Lecture 11: Greedy Algorithms and Dynamic Programming

Spontaneous Right Action

Wholeness Statement

Greedy algorithms are primarily applicable in optimization problems where making the locally optimal choice eventually yields the globally optimal solution. Dynamic programming algorithms are also typically applied to optimization problems, but they divide the problem into smaller subproblems, then solve each subproblem just once and save the solution in a table to avoid having to repeat that calculation. Memoization is a technique for implementing dynamic programming in recursive algorithms to reduce complexity from exponential to polynomial time. *Science of Consciousness*: Pure intelligence always governs the activities of the universe optimally and with minimum effort.

Another Algorithm Design Method

- Greedy Strategy
  - Examples:
    - Fractional Knapsack Problem
    - Task Scheduling
    - Shortest Path (later)
    - Minimum Spanning Tree (later)
- Requires the Greedy-Choice Property

Another Important Technique for Design of Efficient Algorithms

- Useful for effectively attacking many computational problems
- Greedy Algorithms
  - Apply to optimization problems
  - Key technique is to make each choice in a locally optimal manner
  - Many times provides an optimal solution much more quickly than does a dynamic-programming solution

The Greedy Method: Outline and Reading

- The Greedy Design Technique (§5.1)
- Fractional Knapsack Problem (§5.1.1)
- Task Scheduling (§5.1.2)

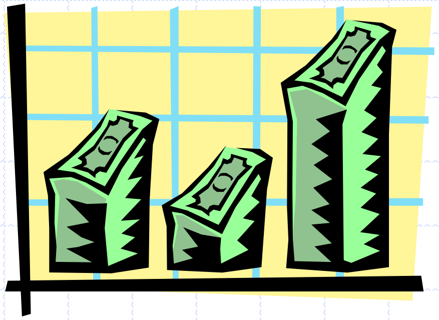
[future lectures]

- Lesson 13: Shortest Path (§7.1)
- Lesson 14: Minimum Spanning Trees (§7.3)

Greedy Algorithms

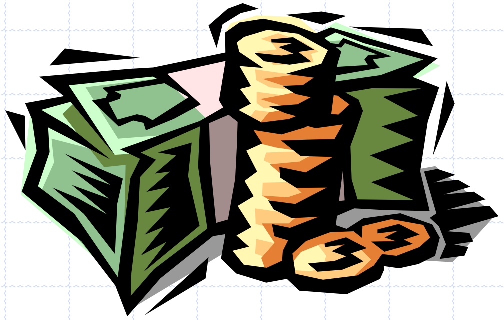
- Used for optimizations
  - some quantity is to be minimized or maximized
- Always make the choice that looks best at each step
  - the hope is that these locally optimal choices will produce the globally optimal solution
- Works for many problems but NOT for others

The Greedy Design Technique



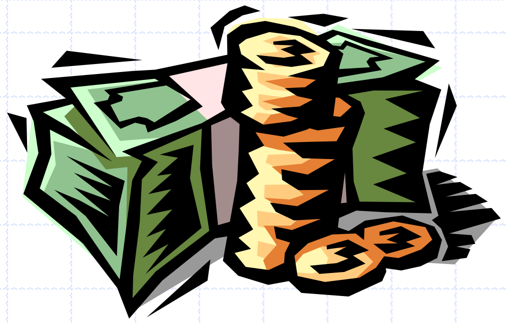
- A general algorithm design strategy,
- Built on the following elements:
  - **configurations**: represent the different choices (collections or values) that are possible at each step
  - **objective function**: a score is assigned to configurations (based on what we want to either maximize or minimize)
- Works when applied to problems with the **greedy-choice** property:
  - A globally-optimal solution can always be found by
    - Beginning from a starting configuration
    - Then making a series of local choices or improvements

Making Change



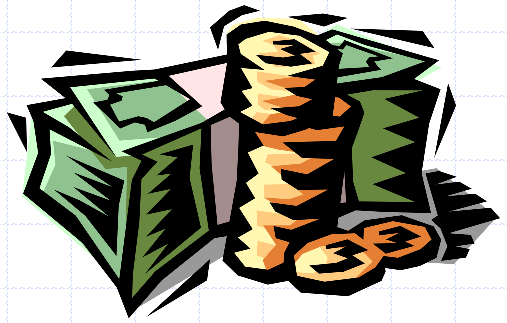
- Problem: A dollar amount to reach and a collection of coin values to use to get there.
- Configuration: A dollar amount yet to return to a customer plus the coins already returned
- Objective function: Minimize number of coins returned.
- Greedy solution: At each step return the largest coin without going over the target
- Example 1: Coins are valued \$.50, \$.25, \$.10, \$.05, \$.01
  - Has the greedy-choice property, since no amount over \$.50 can be made with a minimum number of coins by omitting a \$.50 coin (similarly for amounts over \$.25, but under \$.50, etc.)

Making Change



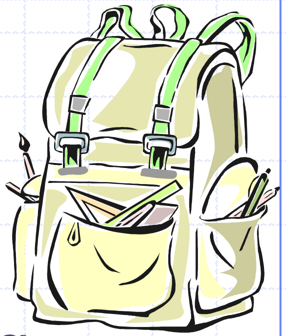
- Example 2: Coins are valued $.30, $.20, $.05, $.01
  - Do coins with these values have the greedy-choice property for making change?
- Example 3: Coins are valued $.32, $.08, $.01
  - Do these coins have the greedy-choice property?

Making Change



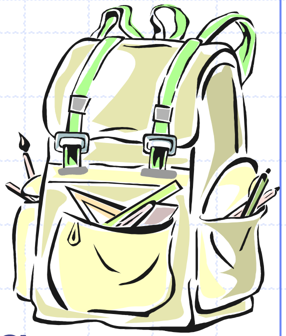
- Example 2: Coins are valued $.30, $.20, $.05, $.01
  - Does not have greedy-choice property, since $.40 is best made with two $.20's, but the greedy solution will pick three coins (which ones?)
  - What if we added a coin worth $.10?
  - What if we removed $.20 and added $.15?
- Example 3: Coins are valued $.32, $.08, $.01
  - Has the greedy-choice property, since no amount over $.32 can be made with a minimum number of coins by omitting a $.32 coin (similarly for amounts over $.08, but under $.32).

The Fractional Knapsack Problem



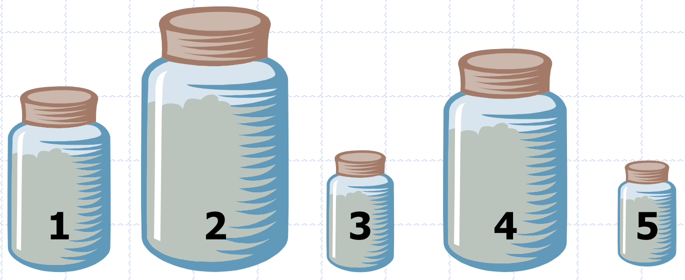
- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**.
  - In this case, we let x_i denote the amount we take of item i
  - Objective: maximize $$\sum_{i \in S} b_i (x_i / w_i)$$
  - Constraint: $$\sum_{i \in S} x_i \le W$$

Example



- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W.

Items:	1	2	3	4	5
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	\$12	\$32	\$40	\$30	\$50
Value: ($ per ml)	3	4	20	5	50

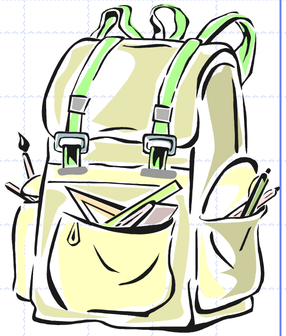


"knapsack"

10 ml

Solution:
- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

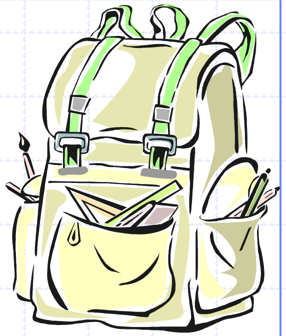
The Fractional Knapsack Algorithm



- Greedy choice: Keep taking item with highest **value** (benefit to weight ratio)
  - Since $\sum_{i \in S} b_i (x_i / w_i) = \sum_{i \in S} (b_i / w_i) x_i$
  - Run time: O(n log n). Why?
- Correctness: Suppose there is a better solution
  - there is an item i with higher value than some chosen item k (i.e., $v_k < v_i$, $x_i < w_i$, and $x_k > 0$) If we substitute some of i for k, we get a better solution
  - How much of i: $\min\{w_i - x_i, x_k\}$
  - Thus, there is no better solution than the greedy one

```
Algorithm fractionalKnapsack(S, W)
    Input: set S of items w/ benefit b_i
        and weight w_i; max. weight W
    Output: amount x_i of each item i
        to maximize benefit with
        weight at most W
    for each item i in S do
        $x_i \leftarrow 0$
        $v_i \leftarrow b_i / w_i$            {value}
    $w \leftarrow 0$                       {total weight}
    while $w < W$ do
        remove item i with highest $v_i$
        $x_i \leftarrow \min\{w_i, W - w\}$
        $w \leftarrow w + \min\{w_i, W - w\}$
```

The Fractional Knapsack Algorithm



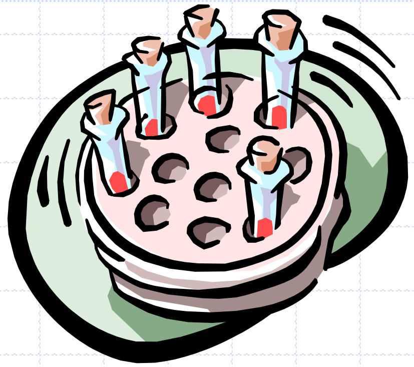
Abstract Algorithm:

```
Algorithm fractionalKnapsack(S, W)
    Input: set S of items w/ benefit $b_i$
        and weight $w_i$; max. weight $W$
    Output: amount $x_i$ of each item $i$
        to maximize benefit with
        weight at most $W$
    for each item i in S do
        $x_i \leftarrow 0$
        $v_i \leftarrow b_i / w_i$          {value}
    $w \leftarrow 0$                    {total weight}
    while $w < W$ do
        remove item $i$ with highest $v_i$
        $x_i \leftarrow \min\{w_i, W - w\}$
        $w \leftarrow w + \min\{w_i, W - w\}$
```

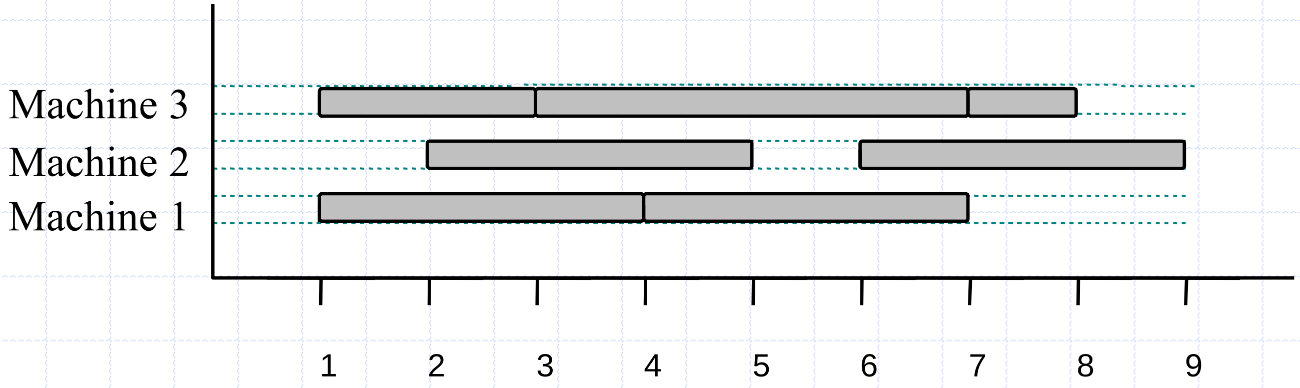
Algorithm Details:

```
Algorithm fractionalKnapsack(S, W)
    $Q \leftarrow$ new Max Priority Queue
    $x \leftarrow$ new Array of size n
    for $i \leftarrow 0$ to $S.size() - 1$ do
        $(bn, wt) \leftarrow S.elemAtRank(i)$
        $x[i] \leftarrow 0$
        $v \leftarrow bn / wt$   {benefit per unit}
        $Q.insertItem(v, (bn, wt, i))$
    $w \leftarrow 0$                {total weight}
    while $w < W$ do
        $(bn, wt, i) \leftarrow Q.removeMax()$
        $x[i] \leftarrow \min\{wt, W - w\}$
        $w \leftarrow w + x[i]$
```

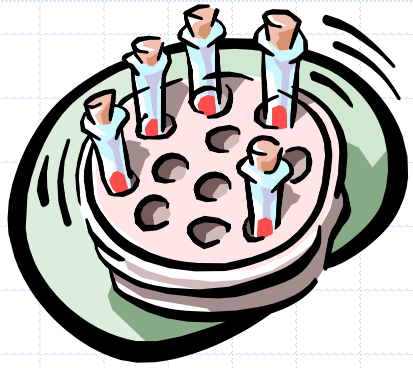
Task Scheduling Problem



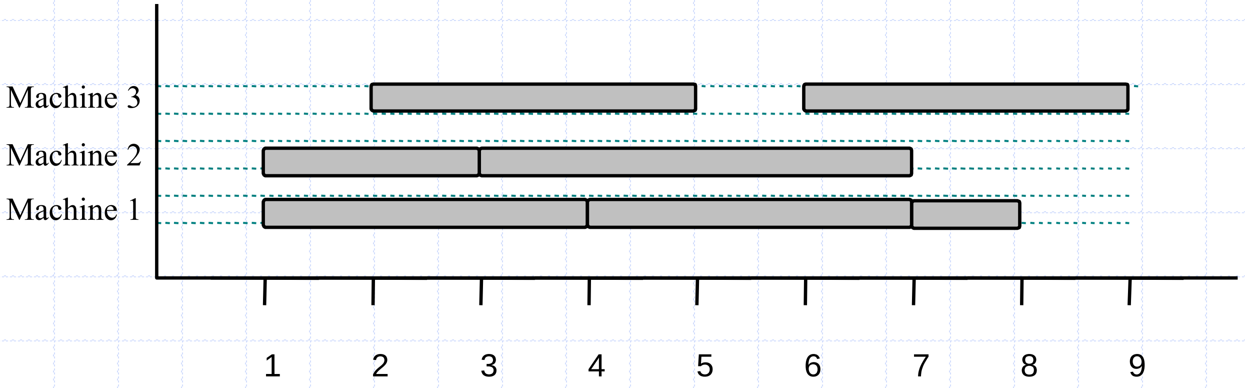
- Given: a set T of n tasks, each having:
  - A start time, s_i
  - A finish time, f_i (where $s_i < f_i$)
- Goal: Perform all the tasks using a minimum number of "machines."



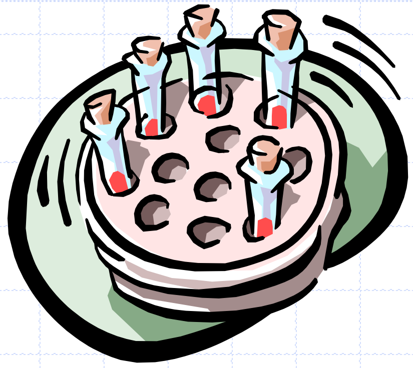
Example



- Given: a set T of n tasks, each having:
  - A start time, s_i
  - A finish time, f_i (where $s_i < f_i$)
  - [1,4], [1,3], [2,5], [3,7], [4,7], [6,9], [7,8] (ordered by start)
- Goal: Perform all tasks on min. number of machines



Task Scheduling Algorithm



- Greedy choice: consider tasks by their start time and use as few machines as possible with this order.
  - Run time: O(n log n). Why?
  - Do we need more details?
- Correctness: Suppose there is a better schedule.
  - We can use k-1 machines
  - The algorithm uses k
  - Let i be first task scheduled on machine k
  - Machine i must conflict with k-1 other tasks
  - But that means there is no non-conflicting schedule using k-1 machines

```
Algorithm taskSchedule(T)
    Input: set T of tasks w/ start time $s_i$
    and finish time $f_i$
    Output: non-conflicting schedule
    with minimum number of machines
    $m \leftarrow 0$                {no. of machines}
    while T is not empty
        remove task i w/ smallest $s_i$
        if there's a machine j for i then
            schedule i on machine j
        else
            $m \leftarrow m + 1$
            schedule i on machine m
    return schedule
```

Algorithm Details:

```
Algorithm taskSchedule(T)
    Input: set T of tasks w/ start time $s_i$ and finish time $f_i$
    Output: non-conflicting schedule F with minimum number of machines
    $m \leftarrow 0$                    {no. of machines}
    $Q \leftarrow$ new heap based priority queue {for scheduling tasks}
    $M \leftarrow$ new heap based priority queue {for allocating machines}
    for each task (s, f) in T.elements()  do
        Q.insertItem(s, (s, f))
    while ! Q.isEmpty()  do
        $(s, f) \leftarrow$ Q.removeMin()     {task with earliest start is scheduled next}
        if  $M.size() > 0 \wedge M.minKey() \leq s$ then {is there a machine for task (s, f)}
            $j \leftarrow$ M.removeMin()   {schedule on machine j}
            M.insertItem(f, j)  {indicate machine j is in use until time f }
        else
            $m \leftarrow m + 1$       {allocate on another machine j}
            M.insertItem(f, m)     {indicate machine m is in use until time f }
    return  m
```

Algorithm Details (version 2):

```
Algorithm taskSchedule(T)
    Input: set T of tasks w/ start time $s_i$ and finish time $f_i$
    Output: non-conflicting schedule F with minimum number of machines
    $m \leftarrow 0$                    {no. of machines}
    Sort T by starting time {for scheduling tasks}
    $M \leftarrow$ new heap based priority queue {for allocating machines}
    for each task (s, f) in T .elements() do
        if  $M.size() > 0 \wedge M.minKey() \leq s$ then {is there a machine for task (s, f)}
            $j \leftarrow M.removeMin()$   {schedule on machine j}
            M.insertItem(f, j)  {indicate machine j is in use until time f }
        else
            $m \leftarrow m + 1$       {allocate on another machine j}
            M.insertItem(f, m)     {indicate machine m is in use until time f }
    return  m
```

What if we need the actual schedule?

Algorithm Details (version 3):

```
Algorithm taskSchedule(T)
    Input: set T of tasks w/ start time $s_i$ and finish time $f_i$
    Output: non-conflicting schedule F with minimum number of machines
    $m \leftarrow 0$                {no. of machines}
    Sort T by starting time {for scheduling tasks}
    $M \leftarrow$ new heap based priority queue {for allocating machines}
    $F \leftarrow$ new Sequence {for the final schedule}
    for each task (s, f, tid) in T  do
        if  M.size() > 0 $\wedge$ M.minKey() $\leq$ s then {is there a machine for task (s, f)}
            $j \leftarrow$ M.removeMin()   {schedule on machine j}
            F.insertLast( ((s, f, tid), j) ) {schedule task (s, f, tid) on machine j}
            M.insertItem(f, j)  {indicate machine j is in use until time f }
        else
            $m \leftarrow m + 1$       {allocate on another machine j}
            F.insertLast( ((s, f, tid), m) ) {schedule task (s, f, tid) on machine m}
            M.insertItem(f, m)     {indicate machine m is in use until time f }
    return (m, F)
```

Main Point

1. Greedy algorithms make locally optimal choices at each step in the hope that these choices will produce the globally optimal solution. However, not all optimization problems are suitable for this approach.
   Science of Consciousness: “Established in Being perform action” means that each of us would spontaneously make optimal choices.

Important Techniques for Design of Efficient Algorithms

- Divide-and-Conquer
- Prune-and-Search
- Greedy Algorithms
  - Applies primarily to optimization problems
- Dynamic Programming
  - Also applies primarily to optimization problems

Dynamic Programming

- Typically applies to optimization problems with the goal of an optimal solution through a sequence of choices
- Effective when a specific subproblem may arise from more than one partial set of choices
- Key technique is to store solutions to subproblems in case they reappear

Motivation

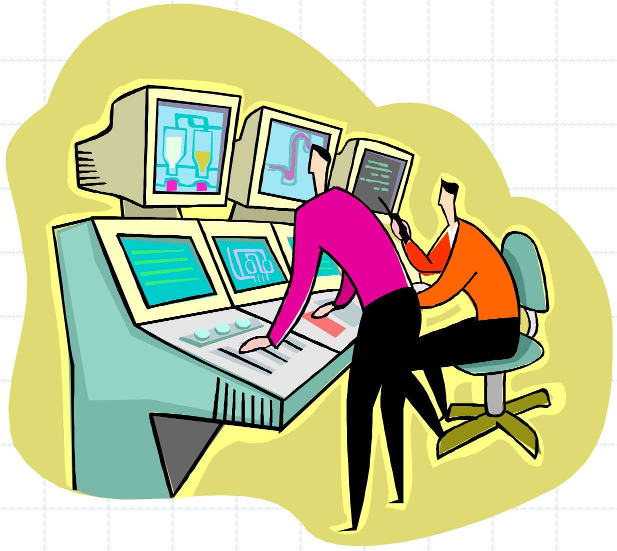
- All computational problems can be viewed as a search for a solution
- Suppose we wish to find the best way of doing something
- Often the number of ways of doing that "something" is exponential
  - i.e., the search space is exponential in size
- So a brute force search is infeasible, except on the smallest problems
- Dynamic programming exploits overlapping subproblem solutions to make the infeasible feasible

Important Note

- Each solution has a value
- Our goal is to find a solution with the optimal value
- However, there may be several solutions with the optimal value
- So our solution will be one optimal solution,
  - not the optimal solution nor all optimal solutions
  - otherwise we would have to search the entire search space (which is infeasible in general)

Outline and Reading

- The General Technique (§5.3.2)
- 0-1 Knapsack Problem (§5.3.3)
- Longest Common Subsequence (§9.4) (tomorrow)

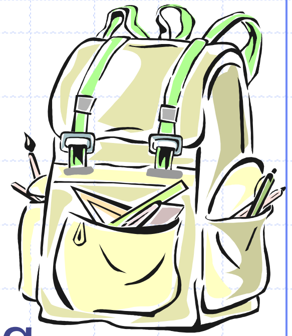


Dynamic Programming

- Top down algorithm design is natural and powerful
  - Plan in general first, then fill in the details
  - Highly complex problems can be solved by breaking them down into smaller instances of the same problem (e.g., divide-and-conquer)
- The results for small subproblems are stored and looked up, rather than recomputed
- Could transform an exponential time algorithm into a polynomial time algorithm
- Well suited to problems in which a recursive algorithm would solve many of the subproblems over and over again
- Best understood through examples

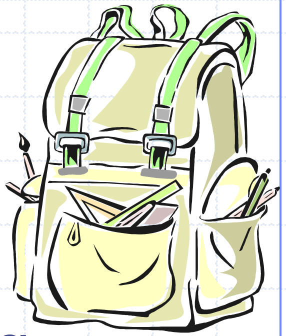
The 0/1 Knapsack Problem

The 0/1 Knapsack Problem (§5.3.3)



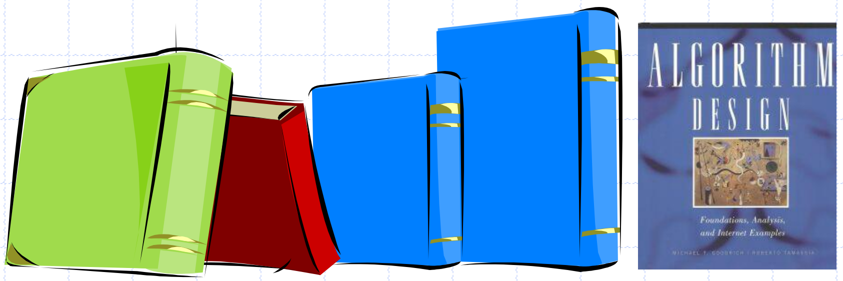
- Given: A set S of n items, with each item i having
  - w_i - a positive weight
  - b_i - a positive benefit
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are **not** allowed to take fractional amounts, then this is the **0/1 knapsack problem**.
  - In this case, we let T denote the set of items we take
  - Objective: maximize $$\sum_{i \in T} b_i$$
  - Constraint: $$\sum_{i \in T} w_i \leq W$$

Example



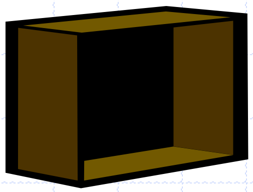
- Given: A set S of n items, with each item i having
  - b_i - a positive “benefit”
  - w_i - a positive “weight”
- Goal: Choose items with maximum total benefit but with weight at most W.

Items:



	1	2	3	4	5
Weight:	5 in	2 in	2 in	6 in	2 in
Benefit:	$20	$3	$7	$26	$80
	4	1,5	3,5	4,3	40

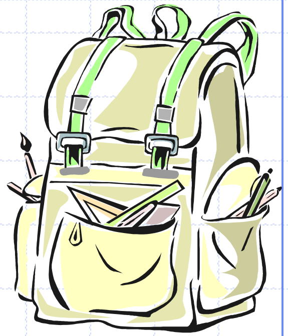
“knapsack”



box of width 9 in

Solution:
- item 5 ($80, 2 in)
- item 1 ($20, 5in)
- item 3 ($7, 2in)

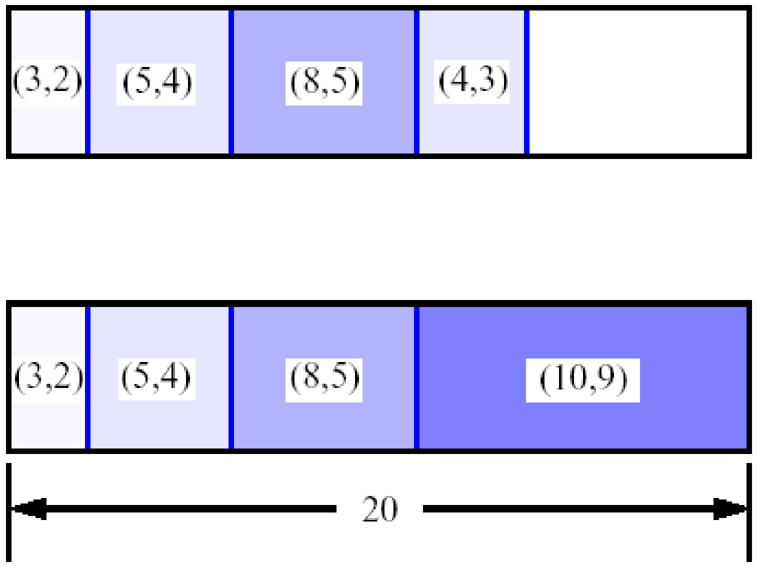
A 0/1 Knapsack Algorithm, First Attempt



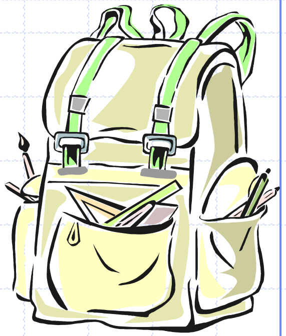
- S_k: Set of items numbered 1 to k.
- Define B[k] = best selection from S_k.
- Problem: does not have subproblem optimality:
  - Consider set S={(3,2),(5,4),(8,5),(4,3),(10,9)} of (benefit, weight) pairs and total weight W = 20

Best for S_4:

Best for S_5:



A 0/1 Knapsack Algorithm, Second Attempt



- S_k: Set of items numbered 1 to k.
- Define B[k,w] to be the best selection from S_k with weight at most w,
  - Note that the kth pair in S_k is (b_k, w_k)
- Good news: this does have subproblem optimality.

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- I.e., the best subset of S_k with weight at most w is either
  - the best subset of S_{k-1} with weight at most w or
  - the best subset of S_{k-1} with weight at most $w-w_k$ plus item k which is (b_k, w_k)

Example: 0/1 Knapsack Algorithm

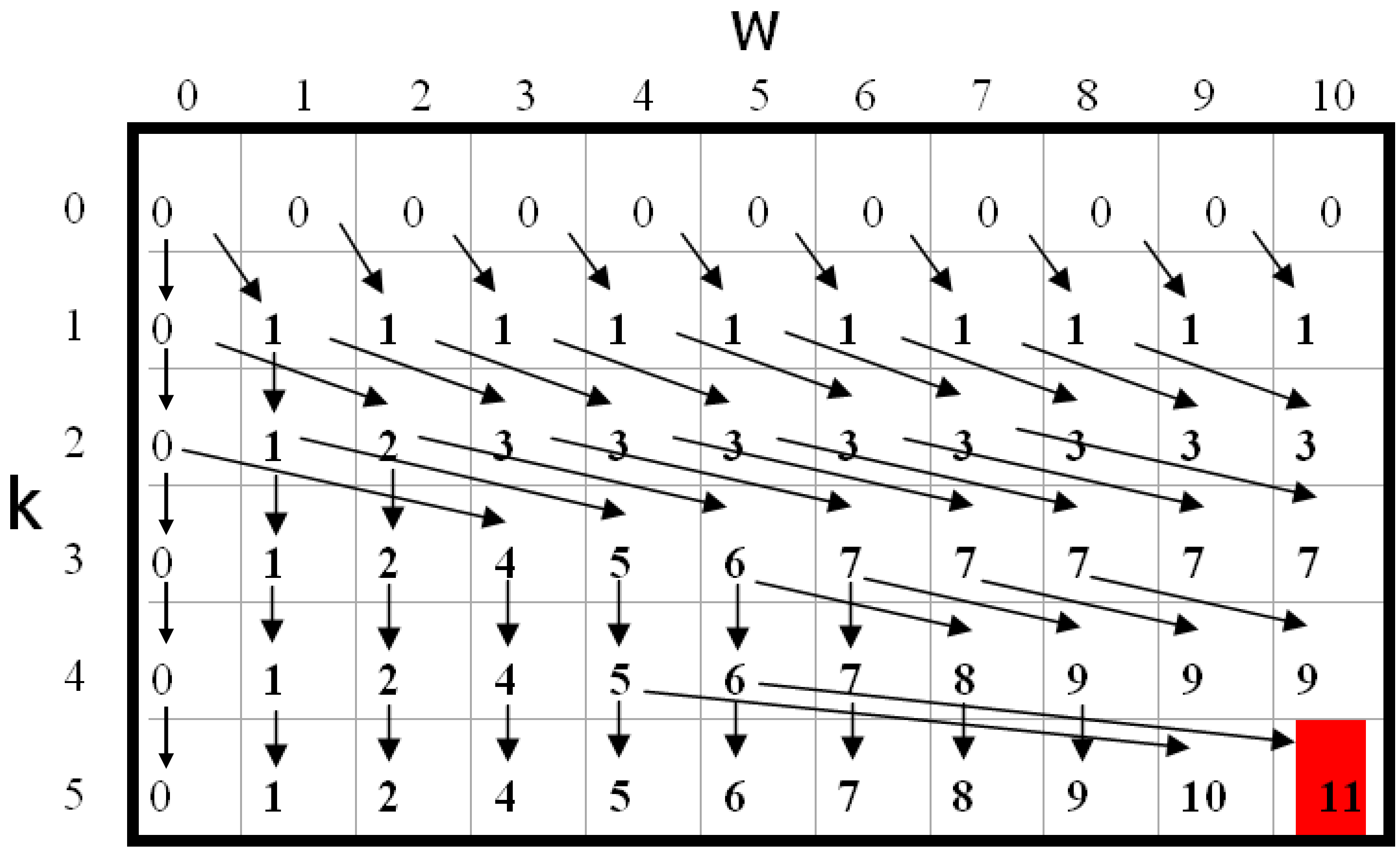
- Consider the set of (benefit, weight) pairs

  S={(1,1),(2,2),(4,3),(2,2),(5,5)}

- Total weight W = 10

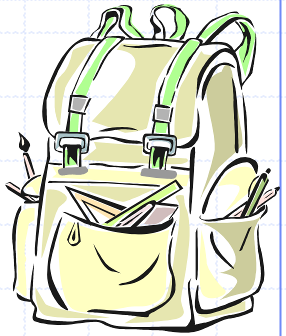
0/1 Knapsack Algorithm

- Example: set S={(1,1),(2,2),(4,3),(2,2),(5,5)} of (benefit, weight) pairs and total weight W = 10



k \ w	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1
2	0	1	2	3	3	3	3	3	3	3	3
3	0	1	2	4	5	6	7	7	7	7	7
4	0	1	2	4	5	6	7	8	9	9	9
5	0	1	2	4	5	6	7	8	9	10	11

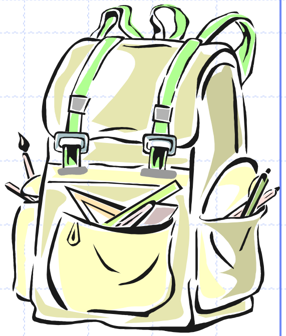
0/1 Knapsack Algorithm



$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- Since B[k,w] is defined in terms of B[k–1,*], we can use two arrays instead of a matrix

0/1 Knapsack Algorithm



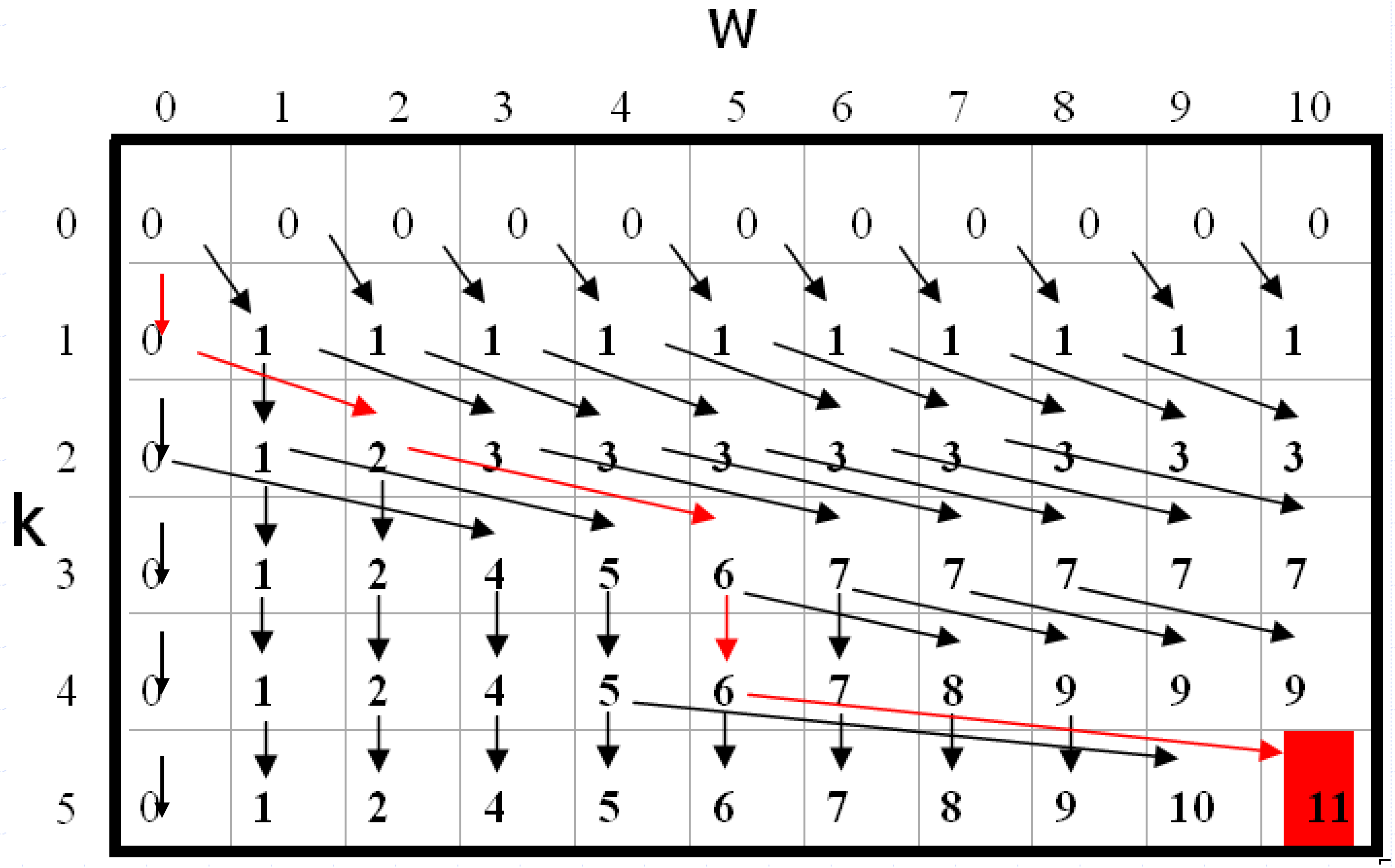
$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- Recall the definition of B[k,w]
- Since B[k,w] is defined in terms of B[k–1,*], we can use two arrays of instead of a matrix
- Running time: O(nW).
- Not a polynomial-time algorithm since W may be large
- This is a pseudo-polynomial time algorithm

```
Algorithm 01Knapsack(S, W):
    Input: set S of n items with benefit $b_i$
           and weight $w_i$; maximum weight W
    Output: benefit of best subset of S with
            weight at most W
    let A and B be arrays of length W + 1
    for $w \leftarrow 0$ to W do
        $B[w] \leftarrow 0$
    for $k \leftarrow 1$ to n do
        copy array B into array A
        $(b_k, w_k) \leftarrow$ S.elemAtRank(k-1)
        for $w \leftarrow w_k$ to W do
            if $A[w-w_k] + b_k > A[w]$ then
                $B[w] \leftarrow A[w-w_k] + b_k$
    return B[W]
```

0/1 Knapsack Algorithm

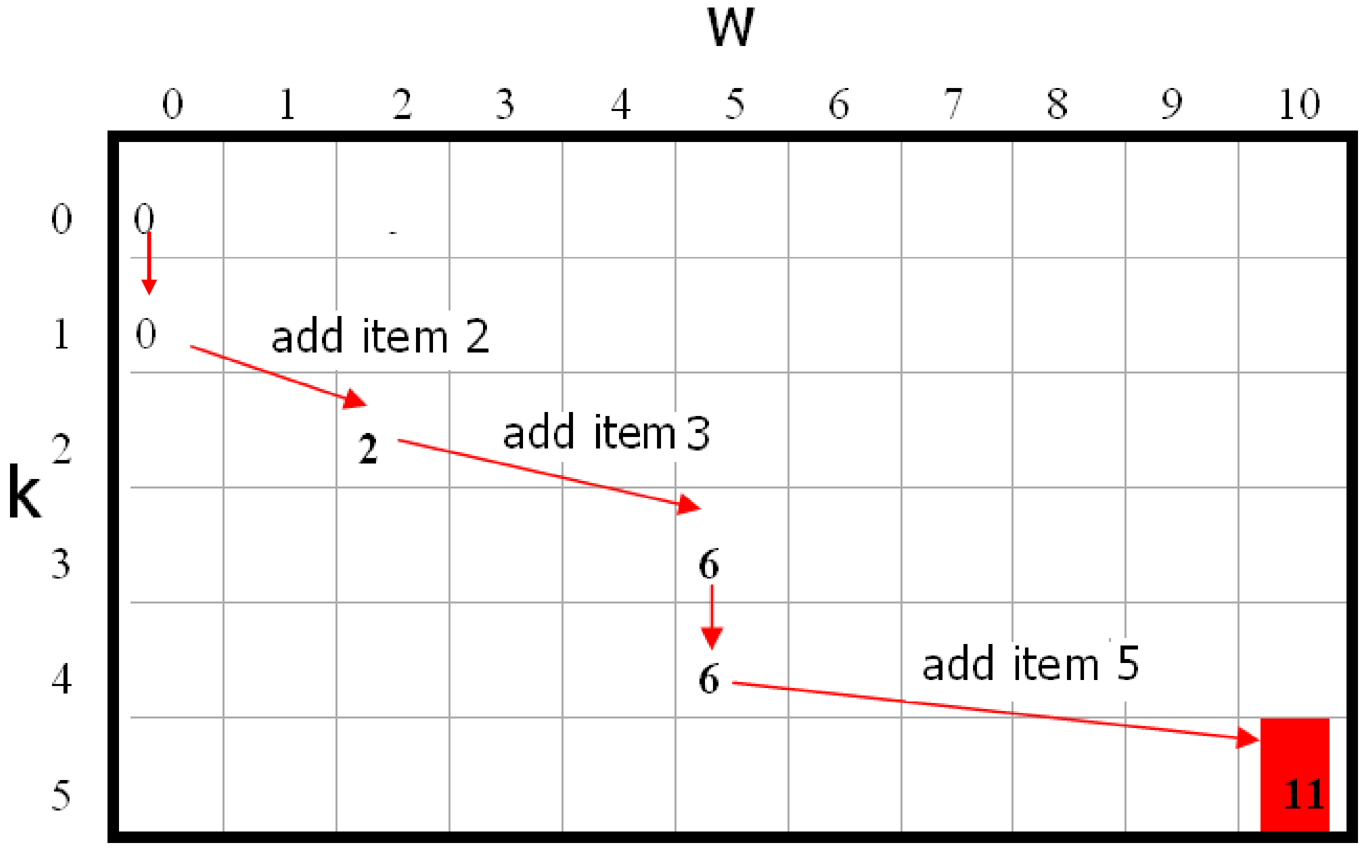
- Trace back to find the items from S={(1,1),(2,2),(4,3),(2,2),(5,5)}



	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1
2	0	1	2	3	3	3	3	3	3	3	3
3	0	1	2	4	5	6	7	7	7	7	7
4	0	1	2	4	5	6	7	8	9	9	9
5	0	1	2	4	5	6	7	8	9	10	11

0/1 Knapsack Algorithm

- Each diagonal arrow corresponds to adding one item into the bag
- Pick items 2,3,5
- {(2,2),(4,3),(5,5)} are what you will take away



Main Point

2. A dynamic programming algorithm divides a problem into subproblems, then solves each subproblem just once and saves the solution in a table to avoid having to repeat that calculation. Dynamic programming is typically applied to optimization problems to reduce the time required from exponential to polynomial time.

   Science of Consciousness: Pure intelligence governs the activities of the universe in accord with the law of least action. When we infuse pure intelligence into our awareness, our actions become more and more optimal.

Example: 0/1 Knapsack Algorithm

- Consider the set of (benefit, weight) pairs

  S={(2,1),(3,2),(4,3),(2,2),(7,5)}

- Total weight W = 10

- Solve this for homework

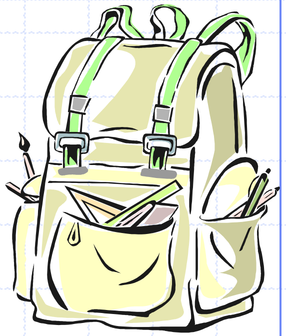
Dynamic Programming The General Technique

- Simple subproblems:
  - Must be some way of breaking the global problem into subproblems, each having similar structure to the original
  - Need a simple way of keeping track of solutions to subproblems with just a few indices, like i, j, k, etc.
- Subproblem optimality:
  - Optimal solutions cannot contain suboptimal subproblem solutions
  - Should have a relatively simple combining operation
- Subproblem overlap:
  - This is where the computing time is reduced

Basis of a Dynamic-Programming Solution

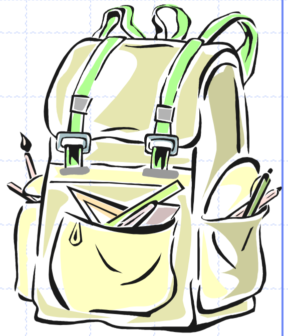
- Five steps
  1. Characterize the structure of a solution
  2. Recursively define the value of a solution in terms of solutions to subproblems
  3. Locate subproblem overlap
  4. Store overlapping subproblem solutions for later retrieval
  5. Construct an optimal solution from the computed information gathered during steps 3 and 4

Recursive Equations for 0/1 Knapsack Algorithm



$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

Recursive Equations for 0/1 Knapsack Algorithm



$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\, B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

```
Algorithm 0-1-Knapsack(S, k, w)
    if k=0 \/ w=0 then
            return 0
    else
            e ← S.elemAtRank(k-1)  // retrieve item k from S
            bk ← e.benefit()
            wk ← e.weight()
            if wk > w then // item k does not fit in knapsack of size w
                return 0-1-Knapsack(S, k-1, w)
            else
                return MAX(0-1-Knapsack(S, k-1, w),
                           0-1-Knapsack(S, k-1, w-wk) + bk)
```

Step 5

- Can be omitted if only the value of an optimal solution is required
- When step 5 is required, sometimes we need to maintain additional information during the computation in step 4 to ease construction of an optimal solution

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 (as before), but the control structure for filling in the table occurs during normal execution of the recursive algorithm
- Advantages
  - The algorithm does not have to be transformed into an iterative one
  - Often offers the same (or better) efficiency as the usual dynamic-programming approach

Example: Calculate Fibonacci Numbers

Mathematical definition:

$$fib(0) = 0$$

$$fib(1) = 1$$

$$fib(n) = fib(n-2) + fib(n-1) \qquad \text{if } n > 1$$

Fibonacci solution1

```
Algorithm Fib( n ):
   Input: integer $n \geq 0$
   Output: the n-th Fibonacci number
    if n=0 then
        return 0
    else if n=1 then
        return 1
    else
        return Fib(n - 2) + Fib( n - 1)
```

Fibonacci Solution 2

```
Algorithm Fib(n):
  Input: integer n ≥ 0
  Output: the n-th Fibonacci number

    F ← new array of size n+1
    for i ← 0 to n do
        F[i] ← -1

    return memoizedFib(n, F)
```

```
Algorithm memoizedFib(n, F):
  Input: integer n ≥ 0
  Output: the n-th Fibonacci number
    if F[n] < 0 then  // If Fib(n) has not been computed?
        if n=0 then
            F[n] ← 0
        else if n=1 then
            F[n] ← 1
        else
            F[n] ← memoizedFib(n-2, F) + memoizedFib(n-1, F)

    return F[n]
```

Summary: Memoized Recursive Algorithms

- A memoized recursive algorithm maintains a table with an entry for the solution to each subproblem (same as before)
- Each table entry initially contains a special value to indicate that the entry has yet to be filled in
- When the subproblem is first encountered, its solution is computed and stored in the table
- Subsequently, the value is looked up rather than computed

Exercises

1. Memoize the algorithm to compute Fibonacci numbers using two integer parameters instead of table F
2. Memoize the algorithm to compute Fibonacci numbers using one integer parameter

Main Point

3. Memoization is a technique for doing dynamic programming recursively. It often has the same benefits as regular dynamic programming without requiring major changes to the original more natural recursive algorithm.

   Science of Consciousness: The TM program provides natural, effortless techniques for removing stress and bringing out spontaneous right action.

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.