Faculty of Information Technology, Monash University

FIT2004, S2/2016

Week 4: Dynamic Programming

Lecturer: Muhammad Aamir Cheema

ACKNOWLEDGMENTS

The slides are based on the material developed by Arun Konagurthu and Lloyd Allison.

Overview

- Dynamic Programming Paradigm
  - Fibonacci numbers
  - Coins change
  - Subset sum
  - Edit distance
- Hirschberg's Algorithm (A beautiful combination of Dynamic Programming and Divide and Conquer)

Recommended Reading

- 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/
- Hirschberg's algorithm: http://www.csse.monash.edu.au/courseware/cse2304/2006/08hirsch.shtml
- Practice: http://www.geeksforgeeks.org/tag/dynamic-programming/

Dynamic Programming Paradigm

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

- The term Dynamic Programming was coined by Richard Bellman in 1940s to describe the process of solving problems where one needs to find the best decisions one after another

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

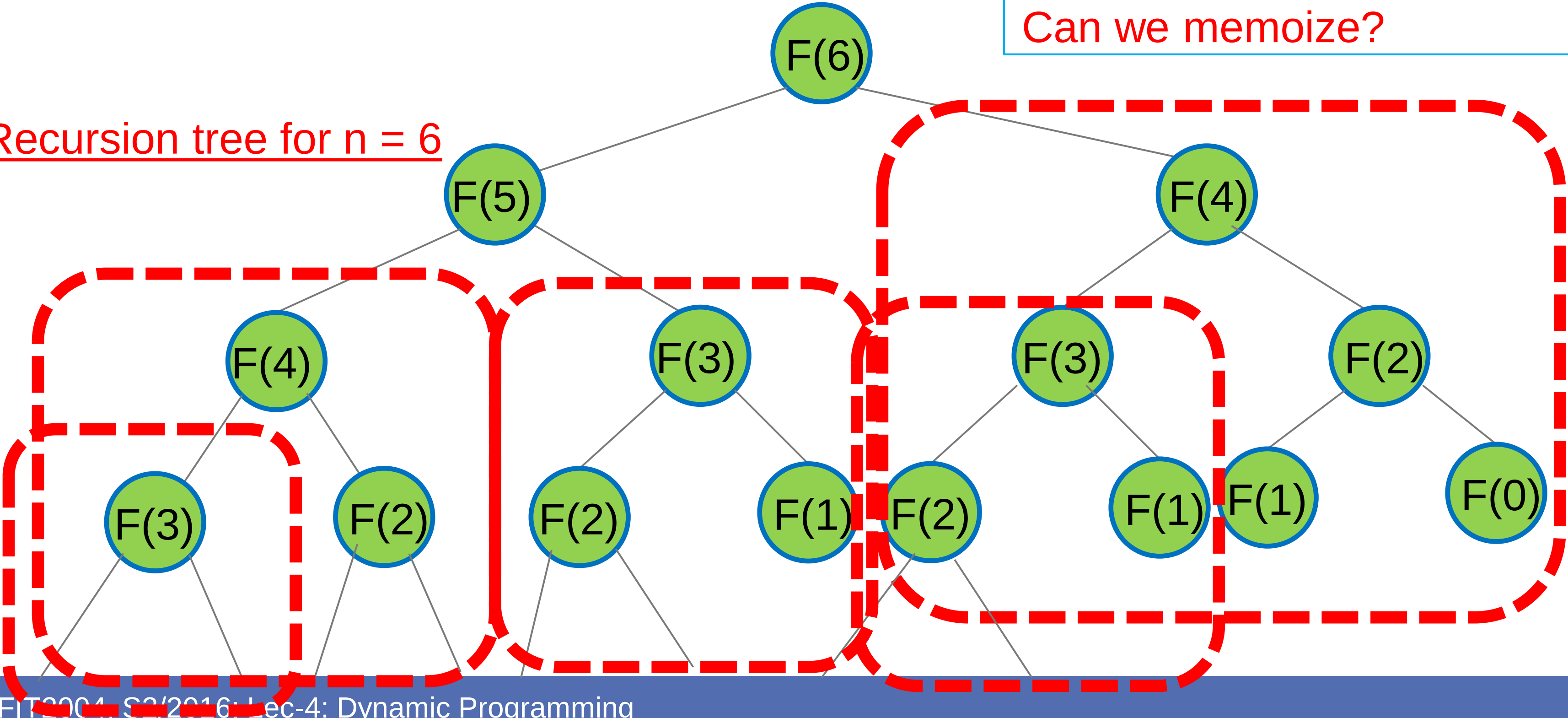
$T(1) = b$ // b and c are constants

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

$= O(2^N)$

Can we memoize?

Recursion tree for n = 6



N-th Fibonacci Number with Memoization

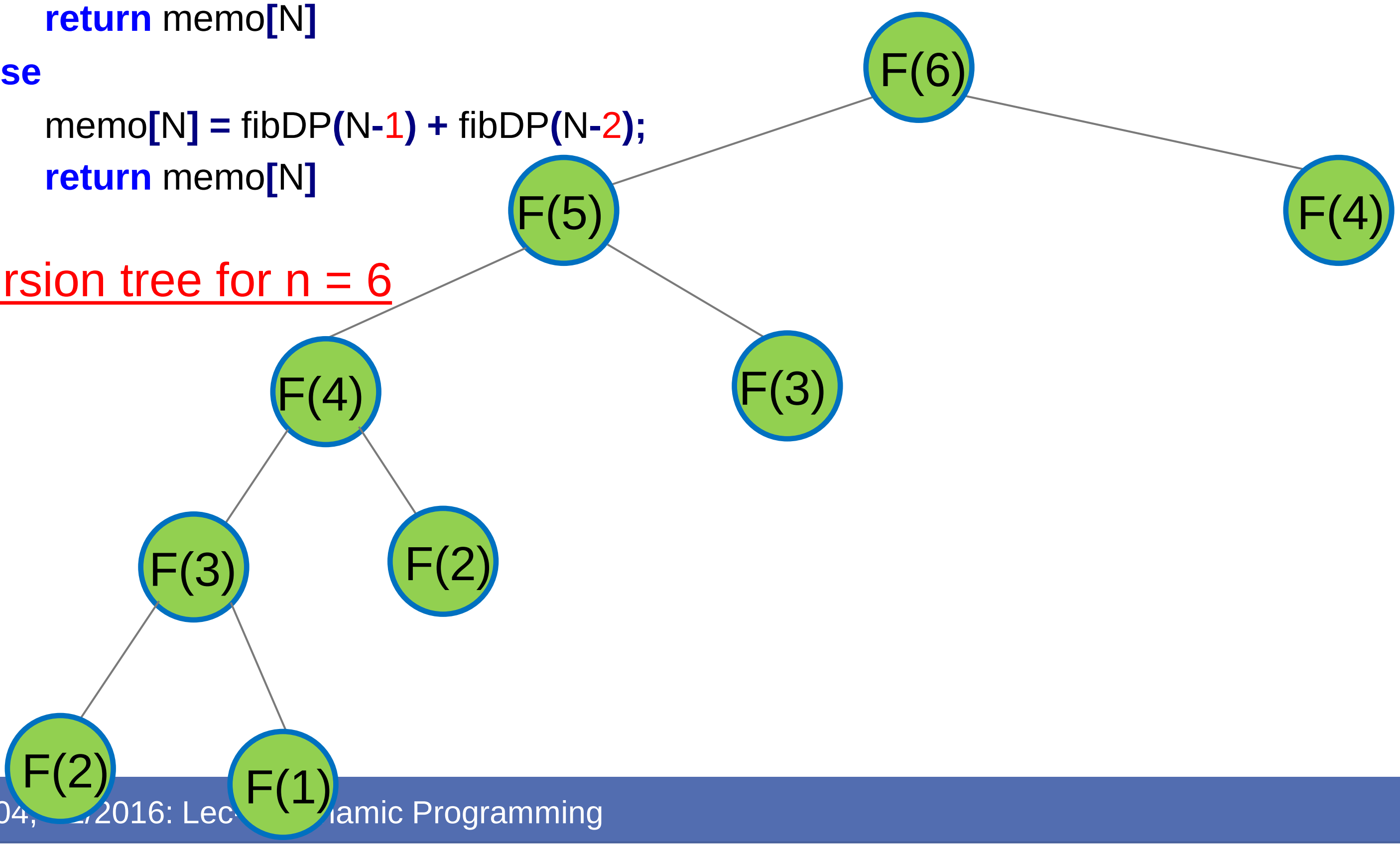
```
memo[0] = 0  // 0th Fibonacci number
memo[1] = 1  // 1st Fibonacci number
for i=2 to i=N:
    memo[i] = -1
fibDP(n)
    if memo[N] != -1
        return memo[N]
    else
        memo[N] = fibDP(N-1) + fibDP(N-2);
        return memo[N]
```

Time Complexity

calls fibDP() roughly 2*N times

So the complexity is O(N)

Recursion tree for n = 6



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 if you have memoized the solutions of subproblems
   - E.g., Fib(n) = Fib(n-1) + Fib(n-2)
3. Use the observations and iteratively solve the sub-problems and memoize the solutions
   - E.g., memoize Fib(0), Fib(1), Fib(2), …, Fib(n)

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

Coins = {9, 5, 6, 1}. You need to make the value V=12.

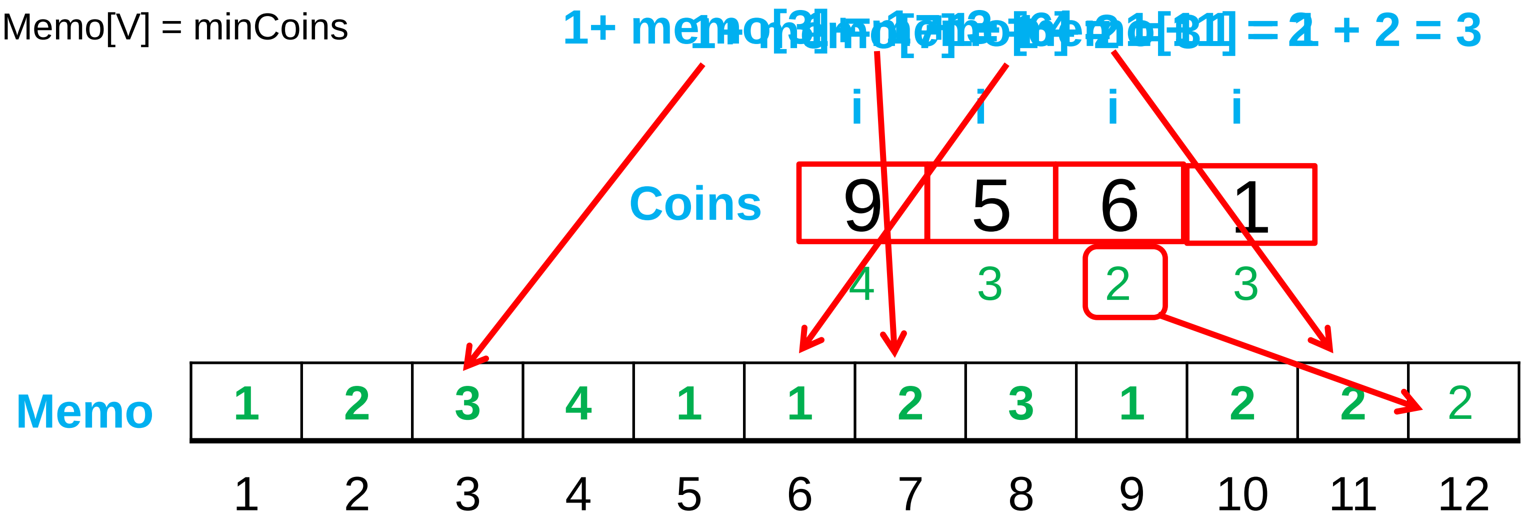
Assume we know the optimal solutions for every V < 12 and results are stored in Memo[ ]

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
```



DP Solution for Coins Change

```
// 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

Subset Sum Problem

Given a set of numbers N = {a1, a2, …, an} and a value V. Is their a subset of N such that the sum of elements is V.

Note: Unlike Coins Change problem, a number can only be used once to make the value V.

Example: Suppose N = {1, 5, 6, 9} and the value V is 13. The answer is FALSE because no subset of N adds to 13. For V=15, the answer is TRUE because 9 + 6 = 15.

What is the answer for V = 12?

Subset Sum Problem

- Suppose N = {1, 5, 6, 9} and V = 16. Assume that I tell you that the subset must contain the value 9. What is the answer of subset problem? E.g., is there a subset that includes 9 and adds up to 16?
  - If the subset must contain 9
    - The answer is True if and only if {1, 5, 6} has a subset adding up to 16-9 = 7

- Suppose N = {1, 5, 6, 9} and V = 11. Assume that I tell you that the subset must **NOT** contain the value 9. What is the answer of subset problem? E.g., is there a subset that excludes 9 and adds up to 11?
  - If the subset must not contain 9
    - The answer is True if {1, 5, 6} has a subset adding up to 11

```
If the Subset contains a number x
      Answer is True if {Set \ x} has a subset adding up to V - x
Else
      Answer is True if {Set \ x} has a subset adding up to V
```

Subset Sum Problem

Set = {1, 3, 6}. We need to check for V = 7 (e.g., is there a subset with sum = 7)

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

Memo[v][i] contains True **if and only if** there exists a subset of Set[1 … i] that adds to v

```
For v in 1 to V
    For i=1 to n
        x = Set[ i ]
        if Memo[v - x ] [ i -1] == True or Memo [ v ] [ i – 1] == True
            Memo [v][i] = True
        Else
            Memo [v][i] = False
//Fill column for 9
```

Time Complexity:

O(nV)

Space Complexity:

O(nV)

If the Subset contains a number umber 6)

Answer is True if {Set \ x} has a sub set adding up to V - x

Else

Answer is True if {Set \ x} has a subset adding up to V

	0	1	2	3	4	5	6	7	8	9
Φ	T	F	F	F	F	F	F	F	F	
1	T	T	F	F	F	F	F	F	F	
3	T	T	F	T	T	F	F	F	F	
6	T	T	F	T	T	F	T	T	F	

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
  - Autocomplete
  - Query suggestions
- BioInformatics
  - DNA/Protein sequence alignment

Computing Edit Distance

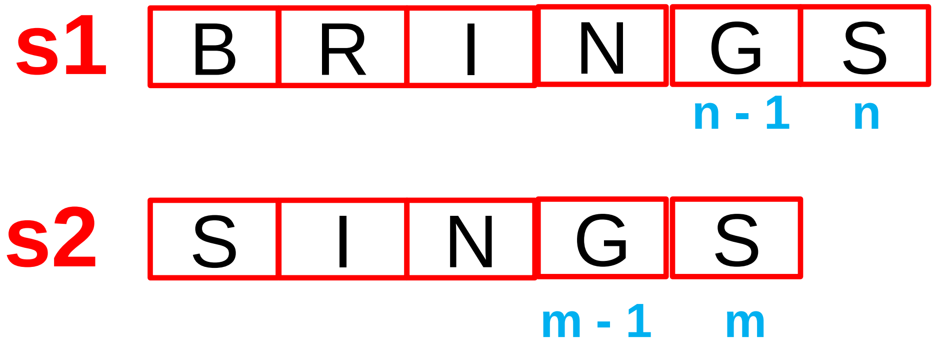
We want to convert s1 to s2 containing n and m letters, respectively.

Assume we have computed and memoized the optimal solution for all subproblems (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

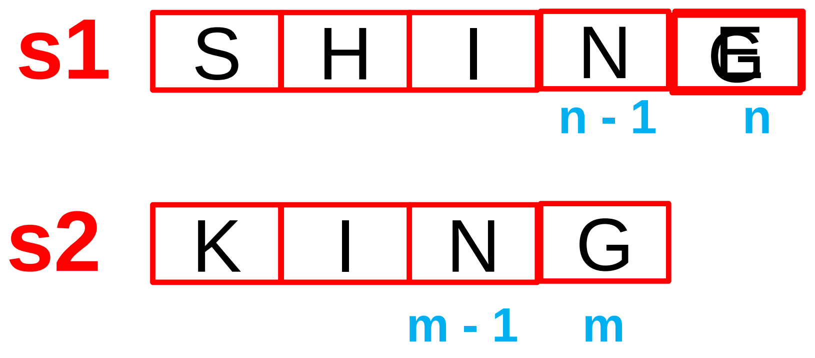
We want to convert s1 to s2. Suppose we have computed and memoized the optimal solution for all subproblems

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

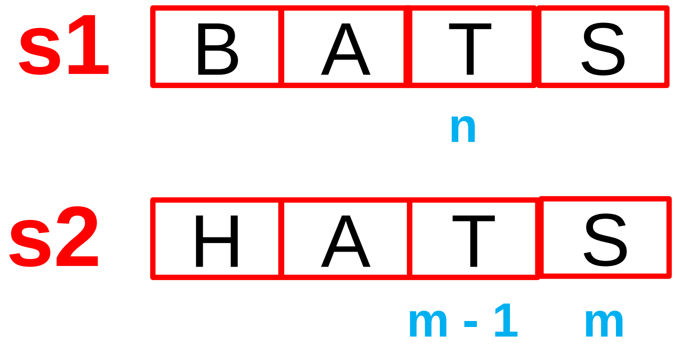
We want to convert s1 to s2. Suppose we have computed and memoized the optimal solution for all subproblems

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])

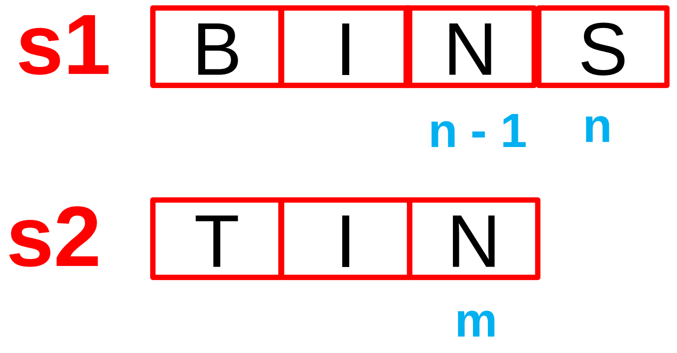


Computing Edit Distance

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

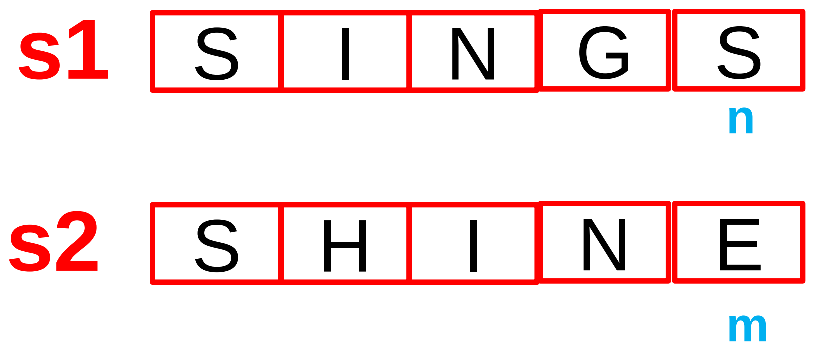
We want to convert s1 to s2. Suppose we have computed and memoized the optimal solution for all subproblems

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[n-1],s2[m-1]),
     dist(s1[1…n],s2[1…m-1])
     dist(s1[1…n-1],s2[1…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[n-1],s2[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

Converting s1 to s2

The algorithm determines the edit distance between two strings. What if we want to recover the operations required to convert s1 to s2?

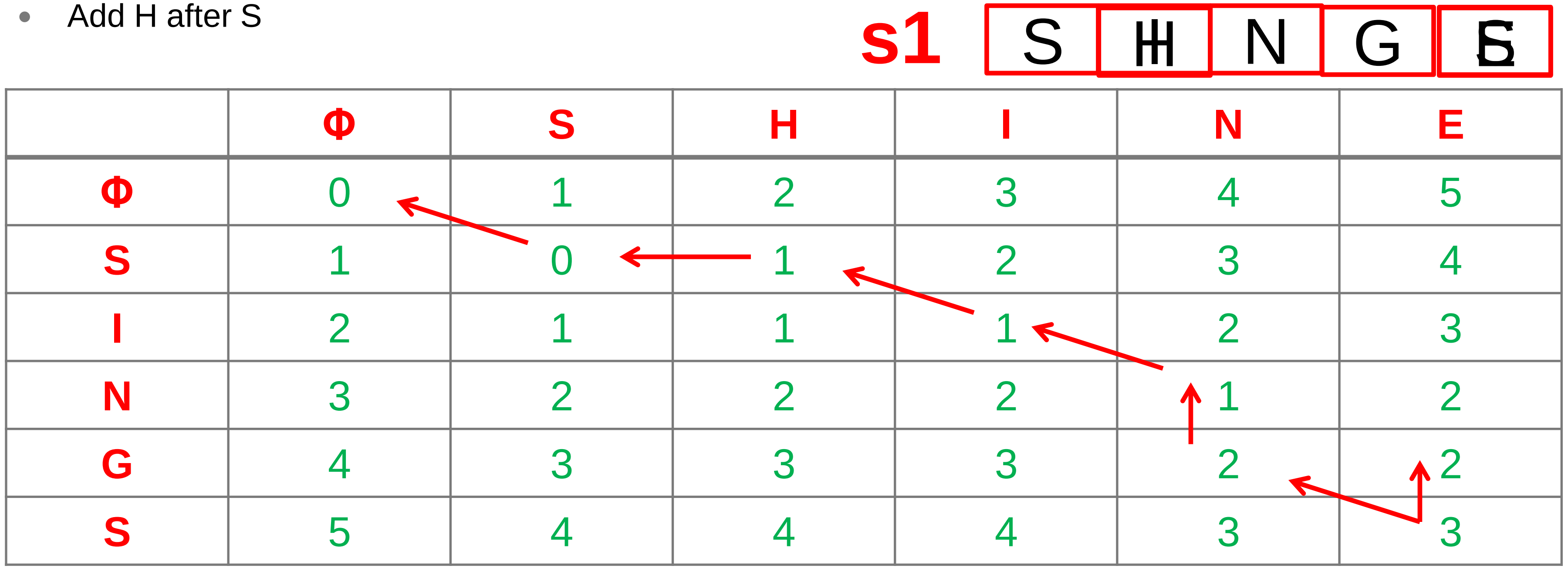
Backtracking: Use the Matrix to determine where the values are coming from (if multiple, pick any of those).

Recall: Diagonal means substitution if the letters are not the same

Upward arrow means deleting the letter s1[i]

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

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



s1

S	H	N	G	E

	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

Computing Edit Distance

As stated earlier, space complexity is O(nm) and time complexity is O(nm).

Can we do any better?

If only edit distance is to be computed, the space complexity can be reduced because we only need the last two rows of the matrix to fill it.
This reduces space complexity to O(m+n).

However, backtracking is not possible because we only keep the last two rows. i.e., the algorithm cannot determine the operations to convert s1 into s2.

Hirschberg proposed a clever algorithm to address this!

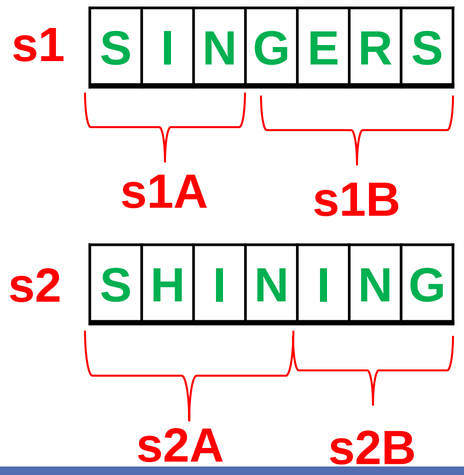
	Φ	**S**	**H**	**I**	**N**	**E**
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

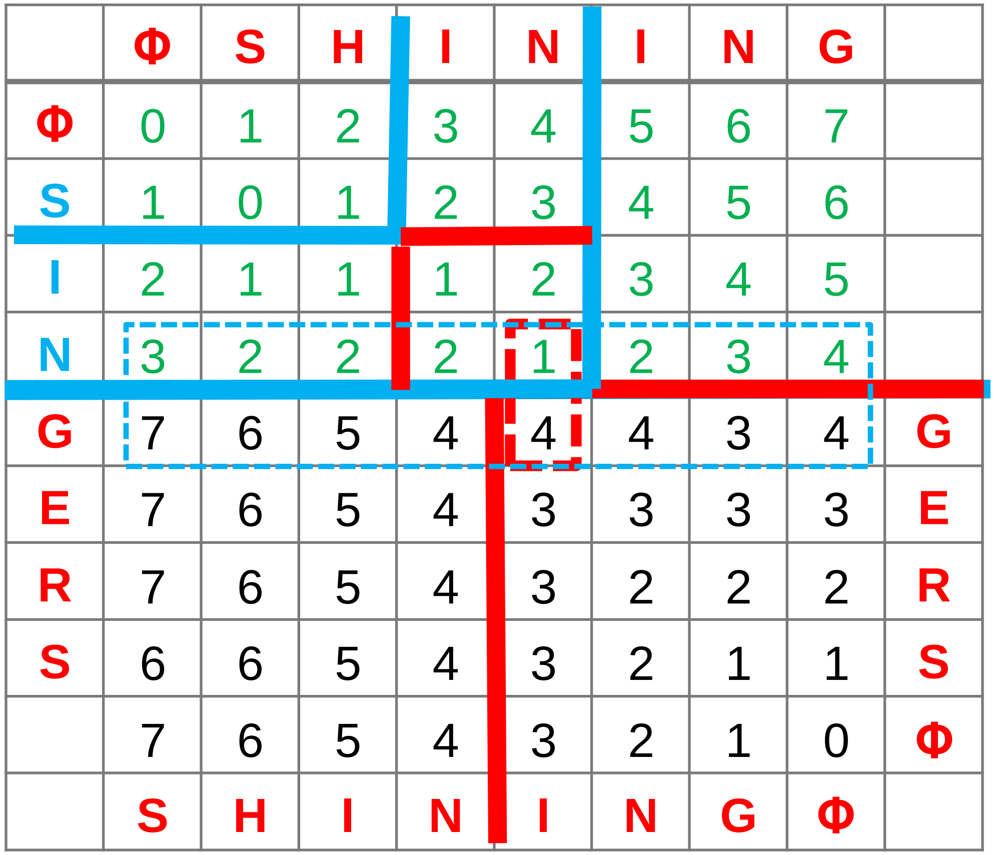
Hirschberg’s Algorithm

Divide and Conquer

- Split s1 into half. Call them s1A and s1B.
- Call edit distance algorithm on s1A and s2 maintaining only last two rows.
- Call edit distance algorithm on rev(s1B) and rev(s2) (in reverse order) maintaining only last two rows.
- Sum up the elements of the last rows for both edit distance calls and choose the cell with minimum sum as the cut point
- Divide s2 into s2A and s2B based on this cell.
- Recursively call Dist(s1A,S2A) and Dist(s1B,S2B)

Split s1 and s2 such that
Dist(s1,s2) = Dist(s1A,s2A)+ Dist(s1B,S2B)





	Φ	S	H	I	N	I	N	G	
Φ	0	1	2	3	4	5	6	7	
S	1	0	1	2	3	4	5	6	
I	2	1	1	1	2	3	4	5	
N	3	2	2	2	1	2	3	4	
G	7	6	5	4	4	4	3	4	G
E	7	6	5	4	3	3	3	3	E
R	7	6	5	4	3	2	2	2	R
S	6	6	5	4	3	2	1	1	S
	7	6	5	4	3	2	1	0	Φ
	S	H	I	N	I	N	G	Φ	

Hirschberg's Algorithm

Divide and Conquer

- Split s1 into half. Call them s1A and s1B.
- Call edit distance algorithm on s1A and s2 maintaining only last two rows.
- Call edit distance algorithm on rev(s1B) and rev(s2) (in reverse order) maintaining only last two rows.
- Sum up the elements of the last rows for both edit distance calls and choose the cell with minimum sum as the cut point
- Divide s2 into s2A and s2B based on this cell.
- Recursively call Dist(s1A,S2A) and Dist(s1B,S2B)

Time Complexity:

// n and m are lengths of s1 and s2 respectively

1st iteration: nm/2 + nm/2 = nm

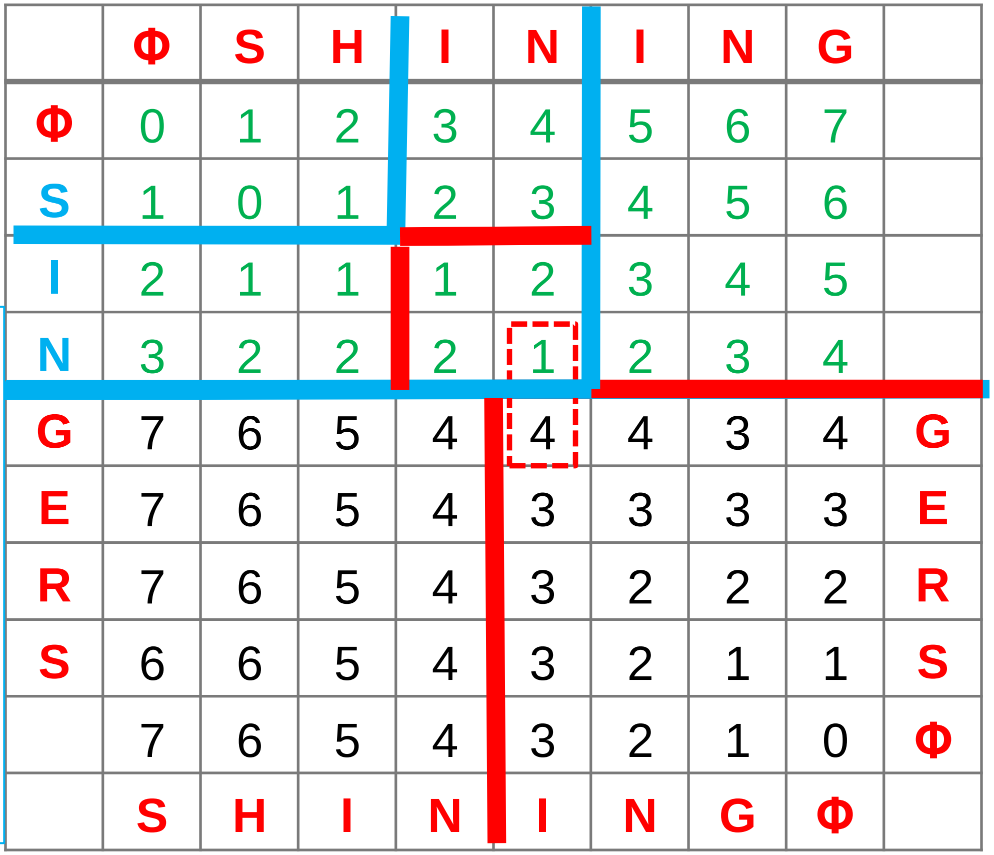
2nd iteration: nm/2

3rd iteration: nm/4

Total cost: nm (1 + ½ + ¼ + …) = O(nm)

Space Complexity:

O(m+n)



	Φ	S	H	I	N	I	N	G	
Φ	0	1	2	3	4	5	6	7	
S	1	0	1	2	3	4	5	6	
I	2	1	1	1	2	3	4	5	
N	3	2	2	2	1	2	3	4	
G	7	6	5	4	4	4	3	4	G
E	7	6	5	4	3	3	3	3	E
R	7	6	5	4	3	2	2	2	R
S	6	6	5	4	3	2	1	1	S
	7	6	5	4	3	2	1	0	Φ
	S	H	I	N	I	N	G	Φ	

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

Coming Up Next

- Hashing, Binary Search Tree, AVL Tree