Lesson 10
Algorithm Design:
Structuring the Laws of Nature in Individual Awareness

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
  - SubsetSum
  - Knapsack
  - Edit Distance
- The Greedy Method
  - Fractional Knapsack
  - Shortest Path (in a graph - later)
  - Minimum Spanning Tree (in a graph - later)

Three Major Techniques:

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

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.

Divide and Conquer Strategy

The method:

- **Divide** the problem into subproblems.
- **Conquer** the subproblems by solving them recursively.
- **Combine** the solutions to the subproblems into a solution to the problem.

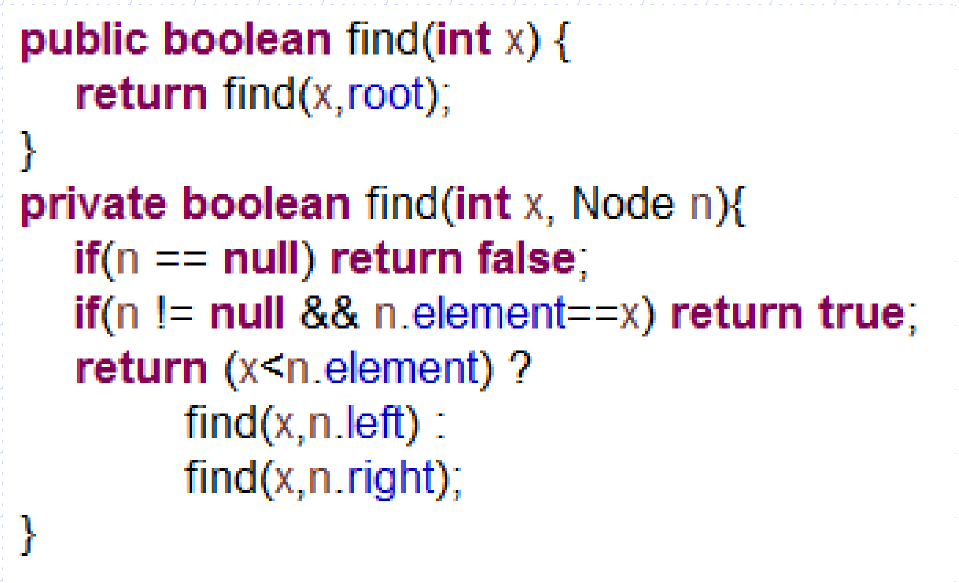
Binary Search

```
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)
```

Operations on a BST



```
public boolean find(int x) {
    return find(x,root);
}
private boolean find(int x, Node n){
    if(n == null) return false;
    if(n != null && n.element==x) return true;
    return (x<n.element) ?
            find(x,n.left) :
            find(x,n.right);
}
```

MergeSort

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

Divide and Conquer Doesn't Always Work

- For Divide and Conquer to be effective, it must be possible to break up the original problem into *non-overlapping* subproblems.
    - 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)
```

Main Point

The Divide and Conquer algorithm design attempts to solve a problem by breaking it into small disjoint subproblems, solving the subproblems separately, and combining into a final solution. This pattern of solving problems is the pattern outlined in the ancient texts by which structure emerges from the unmanifest level of existence: Analysis into parts, synthesis of parts into whole.

Through analysis and synthesis the unmanifest manifests

Absolute Theory of Defence p. 344

Three major techniques:

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

Dynamic Programming

- *Dynamic programming* is a technique that has been used to find more efficient solutions to NP-hard (more on this later) problems, though often even these solutions are still exponential.
- 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

Comparison with Divide-and-Conquer

- Divide and conquer puts together solutions to subproblems that do not overlap, like MergeSort: recursively sorting the left half does not involve any computations that are done in recursively sorting the right half; these subproblems are non-overlapping.
- Dynamic programming puts together solutions to subproblems that do overlap. For example, in order to compute fib(5), Recursive Fibonacci must re-compute fib(4), fib(3), etc. The subproblems overlap; solving one of the subproblems involves solving many others.

Dynamic Programming Example: Fibonacci

- To generate the nth Fibonacci number, the subproblems are computation of the kth Fibonacci numbers for $k < n$.
- To prevent redundant computation, solutions to subproblems can be stored in a table and accessed whenever needed during execution of the algorithm

Dynamic Programming Solution to Recursive Fibonacci

```
Algorithm FastFib(n)
    Input: n nonnegative integer
    Output: $F_n$

    if n = 0 or n = 1 then return n
    table ← new Integer array
    table[0] ← 0
    table[1] ← 1
    return RecurFastFib(n, table)


Algorithm RecurFastFib(n, table)
    Input: n nonnegative integer,
           table an Integer array
    Output: $F_n$

    if table[n-1] is null then
        table[n-1] ← RecurFastFib(n-1,table)
    table[n] ← table[n-1] + table[n-2]
    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.

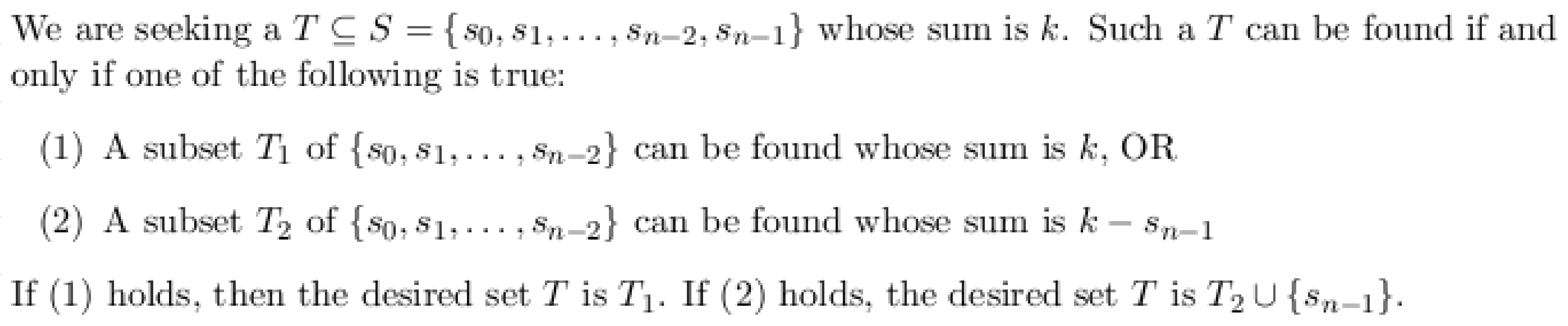
Dynamic Programming: The Subset Sum Problem

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.

(continued)

```
Algorithm RecSubsetSum(S, k)
    Input: $S = \{s_0, s_1, \ldots, s_{n-1}\}$ positive integers,
        k nonnegative integer
    Output: $T \subseteq S$ for which $sum(T) = k$
    //base case
    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()
    T ← RecSubsetSum(S, k)
    if T not NULL then
        return T
    T ← RecSubsetSum(S, k – last)
    if T not NULL then
        return T ∪ {last}
    return NULL
```

- Running time is $\Omega(2^n)$.
- The recursive algorithm tries to find a solution T for $(\{s_0, s_1, \ldots, s_{n-2}, s_{n-1}\}, k)$ by checking if a solution exists for either of the subproblems
  $(\{s_0, s_1, \ldots, s_{n-2}\}, k)$
  $(\{s_0, s_1, \ldots, s_{n-2}\}, k - s_{n-1})$
- 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 .
- For larger input sets S, the recursive solution will repeatedly recalculate solutions for the smaller subproblems (recall how this happened with recursive Fibonacci). See Demo
  `lecture_10.subsetsum.RecursiveSS`

DP Solution: Improve Performance by Storing Computations

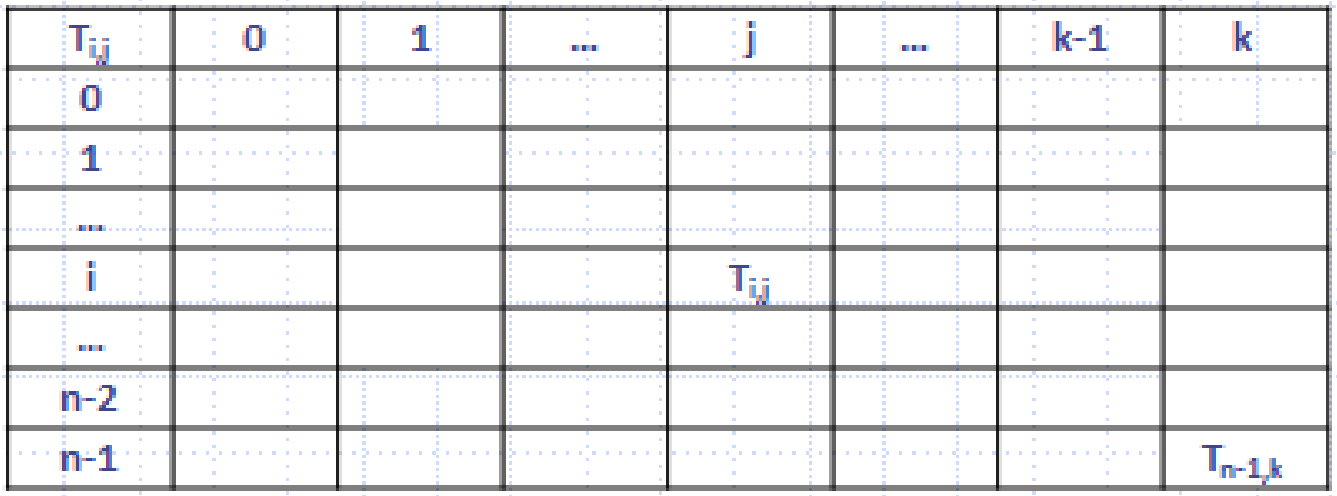
```
//Initialize hashtable H
Algorithm RecSubsetSum(S, k)
    Input: S = {s_0, s_1, . . ., s_{n-1}} positive integers,
           k nonnegative integer
    Output: T ⊆ S for which sum(T) = k
    //base case
    if S.size() = 1 then
        if k = 0 then return {}
        else if k = s_0 then return {s_0}
        else return NULL
    (S, last) ← S.removeLast()
    //read stored computation if possible
    T ← H.get((S, k))
    if T not found then
        T ← RecSubsetSum(S, k)
        H.put((S,k), T) //store for future use
    if T not NULL then
        return T
    //read stored computation of possible
    T ← H.get((S,k−last))
    if T not found
        T ← RecSubsetSum(S, k − last)
        H.put((S,k−last), T) //store for future use
    if T not NULL then
        return T ∪ {last}
    return NULL
```

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

`lecture_10.subsetsum.DynamicProgRecursive`

Organizing Stored Computations in a Table

- We can organize the stored computations in the recursive algorithm in a table. See RecDynamSubsetSum-Demo.pdf.



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

$T_{i,\,j}$ is a solution to the SubsetSum problem

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

Organizing Stored Computations in a Table

- Typically, the amount of work done to fill in the table is polynomial bounded, and the rest of the running time is insignificant.
- A "bottom-up" approach is often used instead of the recursion to fill in the table from the 0th row to the last row. The correct output is then read from the bottom right corner of the table. See BottomUpDynamicSubsetSum-Demo.pdf.

Bottom-up Approach

There are only (k+1) * n problems to solve, namely:

For $0 \leq i \leq n-1, 0 \leq j \leq k$,
find a subset $T \subseteq \{s_0, s_1, \ldots, s_i\}$ so that
$\sum_{s_r \in T} s_r = k$.

Build a solution for bigger values of i and j using stored solutions for smaller values of i and j.

Values for Each Cell of Table

Obtain a 2-dimensional array (a matrix) A so that

$$A[i,j] = \begin{cases} T & \text{where } T \subseteq \{s_0, s_1, \ldots, s_i\}, \sum_{s_r \in T} s_r = j \\ \text{NULL} & \text{if such a } T \text{ does not exist} \end{cases}$$

- If S contains values > k, we ignore them since they don't contribute to the solution (computations for which j is too big are skipped – see the implementation in code)
- Fill row i = 0 first, then fill later rows based on values of earlier rows.

Details

Row 0:

$$\begin{aligned} A[0,0] &= \emptyset \quad \text{and} \quad A[0,s_0] = \{s_0\} \\ A[0,e] &= \text{NULL whenever } e \neq 0 \text{ and } e \neq s_0 \end{aligned}$$

Note: $\sum_{s_r \in \emptyset} s_r = 0$ and $\sum_{s_r \in \{s_0\}} s_r = s_0$.

Row i:

$$A[i,j] = \begin{cases} T = A[i-1,j] & \text{if } \sum_{s_r \in T} s_r = j \\ T = A[i-1,j-s_i] \cup \{s_i\} & \text{if } \sum_{s_r \in T} s_r = j \end{cases}$$

Note: In computation of $A[i,j]$, a value of NULL in both $A[i-1,j]$ and $A[i-1,j-s_i]$ means that $A[i,j] = \text{NULL}$.

Pseudo-polynomial time

- The dynamic programming solution to SubsetSum (using either the recursive or bottom-up approach) runs in O(kn)

- However, k may be much bigger than n, and even if k is Θ(n), the true running time is based on the number of bits in k, not on the value of k. So even this algorithm runs in exponential time in terms of input size.

- Whenever an algorithm’s running time is polynomial in the numeric value of the input, but is exponential in the size of the input (size = the number of bits required to represent it), the algorithm is said to run in *pseudo-polynomial time*.

  [These points will be discussed in more detail later in the course.]

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 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

Brute Force 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.)

Brute Force 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.

Brute Force 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 and B
    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 min(recursiveED($A_{i-1}$, $B_j$) + 1,
                   recursiveED($A_i$, $B_{j-1}$) + 1,
                   recursiveED($A_{i-1}$, $B_{j-1}$) + 1)
```

Running time

Let n be the sum of the lengths of the two strings.

(Each string has length n/2).

$$T(n) \geq T(n-1) + T(n-1) + T(n-2)$$
$$\geq 3T(n-2)$$

Using guessing method, we can show

$T(n)$ is in $\Omega((\sqrt{3})^n)$

In other words, T(n) is exponential.

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 $B_j = b_1 \ldots. b_j$

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

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

  Note: each recursive call gives us $D_{i,j}$

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

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 $B_j = b_1 \ldots . b_j$

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

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

- **Base cases:**

  $$D_{0,0} = 0$$

  $$D_{i,0} = i \quad \text{for } 1 \le i \le n$$

  $$D_{0,j} = j \quad \text{for } 1 \le j \le m$$

  we can think of A as $a_0 a_1 \ldots a_i \ldots a_n$ and B as $b_0 b_1 \ldots . b_j \ldots . b_m$ (a_0 and b_0 are empty characters), the edit distance between an empty string and another string is the length of the non-empty string.

Iterative Dynamic Programming Solution

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

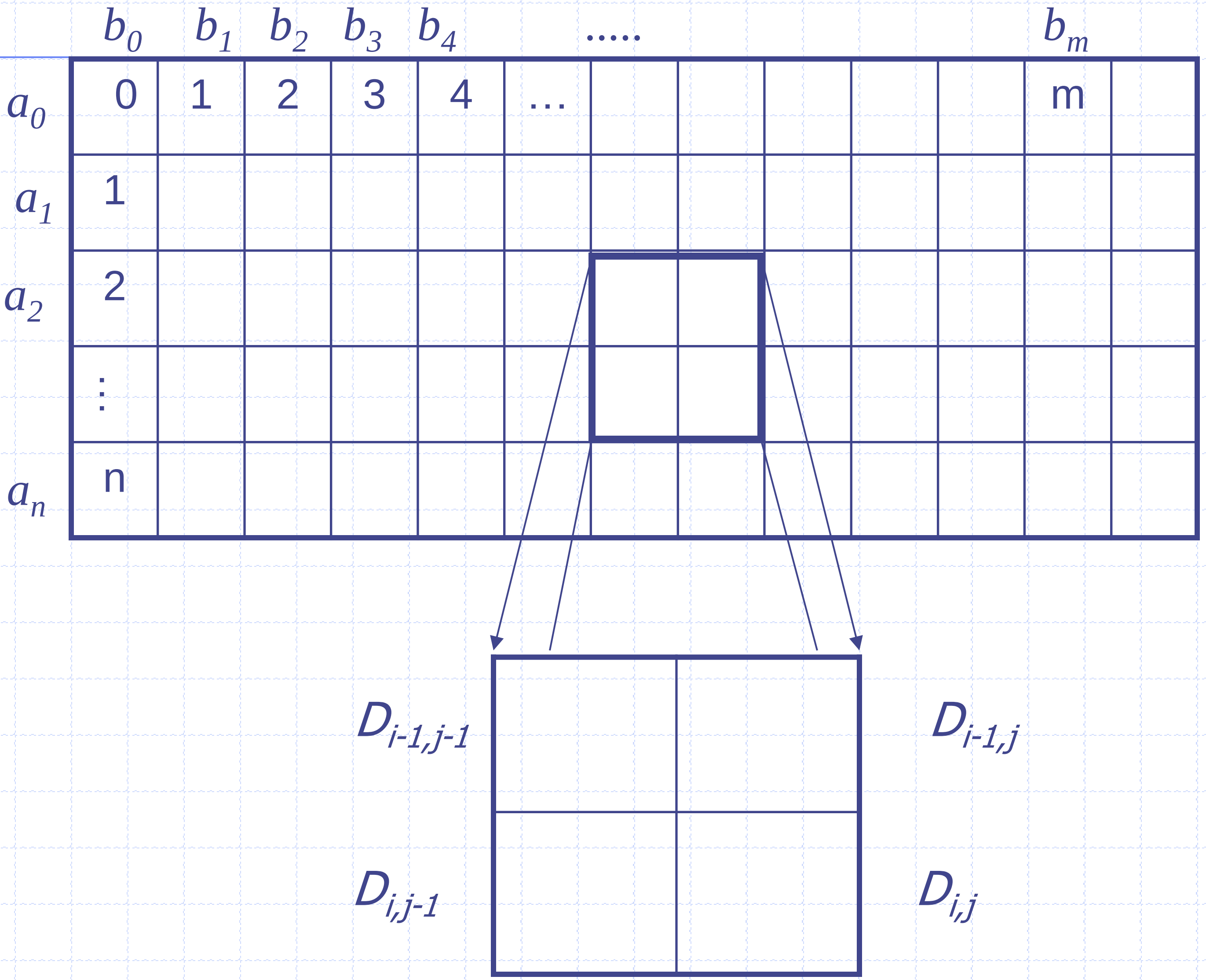
  if (A[i] != B[j])

$$D_{i,j} = \min\begin{Bmatrix} D_{i-1,j-1} & + & 1, \\ D_{i-1,j} & + & 1, \\ D_{i,j-1} & + & 1 \end{Bmatrix}$$

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

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

Iterative Dynamic Programming Solution



Iterative Dynamic Programming Solution – building table D

- Filling out first row and first column for D …

	b_0=''	b_1='T'	b_2='U'	b_3='G'
a_0=''	0	1	2	3
a_1='D'	1			
a_2='U'	2			
a_3='C'	3			
a_4='K'	4			

Iterative Dynamic Programming Solution – building table D

- Filling out the rest cells for D …

	b_0=‘’	b_1=‘T’	b_2=‘U’	b_3=‘G’
a_0=‘’	0	1	2	3
a_1=‘D’	1	1	2	3
a_2=‘U’	2	2	1	2
a_3=‘C’	3	3	2	2
a_4=‘K’	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] ← min( D[i–1][j] + 1,
                           D[i][j-1] + 1,
                           D[i–1][j–1] + 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.

Dynamic Programming: 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.

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 $\sum_{s_i \in T} w_i \le W$.

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

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

Recursive procedure to populate the matrix A.

Row 0:

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

Row i:

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

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

(continued)

Example

```
w[] = {2, 3, 4, 5}
v[] = {1, 2, 3, 4}
W   = 5

Results:
---------
Row 0:
   [], [], [s0], [s0], [s0], [s0]
Row 1:
   [], [], [s0], [s1], [s1], [s0, s1]
Row 2:
   [], [], [s0], [s1], [s2], [s0, s1]
Row 3:
   [], [], [s0], [s1], [s2], [s3]
```

The subset T of the original items that produces the greatest benefit subject to the constraint appears in the bottom right corner of the matrix. For this problem, the set T is $\{s3\}$.

Knapsack

- 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.*
- See the Demo DynamicKnapsack-Demo.pdf

Three major techniques:

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

Greedy Algorithms

- Apply to optimization problems
  - Some quantity is to be minimized or maximized.
- Key technique is to make each choice in a locally optimal manner
  - The hope is that these locally optimal choices will produce the globally optimal solution
- Does not always lead to an optimal solution.

Greedy Approach to Knapsack

- Review of 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.
- Greedy Strategy #1 Try arranging S in decreasing order of value – call the newly arranged elements S'. Then scan S from left to right to populate a solution T in the following way. If the weight of S'[0] is not more than W, put S'[0] into T (if bigger than W, then continue scanning S' from left to right till you find an item whose weight is ≤ W and put into T). Then examine the remaining items and pick the first whose weight, when added to the weight of first item, is ≤ W. Continue like this till you have scanned S' to the end, or till the total sum of weights of items in T is exactly W.

Exercise

- Example: Use Greedy Strategy #1 to solve:
  S = {s0, s1, s2}, v[] = {1, 3, 4}, w[] = {1, 2, 4}, W = 4
- Re-arrange S, v[], w[] so that items occur in decreasing order of value:
  v[] = {4, 3, 1}, w[] = {4, 2, 1}, S= {s2, s1, s0} W =4
- Following the strategy, the final solution is T= {s2}; we stop because weight of s2 is W. Solution is correct!

- Problem: Does the strategy always work?

Answer to Problem

Doesn't always work!

- Try S = {s0, s1, s2}, w[] = {3, 2, 2}, v[] = {4, 3, 2}, W = 4.
- Greedy solution (#1) is T = {s0}, but correct solution is T = {s1, s2}.
- Note: Going for biggest value first overlooks better choices
- Alternative: Go for best value per weight.

Exercise

- Greedy Strategy #2. Try arranging S in decreasing order of *value per weight*. For each i, let $b_i = v_i/w_i$. Scan the new arrangement S′ of S and put in items as long as the weight restriction permits; skip over items that will cause the weight to exceed W.
- Example. $S = \{s_0, s_1, s_2\}$, $v[] = \{1, 3, 4\}$, $w[] = \{1, 2, 4\}$, $W = 4$.
  - Compute: $b_0 = 1$, $b_1 = 1.5$, $b_2 = 1$ and arrange by decreasing order of the b_i:

    $S' = \{s_1, s_0, s_2\}$, $v'[] = \{3, 1, 4\}$, $w'[] = \{2, 1, 4\}$.
  - Now load the knapsack with items from S until becomes impossible to add any more because of the weight restriction.
    - $w'[0] = 2 \leq 4 = W$, so add $S'[0] = s_1$ to T
    - $w'[0] + w'[1] = 3 \leq 4 = W$, so add $S'[1] = s_0$ to T
    - We cannot add the final item because of the weight restriction.
- Solution $T = \{s_1, s_0\}$ is correct! Does this strategy always work? (Lab exercise)

The Fractional Knapsack Problem

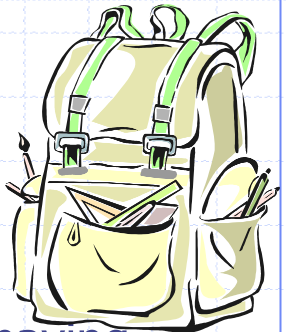
- No greedy algorithm is known for solving Knapsack (but recall, there is a dynamic programming solution)
- There is however a greedy solution to a variation of the Knapsack Problem called *Fractional Knapsack.*
- In Fractional Knapsack, you can pick any fraction of an item that you want – you are not required to use the whole item.

Statement of the Fractional Knapsack Problem

Begin with a max weight W and a set
$S = \{s_0, s_1, \ldots, s_{n-1}\}$ of n items having weights given in the weights array $w[] = \{w_0, w_1, . . ., w_{n-1}\}$ and values in the vales array $v[] = \{v_0, v_1, . . ., v_{n-1}\}$.

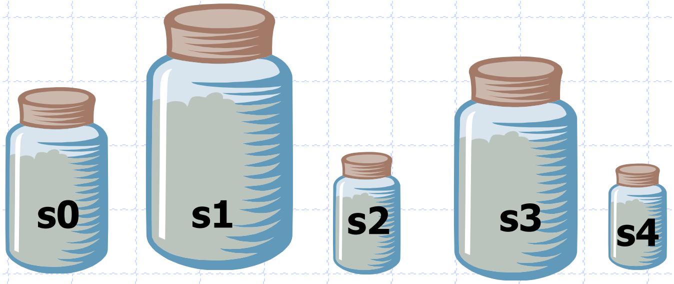
The objective is to come up with fractions $x_0, x_1, . . .,$ x_{n-1}, each in the range [0,1], so that the sum of the numbers $x_i w_i$ for s_i in T is $\leq$ W and the sum of the numbers $x_i v_i$ for s_i in T is the maximum possible. (Note that some of the fractions may equal 0.)

Example



- Given: A set S = {s_0, s_1, . . ., s_{n-1}} of n items, with each item s_i having
  - v_i - a positive value
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W. You may use just a fraction x_i of each item s_i

Items:



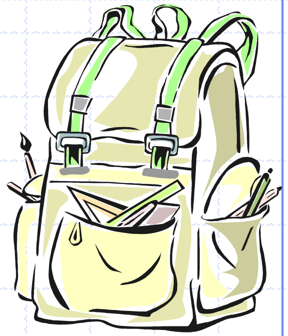
Weights:	4 ml	8 ml	2 ml	6 ml	1 ml
Values:	\$12	\$32	\$40	\$30	\$50
\$/ml:	3	4	20	5	50

W=10 ml

Greedy Solution:

- 1 ml of s4 (x_4=1.0)
- 2 ml of s2 (x_2=1.0)
- 6 ml of s3 (x_3=1.0)
- 1 ml of s1 (x_1=.125)
- 0 ml of s0 (x_0 = 0)

The Fractional Knapsack Algorithm



```
Algorithm fractionalKnapsack(S, W)
    Input: set S of n items with values $v_i$
        weights $w_i$, and max weight $W$
    Output: fraction $x_i$, $0 \le x_i \le 1$, of
    each item $s_i$ to maximize value,
    with weight at most $W$
    for each item $s_i$ in S
        $x_i \leftarrow 0$
        $b_i \leftarrow v_i / w_i$        {benefit of $s_i$}
    $w \leftarrow 0$            {current total weight}
    while $w < W$ and !S.isEmpty()
        remove item $s_i$ with highest $b_i$
        $x_i \leftarrow \begin{cases} 1 & \text{if } w_i \le W - w \\ \frac{W-w}{w_i} & \text{otherwise} \end{cases}$
        $w \leftarrow w + x_i w_i$
```

Greedy choice: Keep picking item with highest **benefit** (value-to-weight ratio v_i/w_i)

Running Time

It takes O(n log n) to sort the benefits and O(n) to scan the sorted list of benefits and perform the needed computations.

Therefore, FractionalKnapsack runs in O(n log n)

Main Point

The Greedy algorithm design attempts to solve a problem by accepting, at each step of computation, a value that is optimal at that step, without regard for future steps or for the emerging context. The greedy strategy is "doing without planning for the future." In life, the greedy strategy, according to SCI, works well only when individual life is directed by a higher quality of intelligence. Then the "planning" is done by cosmic intelligence. But until the home of all the laws of nature is established in individual awareness, it is better to plan carefully for the future and to avoid the "greedy strategy."

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.