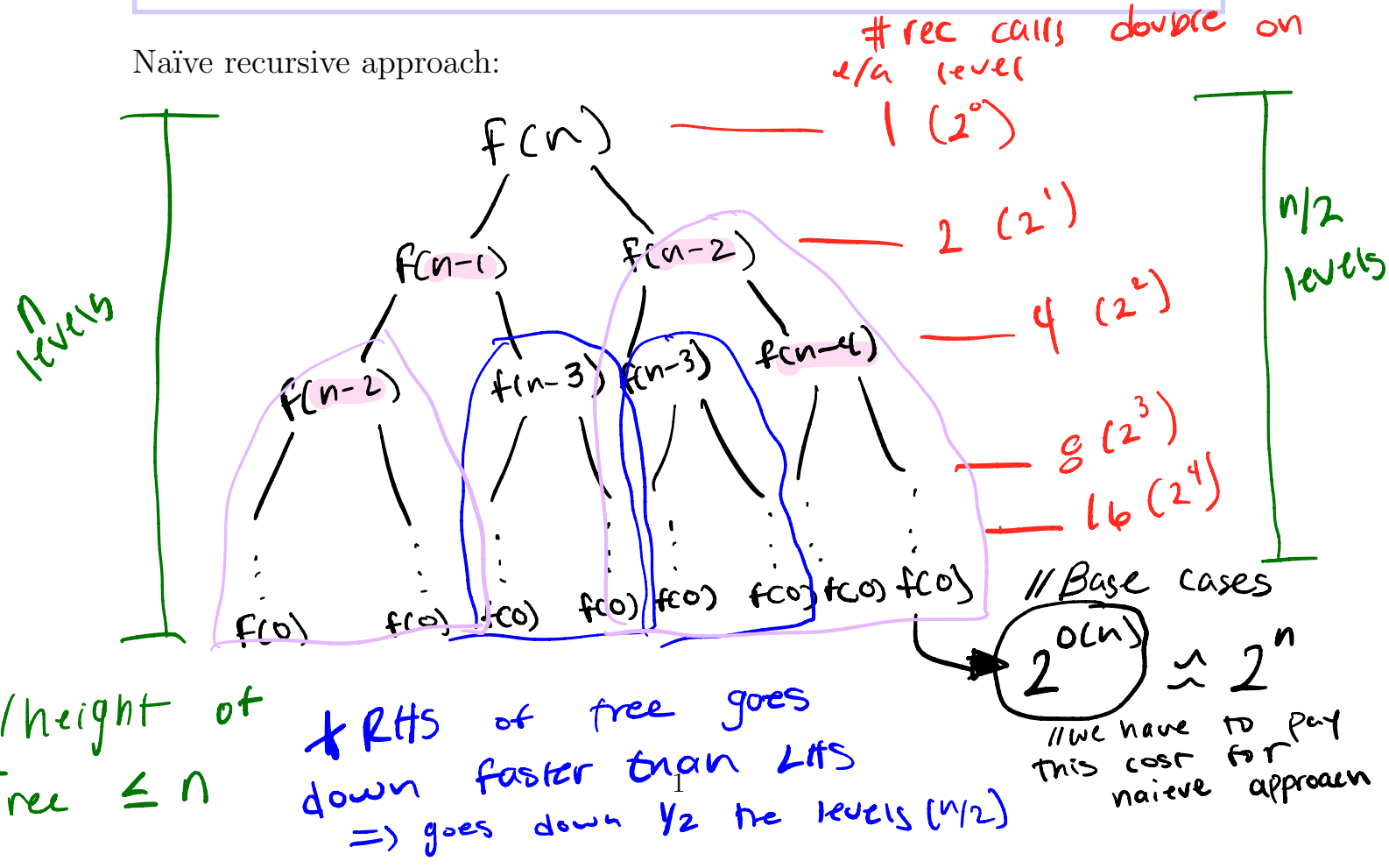
Base case: $F(0) = 0$ $F(1) = 1$

Recursive case: $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$ // 2 recursive calls

Dynamic Programming Hallmark #1:

An optimal solution to the problem depends on one or more optimal solutions to subproblems.

Naïve recursive approach:



=> These subtrees are doing the same amount of work (duplicated work) → tells us we can do better

Runtime of the naïve recursive approach: $O(2^n)$

The number of distinct Fibonacci subproblems is only n. $0, 1, 1, 2, 3, \ldots, n$ (n+1 #'s)

Dynamic Programming Hallmark #2:

A naïve recursive solution has a few distinct subproblems which are solved repeatedly.

There are two variants of dynamic programming:

- Bottom-up dynamic programming (start w/ smaller problems working up to bigger ones)
- Memoization (also called "top-down dynamic programming") (starts w/ big problem seeing how it connects to smaller problems)

Bottom-up dynamic programming:

Store DP-table and fill bottom-up:

0	1	2	3	4	5	6	7		n
0	1	1	2	3	5	8	13	...	

- doesn't use recursion

Algorithm 1 int fibBottomUpDP(int n)

```
1: Create int array F[0...n]
2: F[0] = 0;
3: F[1] = 1;                    } fill base cases
4: for i = 2; i ≤ n; i++ do     } 2, 3, 4, 5, ..., n  O(n)
5:     F[i] = F[i − 1] + F[i − 2];
6: end for
7: return F[n];
```

Runtime: $\Theta(n)$

=> Using an Array to store calculated fib. numbers

=> reduced runtime but we need slightly more memory, as we have an array of n size

Memoization:

Use a recursive algorithm. After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

Algorithm 2 int fibMemoization(int n)

```
1: Create new array F of size n and initialize each element F[i] = null
2: fibMemoizationRec(n,F);
3: return F[n];
```

↳ helper function

Algorithm 3 int fibMemoizationRec(int n, int[] F)

```
1: if F[n] = null then
2:     if n = 0 then
3:         F[n] = 0;
4:     else if n = 1 then
5:         F[n] = 1;
6:     else
7:         F[n] = fibMemoizationRec(n-1,F) + fibMemoizationRec(n-2,F);
8:     end if
9: end if
10: return F[n];
```

Base cases

// How many rec. calls needed?

=> n+1 values, each makes 2 rec. calls beside base cases

=> 0, 1, 1, 2, 3, 5, 8, ...

2 + 2(n-1) = 2n steps
$\Theta(n)$

Time: $\Theta(n)$

Space: $\Theta(n)$ for Array to store saved values

$+\ \Theta(n)$ for recursive calls

$\Theta(n)$

Is this a useful algorithm design technique? Yes, dynamic prog. is very efficient in comparison to the naive approach

=> As always these algorithms serve certain purposes. Bottom-up is a better tool than memoization for this problem.

From Wikipedia: https://en.wikipedia.org/wiki/Dynamic_programming

- Dijkstra's algorithm for finding single source shortest paths.
- The Needleman–Wunsch algorithm and other algorithms used in bioinformatics, including sequence alignment, structural alignment, RNA structure prediction
- The Bellman–Ford algorithm for finding the shortest distance in a graph
- Floyd's all-pairs shortest path algorithm
- Recurrent solutions to lattice models for protein-DNA binding
- Optimizing the order for chain matrix multiplication
- Kadane's algorithm for the maximum subarray problem
- Backward induction as a solution method for finite-horizon discrete-time dynamic optimization problems
- Method of undetermined coefficients can be used to solve the Bellman equation in infinite-horizon, discrete-time, discounted, time-invariant dynamic optimization problems
- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, Levenshtein distance (edit distance)
- Many algorithmic problems on graphs can be solved efficiently for graphs of bounded treewidth or bounded clique-width by using dynamic programming on a tree decomposition of the graph.
- The Cocke–Younger–Kasami (CYK) algorithm which determines whether and how a given string can be generated by a given context-free grammar Knuth's word wrapping algorithm that minimizes raggedness when word wrapping text
- The use of transposition tables and refutation tables in computer chess
- The Viterbi algorithm (used for hidden Markov models, and particularly in part of speech tagging)
- The Earley algorithm (a type of chart parser)
- Pseudo-polynomial time algorithms for the subset sum, knapsack and partition problems
- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. System R) algorithm for relational database query optimization
- De Boor algorithm for evaluating B-spline curves
- Duckworth–Lewis method for resolving the problem when games of cricket are interrupted
- The value iteration method for solving Markov decision processes
- Some graphic image edge following selection methods such as the "magnet" selection tool in Photoshop
- Some methods for solving interval scheduling problems
- Some methods for solving the travelling salesman problem, either exactly (in exponential time) or approximately (e.g. via the bitonic tour)
- Recursive least squares method
- Beat tracking in music information retrieval
- Adaptive-critic training strategy for artificial neural networks
- Stereo algorithms for solving the correspondence problem used in stereo vision
- Seam carving (content-aware image resizing)
- Some approximate solution methods for the linear search problem
- Optimization of electric generation expansion plans in the Wein Automatic System Planning (WASP) package
- ...and more!

Greedy Algorithms

Making a locally optimal solution at every decision point will yield a globally optimal solution. = Focus on e/a subproblem

Greedy Hallmark #1:

An optimal solution to the problem depends on *exactly one* subproblem's solution.

Example: Make change using fewest number of coins (for the coin denominations: $1, 5, 10, 25$)

Greedy strategy: Choose the largest possible coin. Repeat this until we reach the desired total.

Try for 89 cents:

=> 89 — 25 cents

= 64 — 25 cents

= 39 — 25 cents

= 14 — 10 cents

= 4 — 4 cents (1 cent · 4)

8 coins total

Claim: This is least amount of coins possible for 89 cents

=> Is this correct in every case?

We have to prove that a greedy solution is optimal!

(Otherwise it is just a heuristic)

Correctness proof:

Let $n = 25Q + 10D + 5N + P$ be the optimal solution for n cents.

Thus, $C(n) = Q + D + N + P.$ = # coins used

- $P < 5$

  Suppose $P \geq 5$

  => If $P \geq 5$, we could replace 5 pennies for 1 nickle for a better solution.

- $5N + P < 10$ // money in nickles & pennies < 10

  Suppose $5N + P \geq 10$

  => $P \leq 4$,

  => Thus at least 2 nickles, which can be replaced w/ 1 dime

- $10D + 5N + P < 25$ // money in dimes, nickles, pennies < 25

  Suppose $10D + 5N + P \geq 25$

  // 2 ways this can happen

  Case 1: Suppose we had 1 nickle
  => Then we could replace nickle & 2 dimes w/ 1 quarter

  Case 2: Suppose we had 0 nickles // $10D + 5N + P < 25$
  => Then we could replace the 3 dimes w/ 1 quarter & 1 Nickle

Does this greedy algorithm work for any set of coin denominations?

Try adding a 20 set coin denomination:

$$1, 5, 10, 20, 25$$

How many coins will the previous greedy algorithm use to form $n = 40$ cents?

40 cents — 1 Q
15 cents — 1 D
5 cents — 1 N
} 3 coins

Not optimal

What is the optimal solution?

T := 20 cents

40 cents — 1T
20 cents — 1T
} 2 coins

Return to Dynamic Programming

Let's find a dynamic programming solution for the making change problem. Specifically, suppose we have the following k coin denominations:

$d_1, d_2, \ldots, d_k$] denominations of k coins

$C(n) =$ minimum number of coins necessary to make change for n cents. // C(27) = 3 coins

Base cases:

$C(d_1) = 1 \quad C(d_2) = 1 \quad \ldots \quad C(d_k) = 1$ => // C(1) = 1, C(5) = 1, C(25) = 1

Set $C(n) = \infty$ if $n < 0$ // C(-4) = ∞

Recursive case:

$C(n) = \min_{i=1}^{k}[C(n - d_i) + 1]$ // Try every denomination +1 for coin being used

For example:

Let $d_1 = 1, d_2 = 5, d_3 = 10, d_4 = 20, d_5 = 25$

case where dynamic prog. works

i:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
C(i):	1	2	3	4	1	2	3	4	5	1	2	3	4	5			

2+1=3

1+1=2 3+1=4

10 + 1 coin (cent) = 15

Examples:

$C(1) = 1$ (base case)

min of all cases

$C(2) = \min(1+C(2-1), 1+C(2-5), 1+C(2-10), 1+C(2-20), 1+C(2-25))$

$= \min(1 + C(1),\ 1 + C(-3),\ 1 + C(-8),\ 1 + C(-18),\ 1 + C(-23))$

$= \min(1+1, 1+\infty, 1+\infty, 1+\infty, 1+\infty)$

$= \min(2) = 2$ coins

same solution maybe inefficient here

$C(6) = \min(1 + C(5),\ 1 + C(1),\ 1 + C(-4),\ 1 + C(-14),\ 1 + C(-19))$

$= \min(1+1, 1+1, 1+\infty, \ldots\ldots 1+\infty)$

$= \min(2, 2, \ldots\ldots \infty)$

$= 2$ coins

$C(15) = \min(1 + C(14),\ 1 + C(10),\ 1 + C(5),\ 1 + C(-5),\ 1 + C(-10))$

$= \min(1+5, 1+1, 1+1, \ldots\ldots)$

$= \min(2) = 2$ coins

∴ Table above helps with saving work

Bottom up dynamic programming algorithm:

Algorithm 4 int MinCoinBottomUpDP(int n, int[] d)

```
1: //Let k denote the number of denominations in array d
2: //Base Cases
3: for i = 1; i ≤ k; i++ do
4:     C[d[i]] = 1;
5: end for
6:
7: //Recursive Cases
8: for i = 1; i ≤ n; i++ do
9:     min = ∞;
10:    if i is not a base case then
11:        for j = 1; j ≤ k; j++ do
12:            if d[j] < i and C[i − d[j]] < min then
13:                min = C[i − d[j]];
14:            end if
15:        end for
16:    end if
17:    C[i] = min + 1;
18: end for
19: return [redacted];
```

Slightly simpler alternative using a base case of 0:

Algorithm 5 int MinCoinBottomUpDP(int n, int[] d)

```
1: C[0] = 0;
2: for i = 1; i ≤ n; i++ do
3:     min = ∞;
4:     for j = 1; j ≤ k; j++ do
5:         if d[j] < i and C[i − d[j]] < min then
6:             min = C[i − d[j]];
7:         end if
8:     end for
9:     C[i] = min + 1;
10: end for
11: return [redacted];
```

What is the runtime for the algorithm?

Example DP Problem: Longest Common Subsequence (LCS)

Given two sequences $x[1 \ldots m]$ and $y[1 \ldots n]$, find a longest subsequence common to them both.

x

A	B	C	B	D	A	B

y

B	D	C	A	B	A

Brute-force LCS algorithm:

Check every subsequence of $x[1 \ldots m]$ to see if it is also a subsequence of $y[1 \ldots n]$.

Runtime Analysis

Idea for a better algorithm:

- Look at the length of a longest common subsequence.
- Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence s by $\mid s \mid$.

Strategy: Consider prefixes of x and y.

- Define $c[i, j] = \mid LCS(x[1 \ldots i], y[1 \ldots j]) \mid$.
- Then, $c[m, n] = \mid LCS(x, y) \mid$

Recursive formulation

Theorem.

$c[i, j] =$

Proof. Case $x[i] = y[j]$

Suppose $z[1 \dots k] = LCS(x[1 \dots i], y[1 \dots j])$.

Therefore, $c[i, j] =\mid z \mid=\mid LCS(x[1 \dots i], y[1 \dots j]) \mid$.

It must be the case that $z[k] = x[i] = y[j]$

Thus, $z[1 \dots k-1]$ is common subsequence of $x[1 \dots i-1]$ and $y[1 \dots j-1]$

We want to show $z[1 \dots k-1]$ is $LCS(x[1 \dots i-1], y[1 \dots j-1])$.

Suppose w is a larger common subsequence of $LCS(x[1 \dots i-1], y[1 \dots j-1])$.

That is to say $\mid w \mid > k-1$.

Then concatenate $z[k]$ onto the end of w. Clearly, $\mid w + z[k] \mid > k$.

This contradicts the assumption that $c[i, j] = k$. Therefore, no such w exists.

Naïve recursive approach based on the previous recurrence:

Algorithm 6 int LCS(String x, String y, int i, int j)

```
1: if i = 0 or j = 0 then
2:     return 0;
3: end if
4: if x[i] = y[j] then
5:     return LCS(x, y, i − 1, j − 1) + 1;
6: else
7:     return Max{LCS(x, y, i − 1, j), LCS(x, y, i, j − 1)};
8: end if
```

Worst case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion tree for $x = EDCBA$, $y = WXYZ$, $i = 5$, $j = 4$

The number of distinct LCS subproblems for two strings of lengths m and n is only mn.

Thus, use should try to solve this using dynamic programming.

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

Algorithm 7 int LCSMemoization(String x, String y, int m, int n)

```
1: Create new 2d array c of size m by n.
2: Initialize each element c[i, j] = null
3: LCSMemoizationRec(x,y,m,n,c);
4: return c[m, n];
```

Algorithm 8 int LCSMemoizationRec(String x, String y, int i, int j, int[] c)

```
1: if c[i, j] = null then
2:     if i = 0 then
3:         c[i, j] = 0;
4:     else if j = 0 then
5:         c[i, j] = 0;
6:     else if x[i] = y[j] then
7:         c[i, j] =LCSMemoizationRec(x,y,i − 1,j − 1,c)+1;
8:     else
9:         c[i, j] =Max{LCSMemoizationRec(x,y,i − 1,j,c),
10:                    LCSMemoizationRec(x,y,i,j − 1,c)};
11:    end if
12: end if
13: return c[i, j];
```

Simple LCS example:

		B	A	B
A				
B				

Longer LCS example:

		A	B	C	B	D	A	B
B								
D								
C								
A								
B								
A								

Our solution to LCS can extended to solve a variety of problems which require measuring the similarity of two strings:

- Spell-checking
- Optical character recognition
- Comparing DNA sequences
- etc.