Faculty of Information Technology, Monash University

FIT2004: Algorithms and Data Structures

Week 4: Dynamic Programming

Lecturer: Reza Haffari

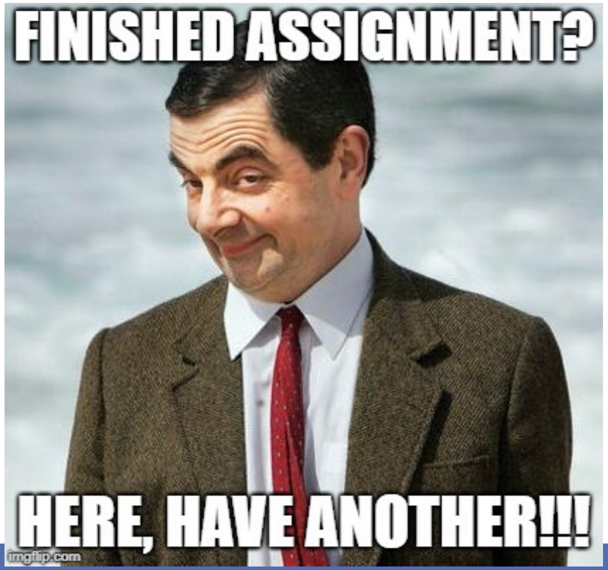
These slides are prepared by M. A. Cheema and are based on the material developed by Arun Konagurthu and Lloyd Allison.

Recommended Reading

- Unit Notes (Chapter 5)
- Weiss “Data Structures and Algorithm Analysis" (Pages 462-466.)
- Edit Distance Problem: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Dynamic/Edit/
- Dynamic Programming: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Dynamic/
- Practice: http://www.geeksforgeeks.org/tag/dynamic-programming/

Things to remember/note

- Assignment 2 released
  - Due 7-April 2019 23:55:00
  - Start early, finish early, and live happily ~~ever after~~ until next one is released



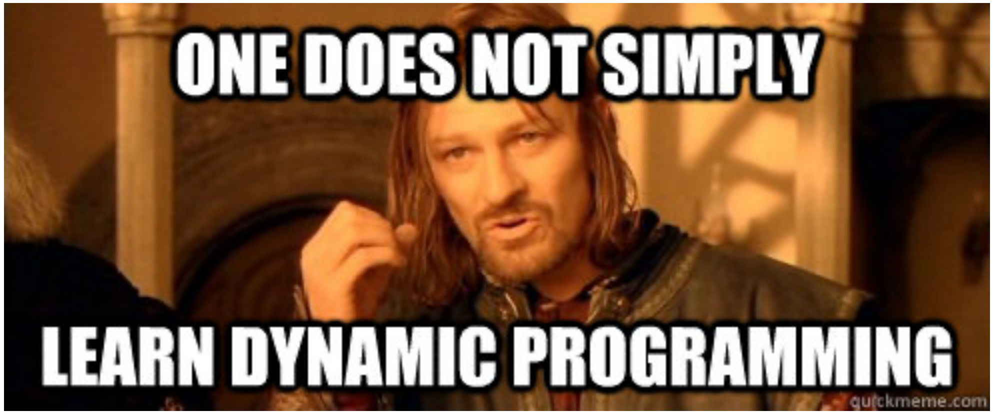
Outline

1. Introduction to Dynamic Programming
2. Coins Change
3. Unbounded Knapsack
4. 0/1 Knapsack
5. Edit Distance
6. Constructing Optimal Solution

Dynamic Programming Paradigm

- A powerful optimization technique in computer science
- Applicable to a wide-variety of problems that exhibit certain properties.

- Practice is the key to be good at dynamic programming



Core Idea

- Divide a complicated problem by breaking it down into simpler subproblems in a recursive manner and solve these.
- Question: But how does this differ from `Divide and Conquer' approach?
- Subproblems are overlapping (in contrast to independent subproblems in Divide and Conquer)
  - Identify the overlapping subproblems
  - Solve the smaller subproblems and **memoize** the solutions
  - use the memoized solutions of subproblems to gradually build solution for the original problem

N-th Fibonacci Number

```
fib(N)
    if N == 0 or N == 1
        return N
    else
        return fib(N - 1) + fib(N - 2)
```

Time Complexity

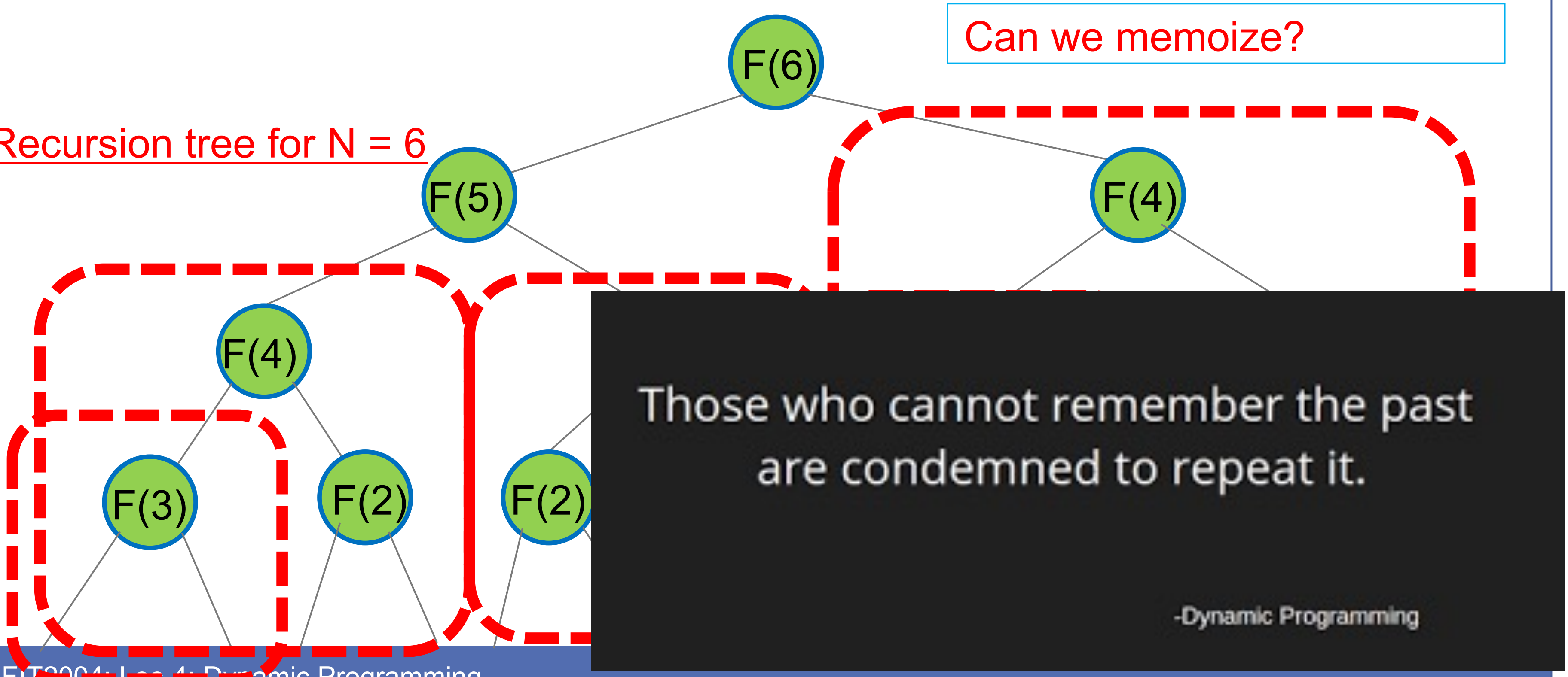
$T(1) = b$ // b and c are constants

$T(N) = T(N-1) + T(N-2) + c$

$= O(2^N)$

Can we memoize?

Recursion tree for N = 6



Fibonacci with Memoization: Version 1

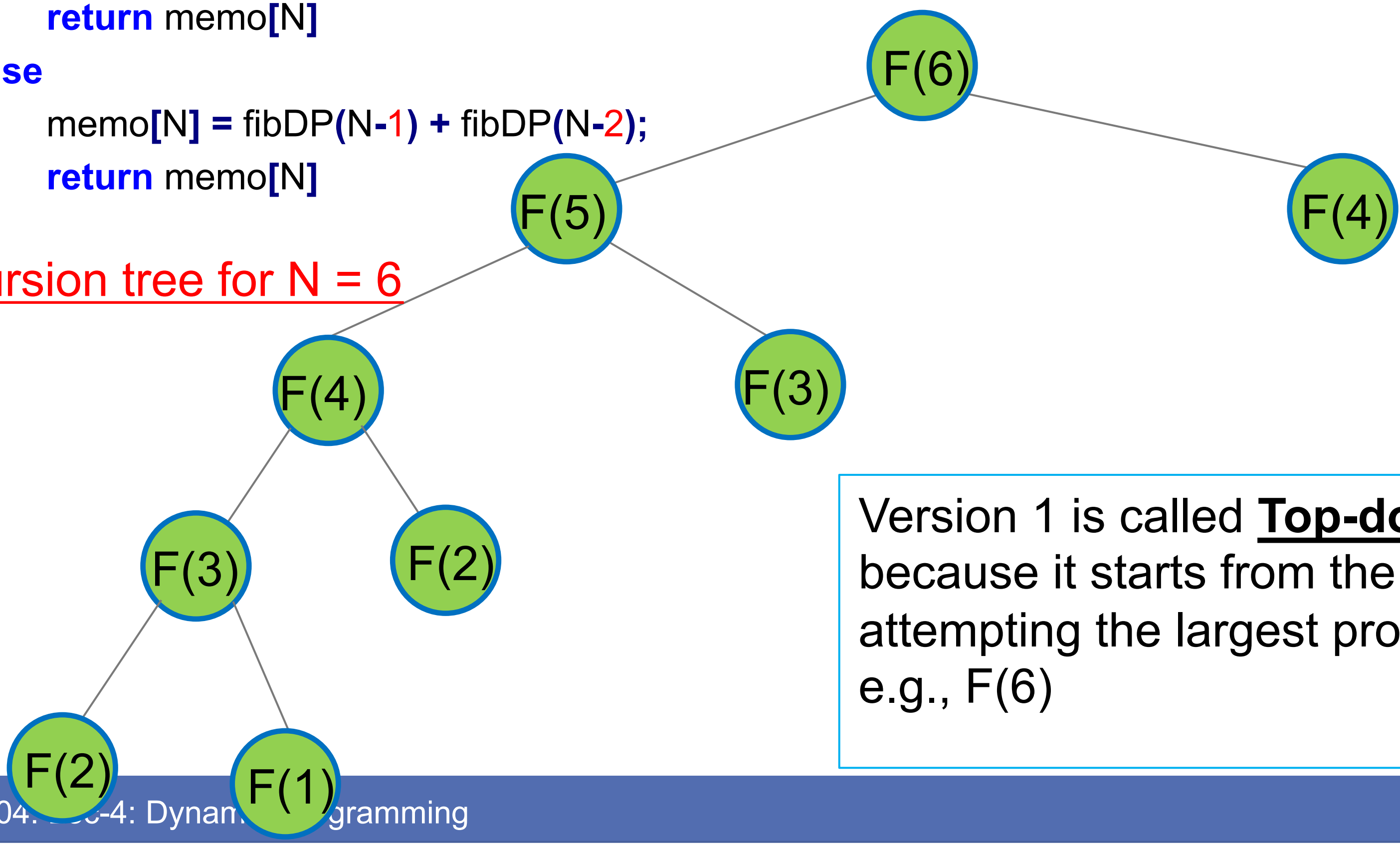
```
memo[0] = 0  // 0th Fibonacci number
memo[1] = 1  // 1st Fibonacci number
for i=2 to i=N:
    memo[i] = -1
fibDP(N)
    if memo[N] != -1
        return memo[N]
    else
        memo[N] = fibDP(N-1) + fibDP(N-2);
        return memo[N]
```

Time Complexity

calls fibDP() roughly 2*N times

So the complexity is O(N)

Recursion tree for N = 6



Version 1 is called **Top-down** because it starts from the top – attempting the largest problem first, e.g., F(6)

Fibonacci with Memoization: Version 2

```
memo[0] = 0  // 0th Fibonacci number
memo[1] = 1  // 1st Fibonacci number
for i=2 to i=N:
        memo[i] = memo[i-1] + memo[i-2]
```

Time Complexity

O(N)

Version 2 is called **<u>Bottom-up</u>** because it starts from the bottom – solving the smallest problem first, e.g., F(0), F(1), and so on

Dynamic Programming Strategy

1. Assume you already know the solutions of all sub-problems and have memoized these solutions
   - E.g., Assume you know Fib(i) for every i < n
2. Observe how you can solve the original problem using memoized solutions
   - E.g., Fib(n) = Fib(n-1) + Fib(n-2)
3. Solve the original problem by building upon solutions to the sub-problems
   - E.g., Fib(0), Fib(1), Fib(2), …, Fib(n)

Outline

1. Introduction to Dynamic Programming
2. Coins Change
3. Unbounded Knapsack
4. 0/1 Knapsack
5. Edit Distance
6. Constructing Optimal Solution

Coins Change Problem

Problem: A country uses N coins with denominations {a1, a2, …, aN}. Given a value V, find the minimum number of coins that add up to V.

Example: Suppose the coins are {1, 5, 10, 50} and the value V is 110. The minimum number of coins required to make 110 is 3 (two 50 coins, and one 10 coin).

Greedy solution does not always work.
E.g., Coins = {1, 5, 6, 9}

The minimum number of coins to make 12 is 2 (i.e., two 6 coins).

What is the minimum number of coins to make 13?

DP Solution for Coins Change

You need to make the value V=12.

Assume we know the optimal solutions for every V < 12 and results are stored in Memo[ ]

If I tell you that you must use at least one coin of value 9, what is the minimum number of coins to make V=12?

If optimal solution contains a coin with value x (e.g., coin 9 in the example):

MinCoins(V) = 1 + Memo[V- x]

// Any of the N coins can be in the optimal solution for V. Pick the one that returns minimum value of MinCoins(V)

```
MinCoins = infinity
For i=1 to N
    if Coins[ i ] <= V // Avoid accessing Memo at a negative index
        c = 1 + Memo[ V - Coins[ i ] ]
        if c < MinCoins
            MinCoins = c
Memo[V] = minCoins
```

1+ memo[3] = 1 + 2 = 3

i

Coins $\boxed{9}$

4

Memo	1	2	3	4	1	1	2	3	1	2	2	2
	1	2	3	4	5	6	7	8	9	10	11	12

Bottom-up Solution

// Construct Memo[ ] starting from 1 until V in a way similar to previous slide .

```
Initialize Memo[ ] to contain infinity for all indices
Memo[0] = 0
for v = 1 to V
    minCoins = Infinity
    for i=1 to N
        if Coins[ i ] <= v
            c = 1 + Memo[v - Coins[ i ] ]
            if c < minCoins
                minCoins = c
    Memo[v] = minCoins
```

Time Complexity:

O(NV)

Space Complexity:

O(V + N)

E.g., Fill Memo[13]

Coins

9	5	6	1

Memo

1	2	3	4	1	1	2	3	1	2	2	2	
1	2	3	4	5	6	7	8	9	10	11	12	13

Top-down Solution

```
Initialize Memo[ ] to contain -1 for all indices # -1 indicates the solution
for this index has not been computed yet
Memo[0] = 0
Function CoinChange(value)
   if Memo[value] != -1:
     return Memo[value]
  else:
     minCoins = Infinity
     for i=1 to N
        if Coins[ i ] <= value
          c = 1 + CoinChange(value - Coins[ i ])
          if c < minCoins
             minCoins = c
     Memo[value] = minCoins
     return Memo[value]
```

Bottom up solution:

1 + Memo[value – Coins[i]]

Outline

1. Introduction to Dynamic Programming
2. Coins Change
3. Unbounded Knapsack
4. 0/1 Knapsack
5. Edit Distance
6. Constructing Optimal Solution

Unbounded Knapsack Problem

Problem: Given a capacity C and a set of items with their weights and values, you need to pick items such that their total weight is at most C and their total value is maximized. What is the maximum value you can take? In unbounded knapsack, you can pick an item as many times as you want.

Example: What is the maximum value for the example given below given capacity is 12 kg?

Answer: \$780 (take two Bs and two Ds)
Greedy solution does not always work.

18th most popular algorithmic problem!!!!

Item	A	B	C	D
Weight	9kg	5kg	6kg	1kg
Value	\$550	\$350	\$180	\$40

DP Solution for Unbounded Knapsack

Assume we know the optimal solutions for every C < 12kg and results are stored in Memo[ ]

If I tell you that you must use at least one item #1 (weight 9kg), what is the maximum value if C=12?

// Any of the N items can be in the optimal solution for C. Pick the one that gives maximum value

If optimal solution contains an item #i (e.g., item 1 with weight 9 and value $550)

MaxValue = Value[i] + Memo[C- weight[i]]

```
MaxValue = 0
For i=1 to N
    if weight[ i ] <= C // Avoid accessing Memo at a negative index
        thisValue = value[i] + Memo[ C - weight[ i ] ]
        if thisValue > MaxValue
            MaxValue = thisValue
Memo[C] = MaxValue
```

i

Item	1
Weight	9kg
Value	$550

670 $\boxed{780}$ 570 $\boxed{780}$

550 + Memo[3] = 670

350+ memo[7] = 780

180+ memo[6] = 570

40+ memo[11] = 780

Or

Memo

Memo	40	80	120	160	350	390	430	470	550	700	740	780
	1	2	3	4	5	6	7	8	9	10	11	12

Bottom-up Solution

```
// Construct Memo[ ] starting from 1 until C in a way similar to previous slide .
Initialize Memo[ ] to contain 0 for all indices
for c = 1 to C
    maxValue = 0
    for i=1 to N
        if Weight[ i ] <= c
            thisValue = Value[i]  + Memo[c - Weight[ i ] ]
            if thisValue > maxValue
                maxValue = thisValue
    Memo[c] = maxValue
```

Time Complexity:

O(NC)

Space Complexity:

O(C + N)

E.g., Fill Memo[13]

Item	1	2	3	4
Weight	9kg	5kg	6kg	1kg
Value	$550	$350	$180	$40

Memo

40	80	120	160	350	390	430	470	550	700	740	780	
1	2	3	4	5	6	7	8	9	10	11	12	13

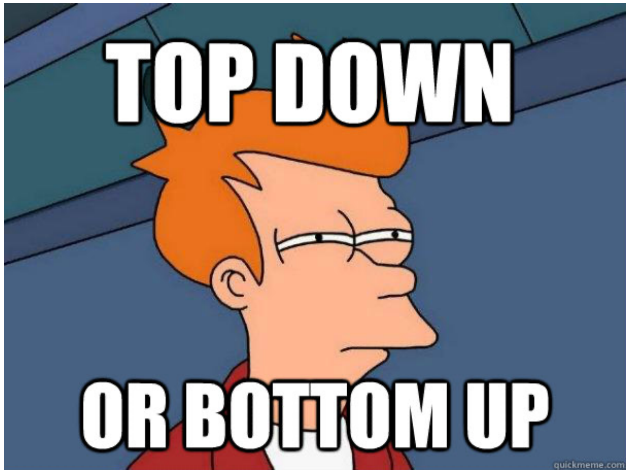
Top-down Solution

Initialize Memo[ ] to contain -1 for all indices // -1 indicates solution for this index has not yet been computed

```
Memo[0] = 0
function knapsack(Capacity)
   if Memo[ Capacity ] != -1:
     return Memo[Capacity]
   else:
     maxValue = 0
     for i=1 to N
       if Weight[ i ] <= Capacity
          thisValue = Value[i]  + knapsack(Capacity - Weight[ i ] )
          if thisValue > maxValue
             maxValue = thisValue
      Memo[Capacity] = maxValue
      return Memo[Capacity]
```

Bottom up solution:

Values[i] + Memo[Capacity – Weights[i]]



- Top-down **may** save some computations (E.g., some smaller subproblems may not needed to be solved)
- Space saving trick may be applied for bottom-up to reduce space complexity

Outline

1. Introduction to Dynamic Programming
2. Coins Change
3. Unbounded Knapsack
4. 0/1 Knapsack
5. Edit Distance
6. Constructing Optimal Solution

0/1 Knapsack Problem

Same as unbounded knapsack except that each item can only be picked at most once.

Example: What is the maximum value for the example given below given capacity is 11 kg?

Answer: $590 (B and D)

Greedy solution may not always work.

Item	A	B	C	D
Weight	6kg	1kg	5kg	9kg
Value	$230	$40	$350	$550

0/1 Knapsack Problem

Problem: What is the solution for 0/1 knapsack for items {A,B,C,D} where capacity = 11.

Assume that we have computed solutions for every capacity<=11 considering the items {A,B,C} (see table below).

What is the solution for capacity=11 and set {A,B,C,D} ?

- Case 1: the knapsack must **NOT** contain D
  - Solution for 0/1 knapsack with set {A,B,C} and capacity 11. 580
- Case 2: the knapsack **must** contain D
  - The value of item D + solution for 0/1 knapsack with set {A,B,C} and capacity 11-9=2

550+40 = 590

- **Solution = max(Case1, Case2)**

Item	A	B	C	D
Weight	6kg	1kg	5kg	9kg
Value	$230	$40	$350	$550

	1	2	3	4	5	6	7	8	9	10	11
{A,B,C}	40	40	40	40	350	390	390	390	390	390	580

0/1 Knapsack Problem

Assume we know the optimal solutions for every subproblem and results are stored in Memo[ ][ ].
Memo[i][c] contains the solution of knapsack for **Set[1 … i]** and capacity **c**

```
for i=1 to N:
  for c in 1 to C:
    excludedValue = Memo[i-1][c]
    includedValue = 0
    if weight[i] <= c:
      includedValue = values[i] + Memo[i-1][c - weight[i]]
    Memo[i][c] = max(excludedValue,includedValue)
```

Item	A	B	C	D
Weight	6kg	1k[illegible]	[illegible]	[illegible]kg
Value	$230	$4[illegible]	$350	$550

Time Complexity:
O(NC)
Space Complexity:
O(NC)

550+40 = 590

c

		1	2	3	4	5	6	7	8	9	10	11	12
0	Φ	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	230	230	230	230	230	230	230
2	B	40	40	40	40	40	230	270	270	270	270	270	270
3	C	40	40	40	40	350	390	390	390	390	390	580	620
i 4	D	40	40	40	40	350	390	390	390	550	590	590	620

0/1 Knapsack Problem

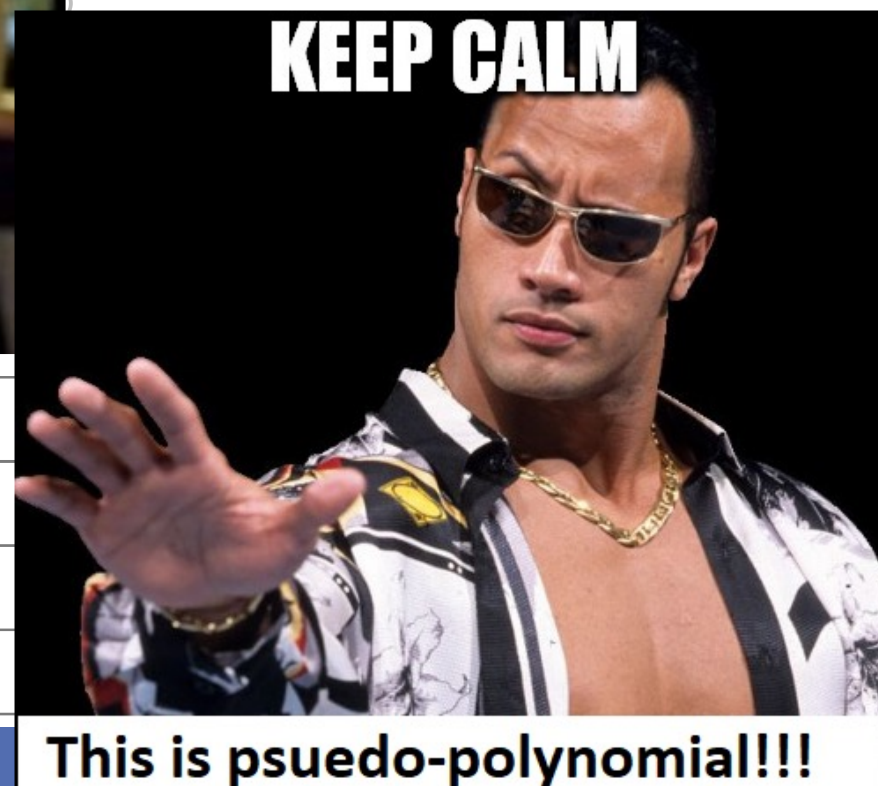
Assume we know the optimal solutions for every subproblem and results are stored in Memo[ ][ ].

capacity **c**

Time Complexity:
O(NC)
Space Complexity:
O(NC)

weight[i]]

	1	A	0	0	0	0	0
	2	B	40	40	40	40	40
	3	C	40	40	40	40	350
i	4	D	40	40	40	40	350



Reducing Space Complexity

```
for i=1 to N:
  for c in 1 to C:
    excludedValue = Memo[i-1][c]
    includedValue = 0
    if weight[i] <= c:
      includedValue = values[i] + Memo[i-1][c - weight[i]]
    Memo[i][c] = max(excludedValue,includedValue)
```

Space Complexity:
O(N+C)

Observe that, at each iteration of the outer for loop, algorithm only accesses row i or row (i-1), i.e., Memo[i] or Memo[i-1]. Therefore, we only need to maintain two rows (i-th and i-1-th) instead of all N rows.

Note: Space saving not possible for top-down dynamic programming

		1	2	3	4	5	6	7	8	9	10	11	12
0	Φ	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	230	230	230	230	230	230	230
2	B	40	40	40	40	40	230	270	270	270	270	270	270
3	C	40	40	40	40	350	390	390	390	390	390	580	620
4	D	40	40	40	40	350	390	390	390	550	590	590	620

Outline

1. Introduction to Dynamic Programming
2. Coins Change
3. Unbounded Knapsack
4. 0/1 Knapsack
5. Edit Distance
6. Constructing Optimal Solution

Edit Distance

- The words computer and commuter are very similar, and a change of just one letter, p → m, will change the first word into the second.
- The word sport can be changed into sort by the deletion of p, or equivalently, sort can be changed into sport by the insertion of p'.
- Notion of editing provides a simple and handy formalisation to compare two strings.
- The goal is to convert the first string (i.e., sequence) into the second through a series of edit operations
- The permitted edit operations are:
    1. insertion of a symbol into a sequence.
    2. deletion of a symbol from a sequence.
    3. substitution or replacement of one symbol with another in a sequence.

Edit Distance

Edit distance between two sequences

- Edit distance is the minimum number of edit operations required to convert one sequence into another

For example:

- Edit distance between computer and commuter is 1
- Edit distance between sport and sort is 1.
- Edit distance between shine and sings is ?
- Edit distance between dnasgivethis and dentsgnawstrims is ?

Some Applications of Edit Distance

- Natural Language Processing
  - Auto-correction
  - Query suggestions
- BioInformatics
  - DNA/Protein sequence alignment

Computing Edit Distance

We want to convert s1 to s2 containing n and m letters, respectively.

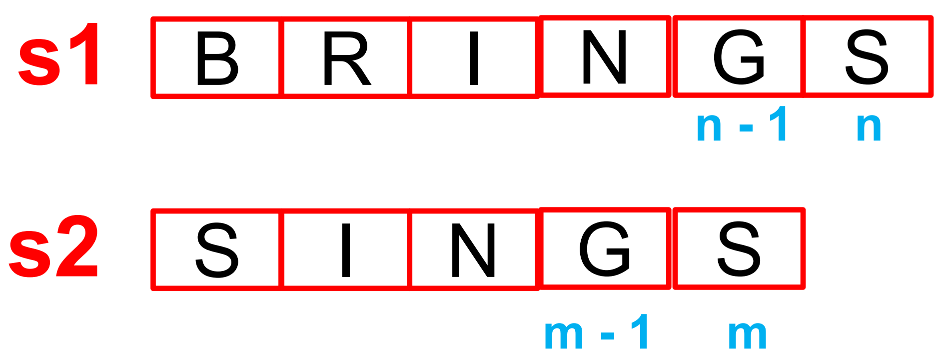
Assume we have computed and memoized the optimal solution for all sub-problems (e.g., convert s1[1…n-1] to s1[1…m-1])

Observations:

// n is length of s1 and m is length of s2

If s1[n] == s2[m]

cost = dist(s1[1…n-1],s2[1… m-1])



Computing Edit Distance

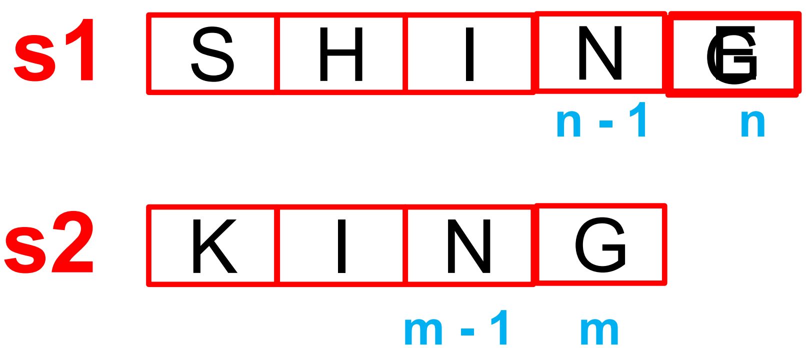
We want to convert s1 to s2. Suppose we have computed and memoized the optimal solution for all subproblems

Observations:

// n is length of s1 and m is length of s2

if optimal solution is substituting s1[n] with s2[m]

cost = 1 + dist(s1[1…n-1],s2[1…m-1])



Computing Edit Distance

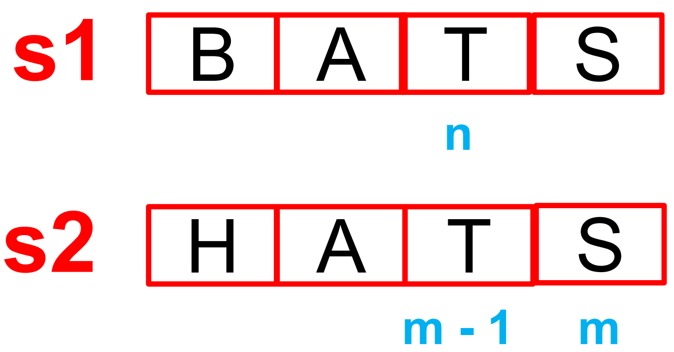
We want to convert s1 to s2. Suppose we have computed and memoized the optimal solution for all subproblems

Observations:

// n is length of s1 and m is length of s2

if optimal solution is adding s2[m] in s1 after s1[n]

cost = 1 + dist(s1[1…n],s2[1…m-1])



Computing Edit Distance

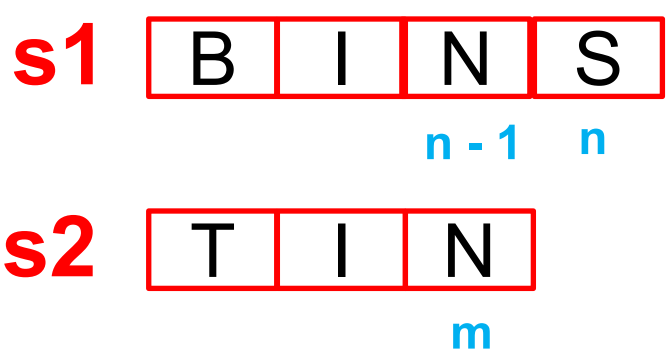
We want to convert s1 to s2. Suppose we have computed and memoized the optimal solution for all subproblems

Observations:

// n is length of s1 and m is length of s2

if optimal solution is removing s1[n]

cost = 1 + dist(s1[1…n-1],s2[1…m])



Computing Edit Distance

We want to convert s1 to s2. Suppose we have computed and memoized the optimal solution for all subproblems

Summary of all observations:

// n is length of s1 and m is length of s2

If s1[n] == s2[m]
    cost = dist(s1[1…n-1],s2[1… m-1])
Else
    if substituting s1[n] with s2[m]
        cost = 1 + dist(s1[1…n-1],s2[1…m-1])
    if adding s2[m] in s1 after s1[n]
        cost = 1 + dist(s1[1…n],s2[1…m-1])
    if removing s1[n]
        cost = 1 + dist(s1[1…n-1],s2[1…m])

Just take the minimum cost.
cost = 1 +
Min (dist(s1[1…n-1],s2[1…m-1]),
    dist(s1[1…n],s2[1…m-1])
    dist(s1[1…n-1],s2[1…m])
)

s1	S	I	N	G	S
					n
s2	S	H	I	N	E
					m

Computing Edit Distance

// Fill Memo[ ] [ ] using the observations

If s1[n] == s2[m]

cost = dist(s1[1…n-1],s2[1… m-1])

Else

cost = 1 + Min (dist(s1[1…n-1],s2[1…m-1]),
dist(s1[1…n],s2[1…m-1]),
dist(s1[1…n-1],s2[1…m]))

//After filling the Memo, return Memo[n][m] (the value of last cell which is the edit distance)

Time Complexity:

O(nm)

Space Complexity:

O(nm)

			1	2	…		m
		Φ	**S**	**H**	**I**	**N**	**E**
	Φ	0	1	2	3	4	5
1	**S**	1	0	1	2	3	4
2	**I**	2	1	1	1	2	3
…	**N**	3	2	2	2	1	2
	G	4	3	3	3	2	2
n	**S**	5	4	4	4	3	3

Reducing Space Complexity

- Similar to 0/1 Knapsack, we only need to access two rows at any time.
- This reduces space complexity to O(n + m)

Note: Space saving is not possible for top-down dynamic programming

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Outline

1. Introduction to Dynamic Programming
2. Coins Change
3. Unbounded Knapsack
4. 0/1 Knapsack
5. Edit Distance
6. Constructing Optimal Solution

Constructing optimal solutions

- The algorithms we have seen determine optimal values, e.g.,
  - Minimum number of coins
  - Maximum value of knapsack
  - Edit distance
- How do we construct optimal solution that gives the optimal value, e.g.,
  - The coins to give the change
  - The items to put in knapsack
  - Converting one string to the other
- There may be multiple optimal solutions and our goal is to return just one solution!
- Two strategies can be used.
  1. Create an additional array recording decision at each step
  2. Backtracking

Finding coins: Using Decision array

```
Initialize Memo[ ] to contain infinity for all indices
Memo[0] = 0
Initialize Decisions[ ] to contain zeroes
for v = 1 to V
    minCoins = Infinity
    for i=1 to N
        if Coins[ i ] <= v
            c = 1 + Memo[v - Coins[ i ] ]
            if c < minCoins
                minCoins = c
                coin = Coins[i]
    Memo[v] = minCoins
    Decisions[v] = coin
```

Do not store ALL coins needed to make the change at each step. Just store the chosen coin at that step.

	i	i	i	i
Coins	9	5	6	1
	4	3	2	3

Memo	1	2	3	4	1	1	2	3	1	2	2	2
	1	2	3	4	5	6	7	8	9	10	11	12

Decisions	1	1	1	1	5	6	6	6	9	9	5	6
	1	2	3	4	5	6	7	8	9	10	11	12

Finding coins: Using Decision array

```
Solution = []
while V!=0:
    Solution.append(Decisions[V])
    V = V - Decision[V]
```

E.g., If V= 12

- Look at Decisions[12], append 6 → Solution = [6]
- Remaining change, V = 12-6=6
- Look at Decisions[6], append 6 → Solution = [6,6].
- Stop because V = 6-6 = 0

If V = 11

- Look at Decisions[11], append 5 → Solution = [5]
- Remaining change, V = 11-5= 6
- Look at Decisions[6], append 6 → Solution = [5,6]
- Stop because V = 6-6 = 0

Coins	9	5	6	1

Decisions	1	1	1	1	5	6	6	6	9	9	5	6
	1	2	3	4	5	6	7	8	9	10	11	12

Finding Coins: Backtracking

Execution to be shown in class

```
// Find coins for optimal solution for V = 13 without using Decision[]
Solution = []
while V > 0:
        for coin_value in Coins:
                if coin_value <= V:
                        if Memo[V] == 1 + Memo[V - coin_value]
                                chosen_coin = coin_value
                                break
        Solution.append(chosen_coin)
        V = V-chosen_coin
```

Solution:

V:

chosen_coin:

Coins

9	5	6	1

Memo

Memo	1	2	3	4	1	1	2	3	1	2	2	2	3
	1	2	3	4	5	6	7	8	9	10	11	12	13

Finding string conversion: Backtracking

Backtracking: Use the Matrix to determine where the values are coming from (if multiple, pick any of those).

Recall: Diagonal means substitution if letters are not same

Upward arrow means removing the letter s1[i]

Left arrow means adding the letter s2[i] in s1

- Substitute S with E
- Delete G
- Add H after S

```
If s1[n] == s2[m]
    cost = dist(s1[1…n-1],s2[1… m-1])
Else
  if substituting s1[n] with s2[m]
    cost = 1 + dist(s1[1…n-1],s2[1…m-1])
  if adding s2[m] in s1 after s1[n]
    cost = 1 + dist(s1[1…n],s2[1…m-1])
  if removing s1[n]
    cost = 1 + dist(s1[1…n-1],s2[1…m])
```

s1 | S | H | N | G | E |

		Φ	S	H	I	N	E
1	Φ	0	1	2	3	4	5
2	S	1	0	1	2	3	4
…	I	2	1	1	1	2	3
	N	3	2	2	2	1	2
	G	4	3	3	3	2	2
n	S	5	4	4	4	3	3
			1	2	…		m

Backtracking Vs Decision array?

- Space usage
  - Backtracking requires less space as it does not require creating an additional array
  - However, space **complexity** is the same
- Efficiency
  - Backtracking requires to **determine** what decision was made which costs additional computation
  - However, time **complexity** is the same
- Note the space saving tricks discussed for 0/1 knapsack and edit distance can only be used when solution is not to be constructed
  - e.g., all rows are needed for backtracking, and all rows must be stored for 2D-decision array

Concluding Remarks

Dynamic Programming Strategy

- Assume you already know the optimal solutions for all subproblems and have memoized these solutions
- Observe how you can solve the original problem using this memoization
- Iteratively solve the sub-problems and memoize

Things to do (this list is not exhaustive)

- Practice, practice, practice
  - http://www.geeksforgeeks.org/tag/dynamic-programming/
  - https://www.topcoder.com/community/data-science/data-science-tutorials/dynamic-programming-from-novice-to-advanced/
  - http://weaklearner.com/problems/search/dp
- Revise hash tables and binary search tree

Coming Up Next

- Hashing, Binary Search Tree, AVL Tree