DYNAMIC PROGRAMMING: MORE FUN!

David Kauchak
CS 140 – Fall 2022

1

Admin

Assignment 6

2

Where did "dynamic programming" come from?

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes.

"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities" (p. 159).

Richard Bellman On the Birth of Dynamic Programming

Stuart Dreyfus

http://www.eng.tau.ac.il/~ami/cd/or50/1526-5463-2002-50-01-0048.pdf

3

Dynamic programming

Method for solving problems where optimal solutions can be defined in terms of optimal solutions to subproblems

AND

the subproblems are overlapping

4

Dynamic programming: steps

1a) optimal substructure: optimal solutions to the problem incorporate optimal solutions to related subproblems
- convince yourself that there is optimal substructure

1b) recursive definition: use this to recursively define the value of an optimal solution

2) DP solution: describe the dynamic programming table:
- size, initial values, order in which it's filled in, location of solution

3) Analysis: analyze space requirements, running time

5

LCS problem

Given two sequences X and Y, a **common subsequence** is a subsequence that occurs in both X and Y

Given two sequences $X = x_1, x_2, \ldots, x_n$ and $Y = y_1, y_2, \ldots, y_n$

What is the longest common subsequence?

6

2: DP solution

$$LCS(X,Y) = \begin{cases} 1 + LCS(X_{1\ldots n-1}, Y_{1\ldots m-1}) & \text{if } x_n = y_m \\ \max(LCS(X_{1\ldots n-1}, Y), LCS(X, Y_{1\ldots m-1})) & \text{otherwise} \end{cases}$$

What types of subproblem solutions do we need to store?

$LCS(X_{1\ldots j}, Y_{1\ldots k})$

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

7

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

How should we fill in the table?

8

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

	j	0	1	2	3	4	5	6
i	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0			?			
4	B	0						
5	D	0						
6	A	0						
7	B	0						

To fill in an entry, we may need to look:
- up one
- left one
- diagonal up and left

Just need to make sure these exist

9

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

	j	0	1	2	3	4	5	6
i	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	?					
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS(A, B)

10

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

	j	0	1	2	3	4	5	6
i	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0					
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

11

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

	j	0	1	2	3	4	5	6
i	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	?		
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS(A, BDCA)

12

$$LCS[i,j]=\begin{cases}1+LCS[i-1,j-1] & \text{if } x_i=y_j\\ \max(LCS[i-1,j],LCS[i,j-1]) & \text{otherwise}\end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1		
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS(A, BDCA)

13

$$LCS[i,j]=\begin{cases}1+LCS[i-1,j-1] & \text{if } x_i=y_j\\ \max(LCS[i-1,j],LCS[i,j-1]) & \text{otherwise}\end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	?	
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCAB)

14

$$LCS[i,j]=\begin{cases}1+LCS[i-1,j-1] & \text{if } x_i=y_j\\ \max(LCS[i-1,j],LCS[i,j-1]) & \text{otherwise}\end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCAB)

15

$$LCS[i,j]=\begin{cases}1+LCS[i-1,j-1] & \text{if } x_i=y_j\\ \max(LCS[i-1,j],LCS[i,j-1]) & \text{otherwise}\end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Where's the final answer?

16

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Space requirements?

Running time?

17

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Space requirements: $\Theta(nm)$

Running time: $\Theta(nm)$

18

Keeping track of the solution

Our LCS algorithm only calculated the length of the LCS between X and Y

What if we wanted to know the actual sequence?

19

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	?	
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCAB)

20

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1,j], LCS[i,j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCAB)

21

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1,j], LCS[i,j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	?
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCABA)

22

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1,j], LCS[i,j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCABA)

23

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1,j], LCS[i,j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

How do we generate the solution from this?

24

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

We can follow the arrows to generate the solution

BCBA

25

Rod splitting

Input: a length n and a table of prices for $i = 1, 2, \dots m$

Output: maximum revenue obtainable by cutting up the rod and selling the pieces

Example:

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

26

1a: optimal substructure

Prove: optimal solutions to the problem incorporate optimal solutions to related subproblems

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

What would a solution look like?

27

1a: optimal substructure

Prove: optimal solutions to the problem incorporate optimal solutions to related subproblems

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

$$\{l_1, l_2, l_3, \dots, l_m\} \text{ where } \sum_{i=1}^{m} l_i \le n$$

What would a subproblem solution look like?

28

1a: optimal substructure

Prove: optimal solutions to the problem incorporate optimal solutions to related subproblems

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

$$\{l_1, l_2, l_3, \ldots, l_m\} \text{ where } \sum_{i=1}^{m} l_i \leq n$$

$$\{l_2, l_3, \ldots, l_m\} \text{ where } \sum_{i=2}^{m} l_i \leq n - l_1$$

29

1a: optimal substructure

Prove: optimal solutions to the problem incorporate optimal solutions to related subproblems

Proof by contradiction:

Assume: $\{l_1, l_2, l_3, \ldots, l_m\}$ is a solution to n, but $\{l_2, l_3, \ldots, l_m\}$ is **not** a solution to $n - l_1$

If that were the case, then some solution to $n - l_1$ exists where the the sum of the prices of the lengths is greater than that for $\{l_2, l_3, \ldots, l_m\}$.

We could add l_1 to this subproblem solution and get a better solution to the n problem... contradiction

30

1b: recursive solution

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

n 

What should be the first cut?

What are the options?

31

1b: recursive solution

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

n 

cut 1
price 1

How much is left?

32

1b: recursive solution

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

n 

cut 1
price 1

n-1

33

1b: recursive solution

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

n 

cut 2
price 5

n-2

34

1b: recursive solution

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

n 

…

Which one should we choose?

Pretend like we have a solver (R) that gives us the answer to suproblems.

What would R take as input and return?

35

1b: recursive solution

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

n 

…

R(x) → price for best set of cuts of length x

(could structure it with the actual cuts, but focusing on just the price is easier for now)

36

1b: recursive solution

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

n 

cut 1
price 1

n-1

What's the best we can do with this cut?

37

1b: recursive solution

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

n 

cut 1
price 1

n-1

1 + R(n-1)

38

1b: recursive solution

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

n 

cut 2
price 5

n-2

What's the best we can do with this cut?

39

1b: recursive solution

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

n 

cut 2
price 5

n-2

5 + R(n-2)

40

1b: recursive solution

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

n 

What should be the first cut?

41

1b: recursive solution

length	i	1	2	3	4	5	6	7	8	9	10
price	p_i	1	5	8	9	10	17	17	20	24	30

n 

$$R(n) = \max_{i:n-l_i\geq 0}\{p_i + R(n-l_i)\}$$

42

2: DP solution (from the bottom-up)

$$R(n) = \max_{i:n-l_i\geq 0}\{p_i + R(n-l_i)\}$$

What are the smallest possible subproblems?

To calculate R(n), what are all the subproblems we need to calculate? This is the “table”.

How should we fill in the table?

Where will the answer be?

43

2: DP solution (from the bottom-up)

$$R(n) = \max_{i:n-l_i\geq 0}\{p_i + R(n-l_i)\}$$

What are the smallest possible subproblems?
R(0) = 0, R(-i) not possible

44

2: DP solution (from the bottom-up)

$$R(n) = \max_{i: n - l_i \geq 0} \{p_i + R(n - l_i)\}$$

To calculate R(n), what are all the subproblems we need to calculate? This is the "table". R(0)…R(n)

Note: This is filling in a table for all possible integer lengths from 1 to n.

45

2: DP solution (from the bottom-up)

$$R(n) = \max_{i: n - l_i \geq 0} \{p_i + R(n - l_i)\}$$

How should we fill in the table?
R(0) → R(n)

The dependencies are on smaller values

46

2: DP solution (from the bottom-up)

$$R(n) = \max_{i: n - l_i \geq 0} \{p_i + R(n - l_i)\}$$

Where will the answer be?
R(n)

47

2: DP solution

```
DP-Rod-Splitting(n)
   r[0] = 0
   for j = 1 to n
      max = 0
      for i = 1 to m
         if l_i ≤ j
            p = p_i + r[j − l_i]
            if p > max
               max = p

      r[j] = max

   return r[n]
```

48

3: Analysis

```
DP-Rod-Splitting(n)
   r[0] = 0
   for j = 1 to n
      max = 0
      for i = 1 to m
         if l_i ≤ j
            p = pi + r[j − li]
            if p > max
               max = p

      r[j] = max

   return r[n]
```

Space requirements?

Running time?

49

3: Analysis

```
DP-Rod-Splitting(n)
   r[0] = 0
   for j = 1 to n
      max = 0
      for i = 1 to m
         if l_i ≤ j
            p = pi + r[j − li]
            if p > max
               max = p

      r[j] = max

   return r[n]
```

Space requirements: $\Theta(n)$

Running time: $\Theta(nm)$

50

0-1 Knapsack problem

0-1 Knapsack – A thief robbing a store finds *m* items worth $v_1, v_2, .., v_m$ dollars and weight $w_1, w_2, \ldots, w_m$ pounds, where v_i and w_i are integers. The thief can carry at most W pounds in the knapsack. Which items should the thief take if they want to maximize value?

Repetition is allowed, that is you can take multiple copies of any item

51

1a: optimal substructure

Prove: optimal solutions to the problem incorporate optimal solutions to related subproblems

Proof by contradiction:

Assume: $\{i_1, i_2, i_3, \ldots, i_k\}$ is a solution to W but $\{i_2, i_3, \ldots, i_k\}$ is **not** a solution to W – w_{i_1}

Then some solution to W – w_{i_1} exists, $\{i'_2, i'_3, \ldots, i'_n\}$ where the sum of the values of the items is greater than for $\{i_2, i_3, \ldots, ik\}$

We could create a solution $\{i_1, i'2, i'3, \ldots, i'_n\}$ to the original problem that has more value… contradiction

52

1b: recursive solution

$$K(w) = \max_{i: w_i \le w} \{K(w - w_i) + v_i\}$$

53

2: DP solution (from the bottom-up)

$$K(w) = \max_{i: w_i \le w} \{K(w - w_i) + v_i\}$$

What are the smallest possible subproblems?
K(0) = 0

To calculate K(w), what are all the subproblems we need to calculate? This is the "table". K(0) … K(W)

How should we fill in the table? K(1) → K(W)

Where will the answer be? K(W)

54

3: Analysis

$$K(w) = \max_{i: w_i \le w} \{K(w - w_i) + v_i\}$$

What are the smallest possible subproblems? K(0) = 0

To calculate K(w), what are all the subproblems we need to calculate? This is the "table". K(0) … K(W)

How should we fill in the table? K(0) → K(W)

Where will the answer be? K(W)

Space requirements: Θ(W)

Running time: Θ(Wm)

55

Memoization

Sometimes it can be a challenge to write the function in a bottom-up fashion

Memoization:
- Write the recursive function top-down
- Alter the function to check if we've already calculated the value
- If so, use the pre-calculate value
- If not, do the recursive call(s)

56

Memoized fibonacci

```
FIBONACCI(n)
  if n = 1 or n = 2
      return 1
  else
      return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

```
FIBONACCI-MEMOIZED(n)
  fib[1] ← 1
  fib[2] ← 1
  for i ← 3 to n
      fib[i] ← ∞
  return FIB-LOOKUP(n)

FIB-LOOKUP(n)
  if fib[n] < ∞
      return fib[n]
  x ← FIB-LOOKUP(n − 1) + FIB-LOOKUP(n − 2)
  if x < fib[n]
      fib[n] ← x
  return fib[n]
```

57

Memoized fibonacci

```
FIBONACCI(n)
  if n = 1 or n = 2
      return 1
  else
      return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

```
FIBONACCI-MEMOIZED(n)
  fib[1] ← 1
  fib[2] ← 1
  for i ← 3 to n
      fib[i] ← ∞
  return FIB-LOOKUP(n)

FIB-LOOKUP(n)
  if fib[n] < ∞
      return fib[n]
  x ← FIB-LOOKUP(n − 1) + FIB-LOOKUP(n − 2)
  if x < fib[n]
      fib[n] ← x
  return fib[n]
```

Use ∞ to denote uncalculated

58

Memoized fibonacci

```
FIBONACCI(n)
  if n = 1 or n = 2
      return 1
  else
      return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

```
FIBONACCI-MEMOIZED(n)
  fib[1] ← 1
  fib[2] ← 1
  for i ← 3 to n
      fib[i] ← ∞
  return FIB-LOOKUP(n)

FIB-LOOKUP(n)
  if fib[n] < ∞
      return fib[n]
  x ← FIB-LOOKUP(n − 1) + FIB-LOOKUP(n − 2)
  if x < fib[n]
      fib[n] ← x
  return fib[n]
```

What else could we use besides an array?

Use ∞ to denote uncalculated

59

Memoized fibonacci

```
FIBONACCI(n)
  if n = 1 or n = 2
      return 1
  else
      return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

```
FIBONACCI-MEMOIZED(n)
  fib[1] ← 1
  fib[2] ← 1
  for i ← 3 to n
      fib[i] ← ∞
  return FIB-LOOKUP(n)

FIB-LOOKUP(n)
  if fib[n] < ∞
      return fib[n]
  x ← FIB-LOOKUP(n − 1) + FIB-LOOKUP(n − 2)
  if x < fib[n]
      fib[n] ← x
  return fib[n]
```

Check if we already calculated the value

60

Memoized fibonacci

```
FIBONACCI(n)
  if n = 1 or n = 2
      return 1
  else
      return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

```
FIBONACCI-MEMOIZED(n)
  fib[1] ← 1
  fib[2] ← 1
  for i ← 3 to n
      fib[i] ← ∞
  return FIB-LOOKUP(n)

FIB-LOOKUP(n)
  if fib[n] < ∞
      return fib[n]
  x ← FIB-LOOKUP(n − 1) + FIB-LOOKUP(n − 2)
  if x < fib[n]
      fib[n] ← x
  return fib[n]
```

calculate the value

61

Memoized fibonacci

```
FIBONACCI(n)
  if n = 1 or n = 2
      return 1
  else
      return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

```
FIBONACCI-MEMOIZED(n)
  fib[1] ← 1
  fib[2] ← 1
  for i ← 3 to n
      fib[i] ← ∞
  return FIB-LOOKUP(n)

FIB-LOOKUP(n)
  if fib[n] < ∞
      return fib[n]
  x ← FIB-LOOKUP(n − 1) + FIB-LOOKUP(n − 2)
  if x < fib[n]
      fib[n] ← x
  return fib[n]
```

store the value

62

Memoization

Pros

- Can be more intuitive to code/understand
- Can be memory savings if you don't need answers to all subproblems

Cons

- Depending on implementation, larger overhead because of recursion (though often the functions are tail recursive)

63