Lesson 10
Algorithm Design

Wholeness of the lesson: Algorithm Design is an intelligent approach to solving problems with algorithms. Rather than simply trying to tackle a problem haphazardly, one can determine whether the problem has the characteristics that make it easy to solve using one of many known algorithm design strategies.

Maharishi's Science of Consciousness: The textbook of SCI, the Bhagavad Gita, declares "Yogastah Kuru Karmani" – Established in Being, perform action. When awareness has a chance to be bathed in the field of pure orderliness, activity afterwards has an orderly quality that naturally leads to success and achievement.

Three major techniques:

- Divide-and-Conquer
- Dynamic Programming
- The Greedy Method

Applications of Each Technique

- Divide-and-Conquer
  - Binary Search (and some operations on a BST)
  - MergeSort
  - QuickSort
- Dynamic Programming
  - Revised Recursive Fibonacci
  - Edit Distance
  - SubsetSum
  - Knapsack
- The Greedy Method
  - Shortest Path (in a graph - later)
  - Minimum Spanning Tree (in a graph - later)

Divide and Conquer

Involves solving a particular computational problem by dividing it into one or more subproblems of smaller size, recursively solving each subproblem, and then “merging” or “marrying” the solutions to the subproblem(s) to produce a solution to the original problem.

Binary Search

```
Algorithm search(A,x)
    Input: An already sorted array A with n elements and search value x
    Output: true or false
    return binSearch(A, x, 0,  A.length-1)

Algorithm binSearch(A, x, lower, upper)
    Input: Already sorted array A of size n, value x to be
           searched for in array section A[lower]..A[upper]
    Output: true or false

 if lower > upper then  return false
 mid ← (upper + lower)/2
 if x = A[mid] then  return true
 if x < A[mid]  then
       return binSearch(A, x, lower, mid – 1)
 else
       return binSearch(A, x, mid + 1, upper)
```

MergeSort

```
Algorithm mergeSort(S)
  Input sequence S with n
  Output sequence S sorted
   if S.size() > 1 then
      (S_1, S_2) ← partition(S, n/2)
      mergeSort(S_1)
      mergeSort(S_2)
      S ← merge(S_1, S_2)
   return S
```

Divide and Conquer Doesn’t Always Work Efficiently

- For Divide and Conquer to be effective, it must be possible to break up the original problem into *non-overlapping* subproblems.
  (Overlapping subproblems: the recursion tends to solve the same subproblems over and over.)
  - Example: In MergeSort, the steps of recursive sorting of the left half of the list do not affect, and are not affected by, the steps of the sorting of the right half of the list

- If something similar to Divide and Conquer is attempted when problems are overlapping, it may result in many redundant computations.
  - Example: Recursive Fibonacci

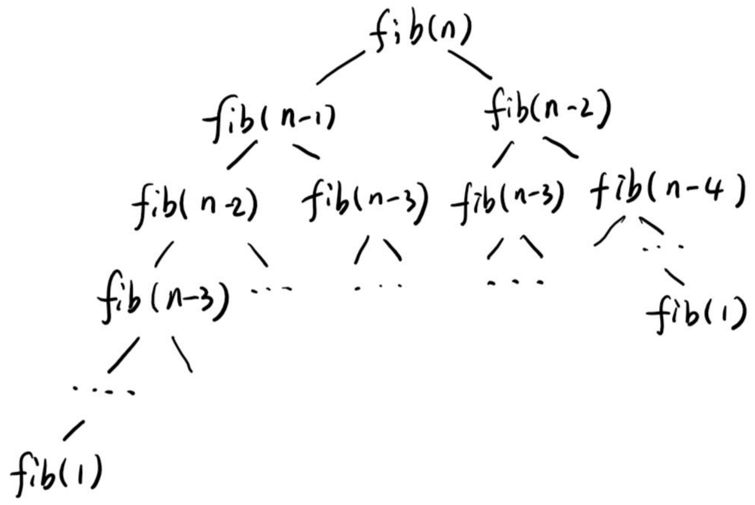
```
Algorithm fib(n)
    Input: a natural number n
    Output: F(n)

  if (n = 0 || n = 1) then return n

  return fib(n-1) + fib(n-2)
```

Running time of Recursive Fib

We will use the technique of counting self-calls here.



- Note that the fib recursion tree is never completely filled. But we know its number of nodes lies between the number of nodes in a completely filled binary tree of height n/2 and the number of nodes in a completely filled binary tree of height (n-1).
  - In a completely filled binary tree of height n/2, the number of nodes is $\Theta((\sqrt{2})^n)$
  - In a completely filled binary tree of height n-1, the number of nodes is $\Theta(2^n)$
  - The number of self-calls is bounded by the two exponential functions, so the running time is exponential.

Running time of Recursive Fib

- A running time $T(n)$ is <u>exponential</u> if there are r, s, with $1<r<s$, such that

  $$r^n \le T(n) \le s^n$$

- We have shown in previous slide that Recursive Fib runs in exponential time.

Dynamic Programming

- The idea: Sometimes problems can be broken down into *overlapping* subproblems, which can be solved, and whose solutions can be combined in some way to obtain a solution to the main problem. Solutions to subproblems are stored and combined stage by stage to produce a solution to the main problem.

(continued)

- When such a problem exhibits the following characteristics, it can in many cases be tackled using dynamic programming:
  - *Overlapping subproblems* – the subproblems "overlap" – the recursion tends to solve the same subproblems over and over (example: recursive fibonacci)
  - *Optimal substructure* – an optimal solution is composed of a combination of optimal solutions to subproblems

Dynamic Programming: Fibonacci

- To generate the nth Fibonacci number, the subproblems are to find the kth Fibonacci number for $k < n$.
- From an earlier lab, we know that in the recursive Fibonacci algorithm, many subproblems are recomputed. (Those redundant computations slow the algorithm down.) To prevent redundant computations, solutions to subproblems can be stored in a table and accessed whenever needed during execution of the algorithm.

Recursive Dynamic Programming Solution to Fibonacci

```
int recDpFib(int n) {
    if(table[n] < 0) {
        if(n == 0 || n == 1) {
            table[n] = n;
            return table[n];
        }
        table[n] = recDpFib(n-2) + recDpFib(n-1);
    }
    return table[n];
}
```

- In this dynamic programming solution, the integer table stores computations as they are made.
- Computations are made only after consulting the table to see if a computed value is already available.

Iterative Dynamic Programming Solution to Fibonacci

```
int fib(int n) {
    int[] table = new int[n+1];
    table[0] = 0;
    table[1] = 1;
    for(int i = 2; i <= n; i++) {
        table[i] = table[i-1] + table[i-2];
    }
    return table[n];
}
```

- In this dynamic programming solution, it simply fills out the table from left to right.

Dynamic Programming: The Edit Distance Problem

- The Problem: Given two strings, what is the smallest number of transformations (delete, insert, substitute) to transform one to the other?
  - Example: Transform "duck" to "tug"
    - First try: (1) delete d-u-c-k (2) insert t-u-g

      Total # transformations = 7
    - Second try: duck -> tuck -> tugk -> tug

      Total # transformations = 3
- Applications:
  - Approximate String Matching
  - Spell checking
  - Google – finding similar word variations
  - DNA sequence comparison

Recursive Solution

- When transforming one word to another one, consider the last characters of strings s and t.

- If we are lucky enough that they ALREADY match,
  - then we can simply recursively find the edit distance between the two strings left when we delete this last character from both strings.

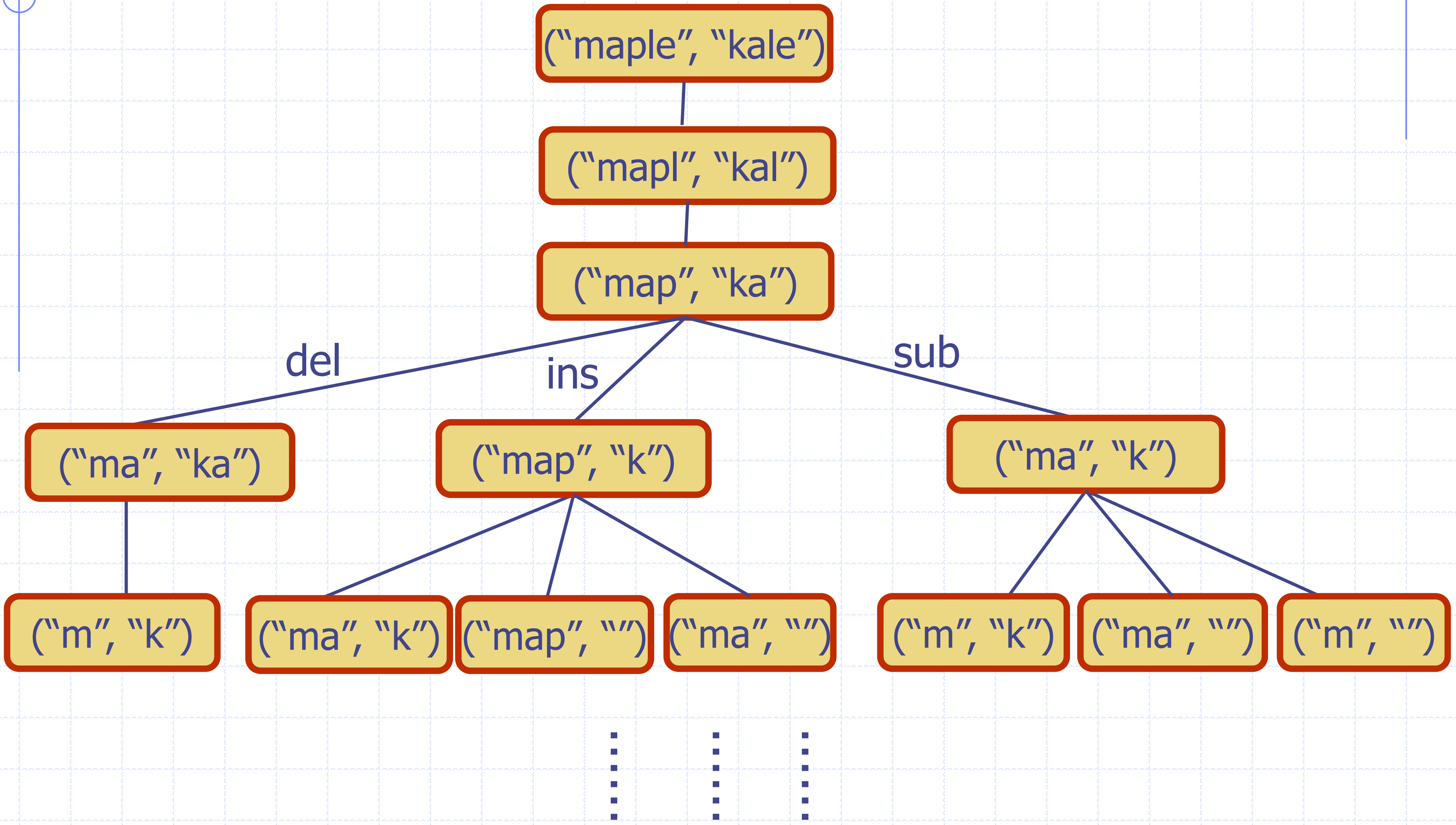
- Otherwise, we MUST make one of three changes then recursively compute the edit distance:
  1) delete the last character of string s
  2) delete the last character of string t
  3) change the last character of string s to the last character of string t.

  Note: In our recursive solution, we must note that the edit distance between the empty string and another string is the length of the second string. (This corresponds to having to insert each letter for the transformation.)

Recursive Solution

- An outline of our recursive solution is as follows:
  1) If either string is empty, return the length of the other string.
  2) If the last characters of both strings match, recursively find the edit distance between each of the strings without that last character.
  3) If they don't match then return 1 + minimum value of the following three choices:
     a) Recursive call with the string s w/o its last character and the string t
     b) Recursive call with the string s and the string t w/o its last character
     c) Recursive call with the string s w/o its last character and the string t w/o its last character.

Execution Example



Recursive Solution

Algorithm recursiveED(A_i, B_j)

Input: two strings $A_i = a_1 .... a_i$ and $B_j = b_1 ... b_j$

Output: the edit distance for A_i and B_j

```
if i = 0 then
    return j
if j = 0 then
    return i
if A[i] = B[j] then
    return recursiveED($A_{i-1}$, $B_{j-1}$)
else
    return 1+ min(recursiveED($A_{i-1}$, $B_j$),
                  recursiveED($A_i$, $B_{j-1}$),
                  recursiveED($A_{i-1}$, $B_{j-1}$))
```

Running time in the worst case is exponential. Notice it has more self-calls than recursive fib.

The Edit Distance Problem

- Now, how do we use this to create a DP solution?
    - Let us consider the subproblems. What are the subproblems?
    - Are they overlapping?
    - If so, how can we store the solutions to overlapping subproblems?

The Edit Distance Problem

- Now, how do we use this to create a DP solution?
  - Let us consider the subproblems. What are the subproblems?

  Answer: solving the edit distance problem for prefixes of s and t.

  - Are they overlapping?

  Answer: Yes

  - If so, how can we store the solutions to overlapping subproblems?

  Answer: All the possible recursive calls we are interested in are determining the edit distance between prefixes of s and t. We simply need to store the answers to all the possible recursive calls in a two dimensional array.

Dynamic Programming Solution

- Let $A = a_1 \ldots a_i \ldots a_n$ and $B = b_1 \ldots . b_j \ldots . b_m$

  define $A_i = a_1 \ldots a_i$ and $B_j = b_1 \ldots . b_j$

  $$D[i][j] = D_{i,j} = EditDistance(A_i, B_j)$$

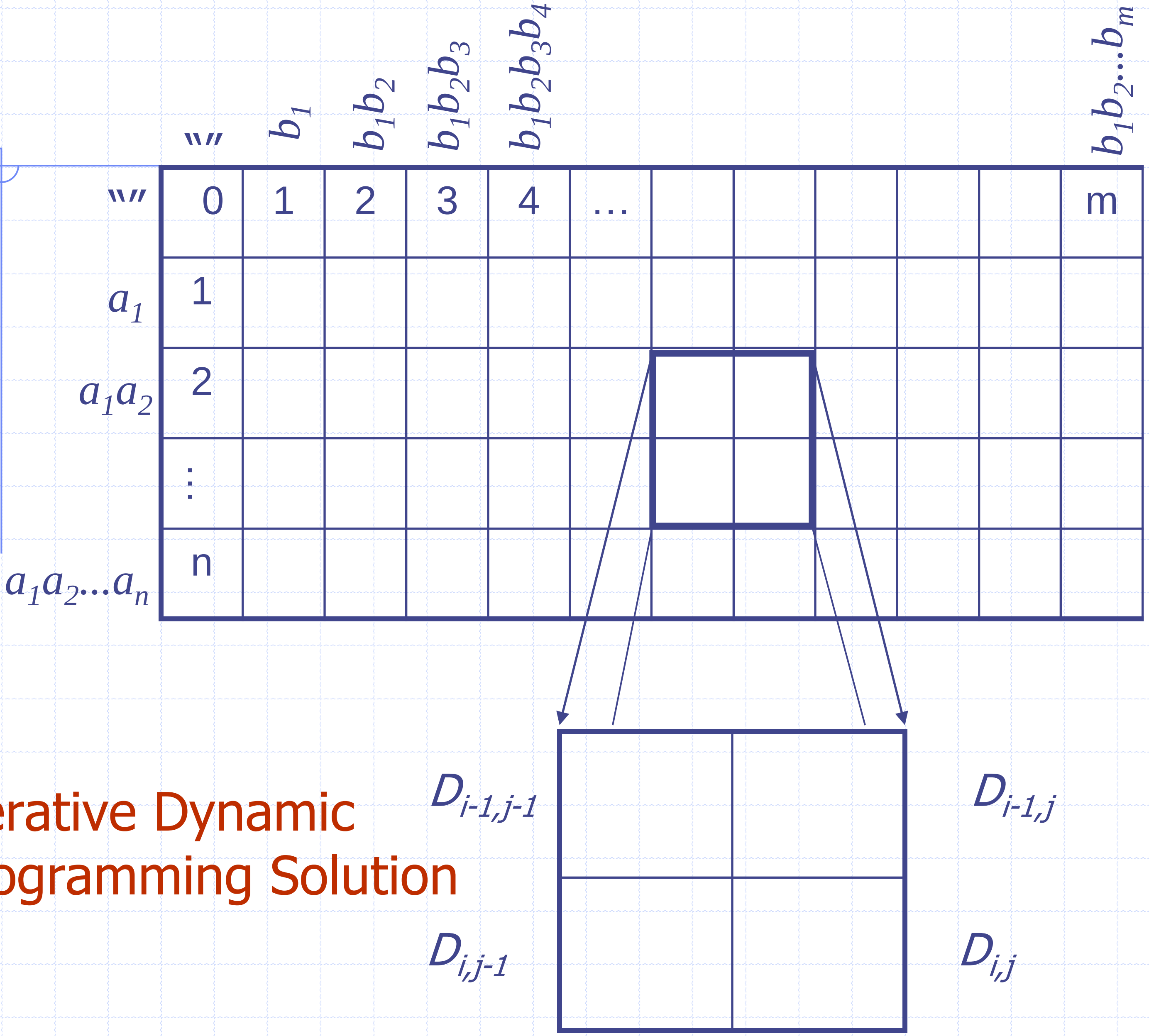
  that is, $D_{i,j}$ is the edit distance for the prefixs A_i *and* B_j

  <u>Note</u>: each recursive call gives us $D_{i,j}$

Recursive Dynamic Programming Solution

```
Algorithm recursiveED($A_i$, $B_j$)
    on first call, initialize a two dimensional array D[i+1][j+1] with all
    cells value -1
    if i = 0 then return j
    if j = 0 then return i
    if D[i][j] != -1 then return D[i][j]
    if A[i] = B[j] then
        D[i][j] $\leftarrow$ recursiveED($A_{i-1}$, $B_{j-1}$)
    else
        D[i][j] $\leftarrow$ 1+ min(recursiveED($A_{i-1}$, $B_j$),
                            recursiveED($A_i$, $B_{j-1}$),
                            recursiveED($A_{i-1}$, $B_{j-1}$))
    return D[i][j]
```

Iterative Dynamic Programming Solution



Iterative Dynamic Programming Solution

- Let $A = a_1 \ldots a_i \ldots a_n$ and $B = b_1 \ldots. b_j \ldots. b_m$
  define $A_i = a_1 \ldots a_i$ and $B_j = b_1 \ldots. b_j$

  $$D[i][j] = D_{i,j} = EditDistance(A_i, B_j)$$

  that is, $D_{i,j}$ is the edit distance for the prefixs A_i *and* B_j

- **Base cases:**

  $$D_{0,0} = 0$$
  $$D_{i,0} = i \quad \text{for } 1 \leq i \leq n$$
  $$D_{0,j} = j \quad \text{for } 1 \leq j \leq m$$

  The edit distance between an empty string and another string s is the length of s.

Iterative Dynamic Programming Solution

- Based on the observation from recursive solution, we come up with the following **Recurrence equation**:

  if (A[i] != B[j])

$$D_{i,j} = \min\left\{\begin{array}{lcl} D_{i-1,j-1} & + & 1, \\ D_{i-1,j} & + & 1, \\ D_{i,j-1} & + & 1 \end{array}\right\}$$

  if (A[i] == B[j])

$$D_{i,j} = D_{i-1,j-1}$$

Iterative Dynamic Programming Solution – building table

- Example: compute edit distance between "DUCK" and "TUG"
- Filling out first row and first column for the table.

	""	"T"	"TU"	"TUG"
""	0	1	2	3
"D"	1			
"DU"	2			
"DUC"	3			
"DUCK"	4			

Iterative Dynamic Programming Solution – building table

- Example: compute edit distance between “DUCK” and “TUG”
- Filling out the rest cells for the table …

	“”	“T”	“TU”	“TUG”
“”	0	1	2	3
“D”	1	1	2	3
“DU”	2	2	1	2
“DUC”	3	3	2	2
“DUCK”	4	4	3	3

Iterative Dynamic Programming Solution

```
Algorithm EditDistance(A,B)
Input: two strings $A = a_1 .... a_n$ and $B = b_1 ... b_m$
Output: the edit distance for A and B
    initiate a two dimensional array D[n+1][m+1]
    D[0][0] = 0
    for i ← 1 to n do D[i][0] ← i
    for j ← 1 to m do D[0][j] ← j
    for i ← 1 to n do
        for j ← 1 to m do
            if A[i] = B[j] then
                D[i][j] ← D[i-1][j-1]
            else
                D[i][j] ← 1 + min( D[i–1][j], D[i][j-1], D[i–1][j–1])
    return D[n][m]
```

Running time

- O(mn) apparently. (m,n are the lengths of the strings.)
- This algorithm is discovered by Wagner Fischer in the 1970s.

Summary

Five Steps of Dynamic Programming:

- 1. Identify subproblems. (and subproblems must be overlapping in order to use dynamic programming technique)
- 2. Characterize the structure of a solution by recursively define the solution in terms of solutions to subproblems
- 3. Locate subproblem overlap
- 4. Store overlapping subproblem solutions for later retrieval
- 5. Construct an optimal solution from the computed information gathered during steps 3 and 4

Memoization

- The basic idea
  - Design the natural recursive algorithm
  - If recursive calls with the same arguments are repeatedly made, then memoize the inefficient recursive algorithm
    - Save these subproblem solutions in a table so they do not have to be recomputed
- Implementation
  - A table is maintained with subproblem solutions and the control structure for filling in the table occurs during normal execution of the recursive algorithm
  - Often we can transform it into an iterative solution (which also called bottom-up solution.)

Main Point

Dynamic Programming is an algorithm design technique that arrives at an optimal solution by computing optimal solutions to overlapping subproblems, storing the results (memoization) to avoid redundant computations, and then combining subproblem solutions to obtain the final solution. In SCI, it is observed that to restore completeness in the life of the individual – to solve the problem of life as a human being – we must restore the *memory* of our unbounded nature. For that, we repeatedly open awareness to its unbounded nature, through the process of transcending.

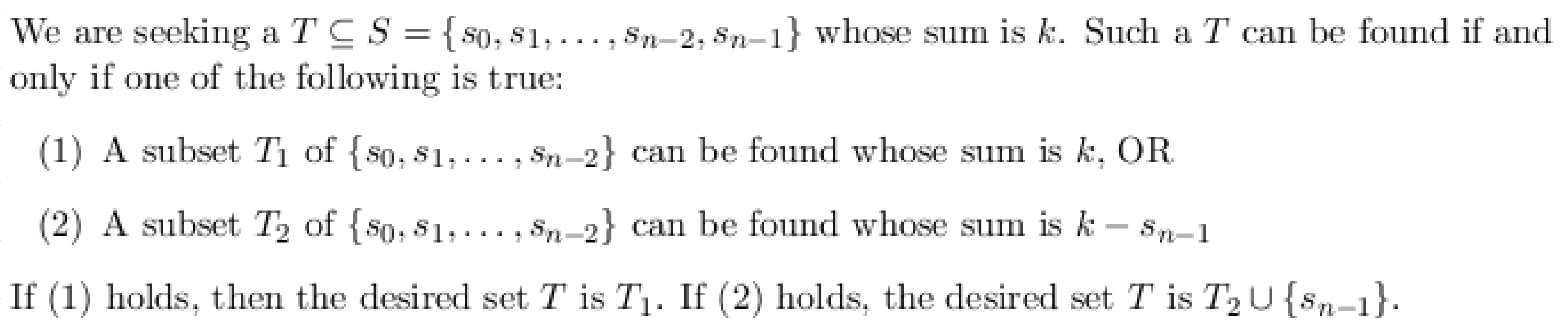
Dynamic Programming: The Subset Sum Problem

The Subset Sum optimization problem says: We have set S = $\{s_0, s_1, \ldots, s_{n-1}\}$ of n positive integers and a non-negative integer k. Find a subset T of S so that the sum of the s_r in T is k.

$$\sum_{s_r \in T} s_r = k$$

SubsetSum: Recursive Solution

A recursive solution is based on the following observation:



We are seeking a $T \subseteq S = \{s_0, s_1, \ldots, s_{n-2}, s_{n-1}\}$ whose sum is k. Such a T can be found if and only if one of the following is true:

(1) A subset T_1 of $\{s_0, s_1, \ldots, s_{n-2}\}$ can be found whose sum is k, OR

(2) A subset T_2 of $\{s_0, s_1, \ldots, s_{n-2}\}$ can be found whose sum is $k - s_{n-1}$

If (1) holds, then the desired set T is T_1. If (2) holds, the desired set T is $T_2 \cup \{s_{n-1}\}$.

The recursion proceeds by considering progressively smaller subsetsum problems, as the input set evolves from $\{s_0, \ldots, s_{n-1}\}$ to $\{s_0, \ldots, s_{n-2}\}$ to $\{s_0, \ldots, s_{n-3}\}$ to … to $\{s_0\}$.

(continued)

A recursive formula for computing T $\subseteq$ {s_0, s_1, . . ., s_{n-1}} whose sum is k is:

$$T = \begin{cases} T_1 & \text{where } T_1 \subseteq \{s_0, \ldots, s_{n-2}\} \text{ and } \sum T_1 = k \\ T_2 \cup \{s_{n-1}\} & \text{where } T_2 \subseteq \{s_0, \ldots, s_{n-2}\} \text{ and } \sum T_2 = k - s_{n-1} \\ \text{NULL} & \text{otherwise} \end{cases}$$

The base case for the recursion is the case in which the input set has been reduced just to {s_0}. The possible values for a solution T $\subseteq$ {s_0} are given by:

$$T = \begin{cases} \{\} & \text{if } k = 0 \\ \{s_0\} & \text{if } k = s_0 \\ \text{NULL} & \text{otherwise} \end{cases}$$

The base case tells us that unless self-calls eventually arrive at a subsetsum problem with input set {s_0} and whose target value is either 0 or s_0, then there is no solution to the problem.

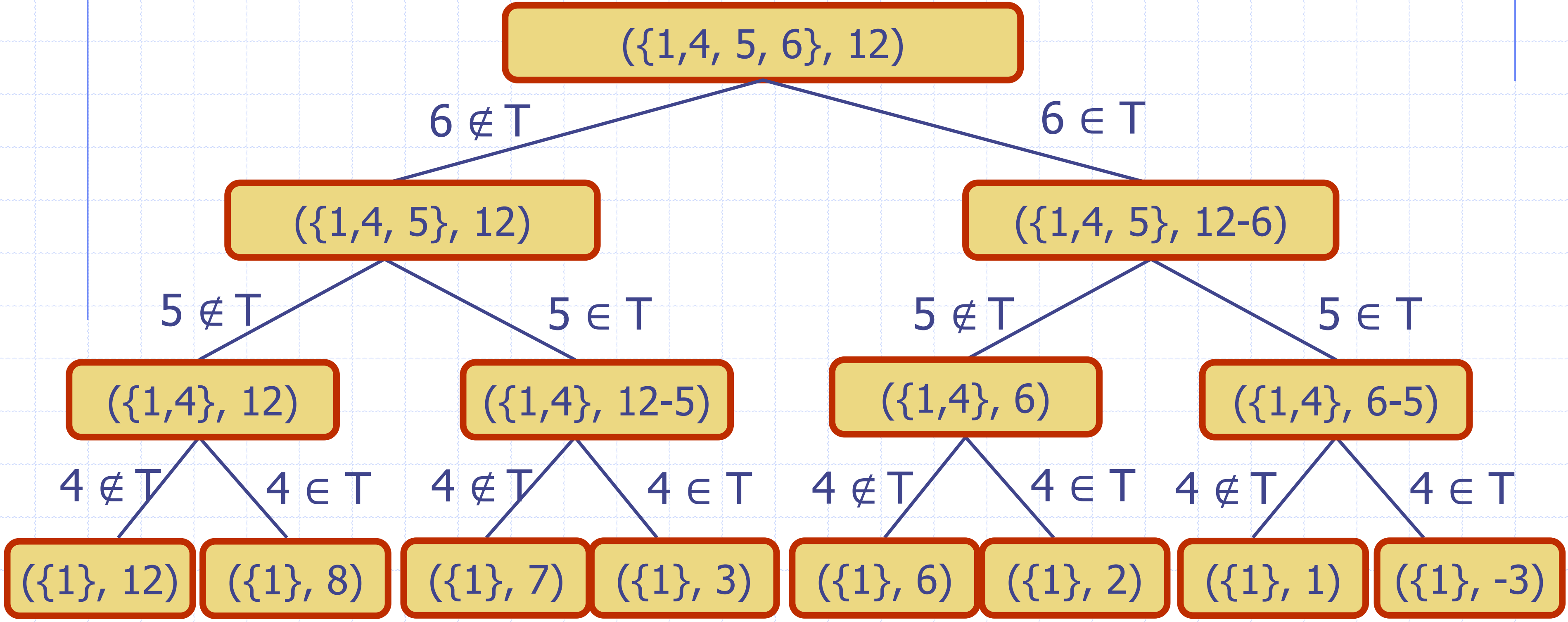
```
Algorithm RecSubsetSum(S, k)
Input: S = {$s_0, s_1, …, s_{n-1}$} positive integers,
        k nonnegative integer
Output: a subset T of S for which sum(T) = k
if S.size() = 1 then
    if k = 0 then return {}
    else if k = $s_0$ then return {$S_0$}
    else return NULL
(S', last) ← S.removeLast()
T1 ← RecSubsetSum(S', k)
if T1 not NULL then
    return T1
T2 ← RecSubsetSum(S', k-last)
if T2 not NULL then
    return T2 U {last}
return NULL
```

- The recursive algorithm tries to find a solution T for ({s_0, s_1, …, s_{n-2}, s_{n-1}}, k) by checking if a solution exists for either of the subproblems
  ({s_0, s_1, …, s_{n-2}}, k)
  ({s_0, s_1, …, s_{n-2}}, k - s_{n-1})
- To find these, it seeks solutions to smaller subproblems.
- Note that when a self-call, running on some (S,k), returns a value for T, if T is not NULL, T is guaranteed to have sum = k .

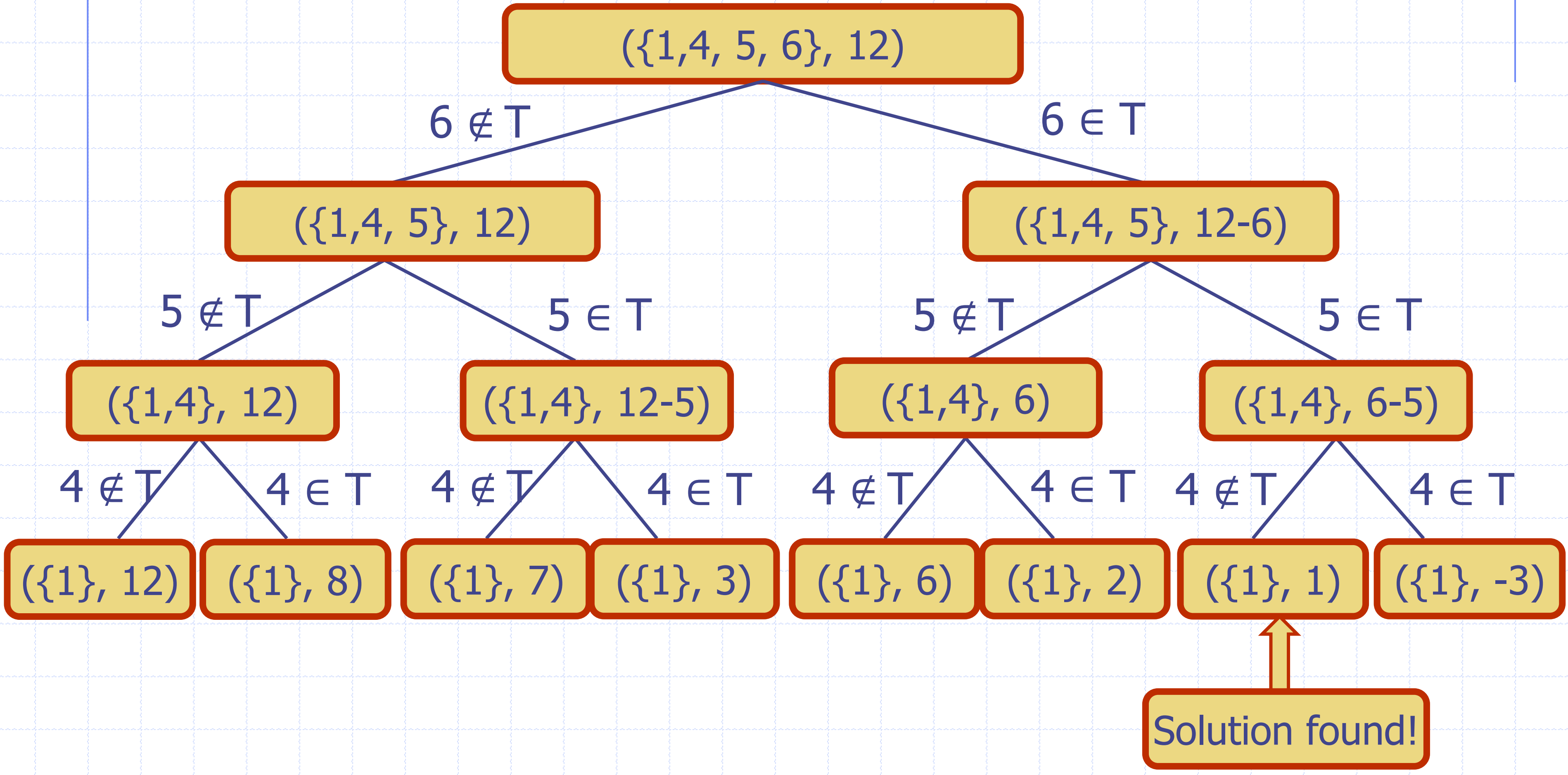
SubsetSum Recursion Tree

- An execution of SubsetSum may be depicted by a binary tree
  - each node represents a recursive call of SubsetSum and stores the set S and k
  - the link between two nodes tests whether the last element is in the solution.
  - the root is the initial call
  - the leaves are calls on set of size 1

Execution Example



Execution Example (cont.)



Running time

- We use the technique of counting self-calls to compute the running time.
- # of self-calls is approximately equal to # of nodes in the recursion tree. The recursion tree is a completely filled binary tree, thus having $2^{h+1}-1 = 2^n-1$ nodes, where h = height of tree, n = size of S (and h=n-1)
- So the running time of recursive SubsetSum is $\Theta(2^n)$.
- For larger input sets S, the recursive solution <u>will repeatedly recalculate solutions for the smaller subproblems</u> (recall how this happened with recursive Fibonacci). See Demo

  `lecture_10.subsetsum.RecursiveSS`

DP Solution: Improve Performance by Storing Computations

```
//initialize HashTable H
**Algorithm** RecSubsetSumDP(S, k)
**Input:** S = {$s_0$, $s_1$, …, $s_{n-1}$} positive integers, k nonnegative integer
**Output:** a subset T of S for which sum(T) = k
key ← (S, k)
T ← H.get(key)
if T not NULL then
    return T
if S.size() = 1 then
    if k = 0 then H.put(key, {})
    else if k = $s_0$ then H.put(key, {$S_0$})
    else then H.put(key, NULL)
    return H.get(key)
//continued…
(S', last) ← S.removeLast()
T1 ← RecSubsetSumDP(S', k)
if T1 not NULL then
    H.put(key, T1)
    return T1
T2 ← RecSubsetSumDP(S', k-last)
if T2 not NULL then
    H.put(key, T2 U {last})
    return T2 U {last}
else
    H.put(key, NULL)
    return NULL
```

DP Solution: Improve Performance by Storing Computations

- In this version, the hashtable H is consulted before performing another self-call. If the computation has already been made and stored, the stored value is used.
- Performance using memoization in this way is greatly enhanced; all redundant computations are eliminated. See demo

```
lecture_10.subsetsum.DynamicProgRecursive
```

Running Time of DP Solution: Organizing Stored Computations in a Table

- We can organize the stored computations in the recursive algorithm in a table.

$A_{i, j}$	0	1	…	j	…	k-1	k
0							
1							
…							
i				$A_{i,j}$			
…							
n-2							
n-1							$A_{n-1, k}$

- $A_{i, j}$ is a solution to the SubsetSum problem

  $(\{s_0, s_1, s_2, \dots, s_i\}, j)$

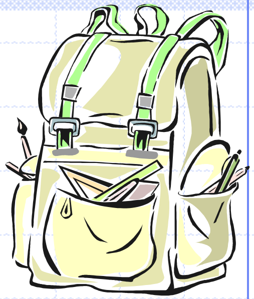
Running Time of DP Solution: Organizing Stored Computations in a Table

- We can organize the stored computations in the recursive algorithm in a table.

$A_{i, j}$	0	1	…	j	…	k-1	k
0							
1							
…							
i				$A_{i,j}$			
…							
n-2							
n-1							$A_{n-1, k}$

- Note that each self call tries to compute a cell in this table. (Same cell will not be computed more than once since solutions are stored.) Running time is O(# of self calls) = O(# of cells in the table) = O(kn)

Optional: Knapsack Problem



The Problem. Given a set $S = \{s_0, s_1, \ldots, s_{n-1}\}$ of items, weights $\{w_0, w_1, \ldots, w_{n-1}\}$ and values $\{v_0, v_1, \ldots, v_{n-1}\}$ and a max weight W, find a subset T of S whose total value is maximal subject to constraint that total weight is at most W.

Observation. If T is a solution, either s_{n-1} belongs to T or it does not.

1. If it does not, then T is a solution to the Knapsack problem with items $\{s_0, s_1, \ldots, s_{n-2}\}$ weights $\{w_0, w_1, \ldots, w_{n-2}\}$ values $\{v_0, v_1, \ldots, v_{n-2}\}$ and max weight W.
2. If s_{n-1} does belong to T, then $T - \{s_{n-1}\}$ is a solution to the Knapsack problem with items $\{s_0, s_1, \ldots, s_{n-2}\}$ weights $\{w_0, w_1, \ldots, w_{n-2}\}$ values $\{v_0, v_1, \ldots, v_{n-2}\}$ and max weight $W - w_{n-1}$.

This shows that T is built up from solutions to subproblems, and suggests a recursive solution that is similar to the recursive solution to SubsetSum.

Optional: Knapsack "Bottom Up" Solution

Knapsack Optimization Problem Given a set $S = \{s_0, s_1, \ldots, s_{n-1}\}$ of n items with positive integer weights given by $w[] = \{w_0, w_1, \ldots, w_{n-1}\}$ and nonnegative integer values $v[] = \{v_0, v_1, \ldots, v_{n-1}\}$ and a maximum weight W (a positive integer), find $T \subseteq S$ so that $\sum_{s_i \in T} v_i$ is maximal (the *maximum benefit*), subject to the constraint that $\sum_{s_i \in T} w_i \leq W$.

We describe the "bottom-up" solution, obtained by building the memoization table recursively (as was done for SubsetSum). For $0 \leq i \leq n-1$ and $0 \leq j \leq W$, we obtain an $n \times (W+1)$ matrix A of subsets of S.

$$A[i,j] = T \subseteq S \text{ where } \sum_{s_r \in T} w_r \leq j \text{ and } \sum_{s_r \in T} v_r \text{ is maximal.}$$

Recursive procedure to populate the matrix A.

Row 0:

$$A[0,t] = \begin{cases} \emptyset & \text{if } t < w_0 \\ \{s_0\} & \text{if } t \geq w_0 \end{cases}$$

Row i:

$$T_a = A[i-1,j], T_b = A[i-1, j-w_i] \cup \{s_i\}, B_a = \sum_{s_r \in T_a} v_r, B_b = \sum_{s_r \in T_b} v_r.$$

$$A[i,j] = \begin{cases} T_a & \text{if } B_a \geq B_b \\ T_b & \text{otherwise.} \end{cases}$$

Optional: Knapsack "Bottom Up" Solution

- See the Demo DynamicKnapsack-Demo.pdf
- The set stored in A[n-1, W] is the solution.
- Running time is O(nW) in terms of values, but, in terms of input size, it's $O(n * 2^{\text{length}(W)})$, which is potentially exponential in n (whenever $n \leq W$). Therefore, as in the case of SubsetSum, this algorithm for Knapsack runs in *pseudo-polynomial time.*

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. Dynamic programming can transform an infeasible (exponential) computation into one that can be done efficiently.

2. Dynamic programming is applicable when many subproblems of a recursive algorithm overlap and have to be repeatedly computed. The algorithm stores solutions to subproblems so they can be retrieved later rather than having to re-compute them.

3. Transcendental Consciousness is the silent, unbounded home of all the laws of nature.

4. *Impulses within Transcendental Consciousness:* The dynamic natural laws within this unbounded field are perfectly efficient when governing the activities of the universe.

5. *Wholeness moving within itself:* In Unity Consciousness, one experiences the laws of nature and all activities of the universe as waves of one's own unbounded pure consciousness.