Lesson 3
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
- Backtracking

Applications of Each Technique

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

Four Major Techniques:

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

Divide andConquer

Involves solving a particular computational problem by dividing it into one or more subproblems of smaller size, recursively solving each subproblem, and then “merging” 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 (divide input array into left and right halves)
- **Conquer** the subproblems by solving them recursively (search recursively in whichever half could potentially contain target element)
- **Combine** the solutions to the subproblems into a solution to the problem (return value found or indicate not found)

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

QuickSort

Algorithm *quickSort*(*S*)

Input sequence *S*

Output S in sorted order

if(|S|=0 or |S|=1) then **return** S

$p \leftarrow$ *pickPivot()*

(*L*,*E*,*G*) $\leftarrow$ *partition*(*S*,*p*)

quickSort(*L*)

quickSort(*G*)

return $L \cup E \cup G$

QuickSelect Pseudo-Code

Algorithm *QuickSelect*(*S*,*k*)
  Input sequence *S*, rank *k*
  Output *k*th smallest element of S
  p ← *pickPivot()*
  (*L*,*E*,*G*) ← *partition*(*S*,*p*)
  if |L| < k ≤ |L| + |E| **then**
    return any element of *E*
  else if k ≤ |L| **then**
    return *QuickSelect*(*L*, k)
  else {k > |L| + |E|}
    return *QuickSelect*(*G*, k – |*L*| – |*E*|)

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

Four major techniques:

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

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

```
public class RecursiveFibFast {
    Integer[] table;
    public int fib(int n) {
        if(n < 0) return -1;
        if(n==0 || n == 1) return n;

        //set up table for storing computations
        table = new Integer[n + 1];
        table[0] = 0;
        table[1] = 1;
        return recursiveFib(n);
    }
    private int recursiveFib(int n) {
        System.out.println("Self-call at n = " + n);
        if(table[n-1] == null) {
            table[n-1] = recursiveFib(n-1);
        }
        table[n] = table[n-1] + table[n-2];
        return table[n];
    }
}
```

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 $T_2 \cup \{s_{n-1}\}$.

(continued)

```
int[] origS;
int[] subsetsum(int[] S, int k) {
    this.origS = S;
    return recSubsetSum(makeCopy(S), k);
}
int[] recSubsetSum(int[] S, int k) {
    if (S == null || S.length == 0) {
        if (k == 0)
            return Constants.EMPTY_SET;
        else
            return Constants.NULL;
    }
    int n = S.length;
    int last = S[n - 1];
    S = removeLast(S);

    // See if restricting T to {0,1,...,n-2} is enough
    int[] T = recSubsetSum(S, k);
    if (sum(T) == k)
        return T;
    else {
        // This step effectively ignores all elements of S that are too big
        if (k >= last) {
            T = recSubsetSum(S, k - last);
            if (sum(T) == k - last)
                return adjoin(T, n - 1);
            else
                return Constants.NULL;
        } else
            return Constants.NULL;
    }
}
```

(continued)

- 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
- As n gets larger, the recursive solution will repeatedly recalculate solutions for the smaller subproblems (recall how this happened with recursive Fibonacci)
- We can speed up the recursive approach by storing solutions to subproblems in a table (*memoization*). See code Demo.

(continued)

- We can organize the stored computations in the recursive algorithm in a table (see RecDynamSubsetSum-Demo.pdf)
- 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 typically used to fill in the table from the 0^{th} row to the last row. The correct output is then read from the bottom right corner of the table. See DynamicSubsetSum-Demo.pdf.

Bottom-up Approach

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

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

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

The Goal

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 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.

Note: For a pseudo-polynomial time algorithm, its running time is polynomial in the numeric value of the input, but is exponential in the length of the input – the number of bits required to represent it.

[This point will be discussed in more detail later]

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: 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 $\sum_{s_i \in T} w_i \leq W$.

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:

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

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

(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 $O(n * 2^{length(W)})$, which is potentially exponential in n (whenever $n \leq W$)
- See the Demo DynamicKnapsack-Demo.pdf

Three major techniques:

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

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)

Introducing 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

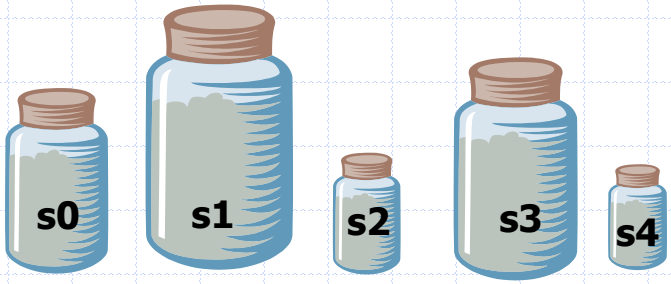
Begin with a max weight W and a set $S = \{s_0, s_1, \ldots, s_{n-1}\}$ of n items having weights given in the weights array $w[] = \{w_0, w_1, \ldots, w_{n-1}\}$ and values in the vales array $v[] = \{v_0, v_1, \ldots, v_{n-1}\}$.

The objective is to come up with fractions $x_0, x_1, \ldots, x_{n-1}$, 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 = $\{s_0, s_1, \ldots, s_{n-1}\}$ 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:

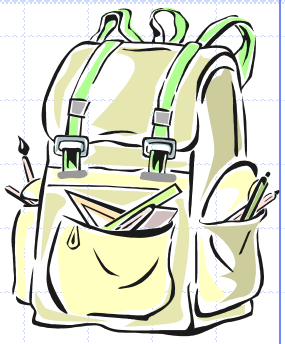


Weights:	4 ml	8 ml	2 ml	6 ml	1 ml
Values:	$12	$32	$40	$30	$50
$/ml:	3	4	20	5	50

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)

Correctness

Idea of Proof of Correctness:

1. We can find an optimal solution that begins with the first step of the greedy algorithm, where the fraction x of the item s having greatest benefit is obtained: Given any solution that uses a smaller amount than x of item s, we can replace any amount of some other item that is used by an equivalent additional amount of s and get a greater value (since s gives greatest benefit)
2. We can remove s from the set S of items under consideration and now make a greedy choice toward an optimal solution for S-{s}. By 1, we can find an optimal solution this way.
3. Keep obtaining optimal solutions for smaller and smaller subproblems. The resulting sequence of x_i gives an optimal solution to the original problem S.

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.