# Dynamic Programming Techniques

1. Define the sub-problems, count the number of sub-problems
2. Guess part of a solution, count number of choices of the guess
   1. Shortest path (u,v): the last edge coming into v, # of edges coming into v
3. Relate sub problem solutions with recursion. (compute time per sub-problem)
   1. Define the **subtasks recursively** (express larger subtask in terms of smaller ones)
4. Recurse/memorize or build a DP table.
5. Find the **right order of solving the subtasks** but do not solve them recursively.
6. Solve the original problem.

# Steps:

1. Find recursive solution first.
   1. Either the solution contains say j, then it is OPT(j)+(some value of j)
   2. Or the solution does not contain j, then it is OPT(j-1)
   3. Or, there is a range (i, j) such that the solution is of the form of
      1. OPT(i,j) = OPT(i, j-1)

OPT(i, k-1) + weight of job j?, OPT(k+1, j)

Try to find the sub problems in it as described in the code

1. Memoize the recursive solution
2. Then build a table.
3. What type of array is used? E.g. making change uses an array where the index is an amount and the value at index “array[idx]” is the # of coins. This means the array indices and the array contents can have their own meaning;

# How to break up the problem:

* Suffix:
  + Pretty Printing: start with the last line n (words k on last line), then solve for n-1 lines.
* Prefix

|  |  |  |
| --- | --- | --- |
| Steps | Fibonacci | Shortest Path |
| Define Sub-problems, count number of sub problems | F(k) for 1 <= k <= n | SP(s,v) for v in V,  0 <= k <= V  Min s->v path using at most k edges. |
| Count # of sub problems | N | V^2 (every vertex is connected to every other vertex). |
| Guess | Nothing | Edge into ‘v’, in degree of v |
| # of Guesses | 1 | Indegree (v) + 1 |
| Recurrence | F(k) = F(k-1) + F(k-2) | SP(s,v) = min { SPk-1(s,v) + W(u,v) |
| Topological Order | For k = 1,2…n | For k = 0, 1….V-1 |
| Total time | Theta(n) | Theta(VE) + V2 |
| Original Problem | F(n) | SPv-1(s,v) for all v in V |

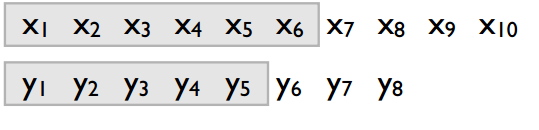
Example: Find Fib(n)

1. Divide the problem into subtasks : Fib(i) where i is < n
2. Define the subtasks recursively (express larger subtasks in terms of smaller subtasks) (e.g. Fib(i) = Fib(i-1) + Fib(i-2).
   1. Use memorization for recursive tasks.
3. Find the right order for solving the subtasks: here (i = 1, 2, 3, …n)

Reference: <http://cseweb.ucsd.edu/classes/wi12/cse202-a/lecture7-final.pdf>

Properly Worked Examples:

# Common DP Subtasks:

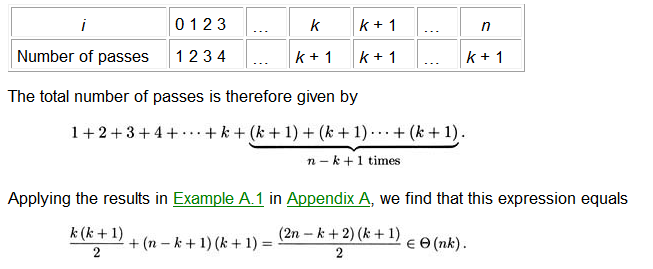
1. Case 1:
   1. **Input**: x1, x2, x3…xn.
   2. **Subproblems**: x1, x2, x3….xi , where i < n
   3. **Example**: 
2. **Case 2:**
   1. **Input: (x1, x2, x3, x4….xn), (y1, y2, y3…ym)**
   2. **Sub problems: (x1, x2….xi), (y1, y2….yj)**
   3. **Example: **
   4. **Order will be a table where rows are from (i = 1 to n), columns are from (j = 1 to m)**
3. Case 3:
   1. Input: x1, x2, x3, …xn
   2. Subproblem: xi….xj
   3. Example: 
4. Case 4:
   1. Input a rooted tree
   2. Subproblem: A subtree

# Weighted Interval Scheduling Problem

Given jobs 1, 2…. N, where there is Start(i) and End(i) and a Weight(i), find a schedule with maximum weight, e.g.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

# Binomial Coefficient Example

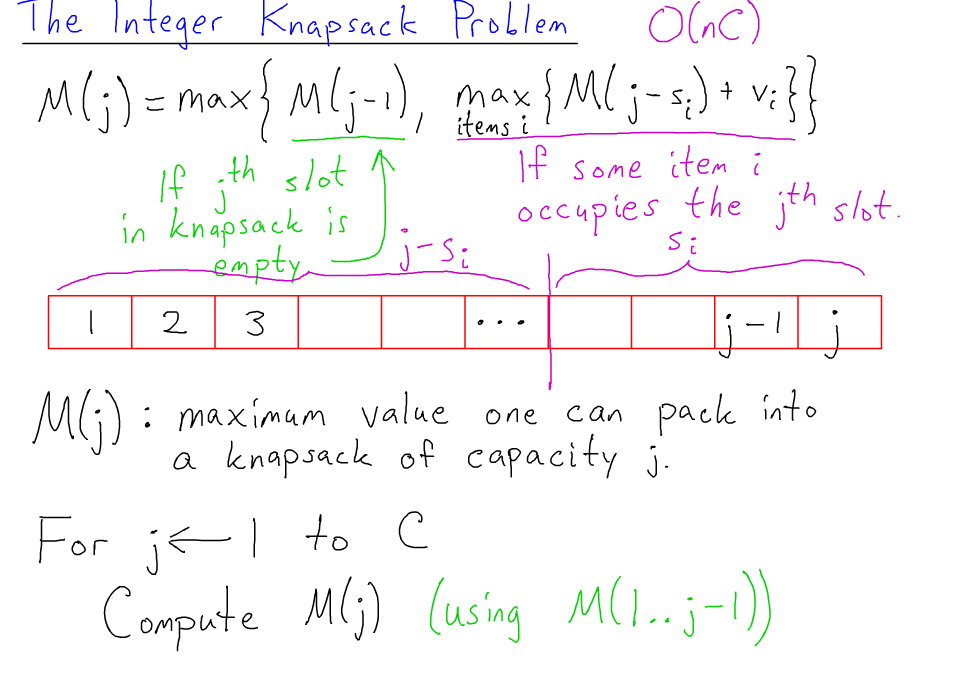
1. Counting Example (Binomial Coefficient Example)
   1. N-k+1 because counting is 0 based.

# Integer Knapsack Problem, Duplicates allowed

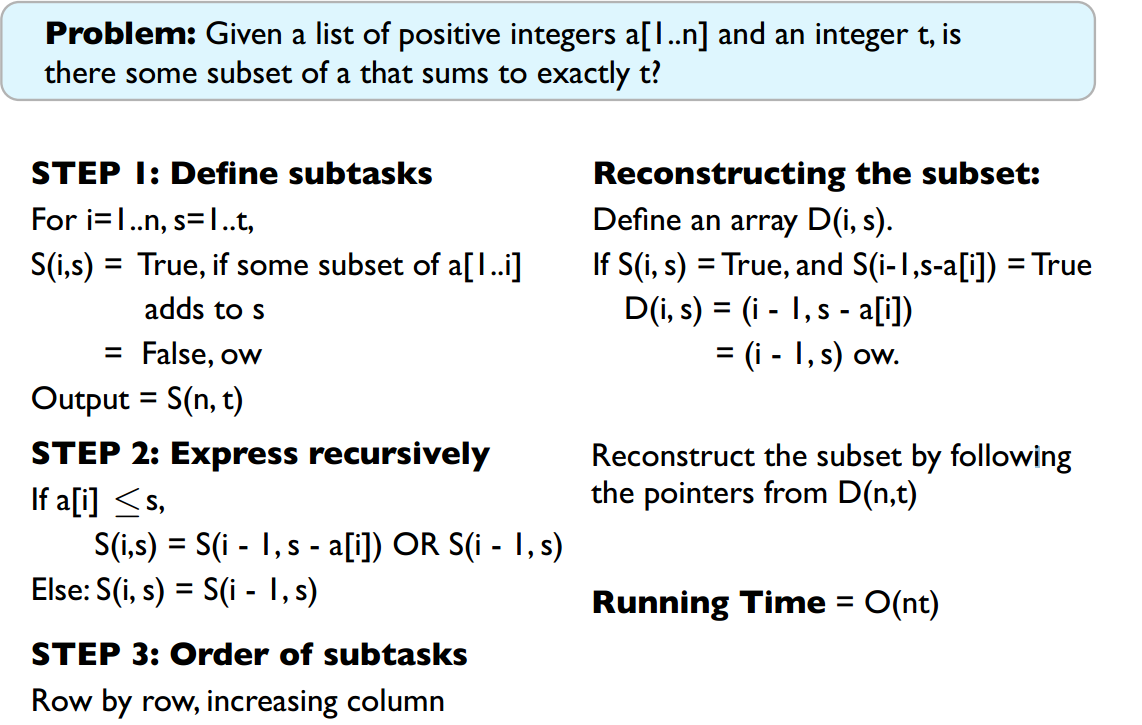
Worked Example:

<http://www.8bitavenue.com/2011/12/dynamic-programming-integer-knapsack/>

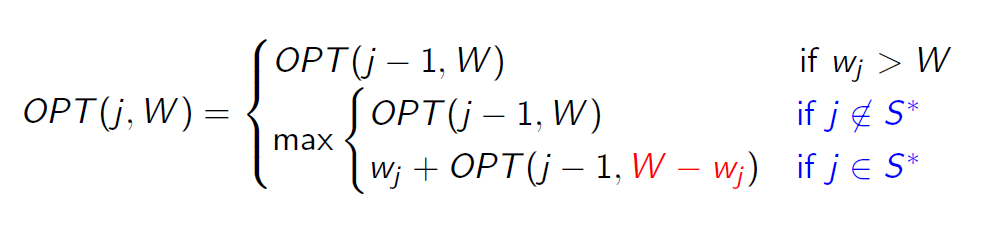
<http://people.cs.clemson.edu/~bcdean/dp_practice/dp_0.swf>

* The knapsack has capacity C, so we want to find M( C)
* To reduce the problem, consider a knapsack of capacity M(j), there are j slots which are filled by items.
* M[j] can either have an item i or not have item i, that is:
  + M(j) = M(j-1) if the jth slot does not have item i. This means M(j-1) is optimal.
  + M(j) = max (M(j-si)+vi)
    - for all i in n items available
    - and si <= j
  + NOTE: j will vary from 1 to j as there are j slots. Thus C will vary from 1 to C as there are C slots in the knapsack
* 

# Subset Sum Problem



**Recurrance**:



References:

<http://cseweb.ucsd.edu/classes/wi12/cse202-a/lecture7-final.pdf>

# Unbounded Knapsack problem

* Knapsack problem involves weights W{i} and values v{i} and a capacity C. The objective is to maximize the value for the given capacity.
* Each of the items ‘i’ can be selected multiple times.
* The DAG of knapsack gives the longest path in a dag.
* Let T(W) be the maximum value of a knapsack of capacity W
  + Subproblem T(W) = max { v[i] + T(W-W[i])} for all ‘i’ where W >= W[i].
    - NOTE: here the last item ‘i’ can be any of the items from W{i} as any item can be selected multiple times.
  + This is similar to making change problem.
  + Remember the memorized version has an array that has its index as the capacity, and the value of the array at index as the total value for that capacity of index.
* C:\Users\kg\Desktop\interview\_files\dynamic\_programming\WorkedExamples\knapsack2.htm

# 0-1 Knapsack

1. Aa
2. C:\Users\kg\Desktop\interviews\dynamic\_programming\subsetsum-knapsack-difference.pdf
3. [C:\Users\kg\Desktop\interview\_files\dynamic\_programming\WorkedExamples\knapsack3](http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/DynProg/knapsack3.html)
4. Either an item ‘i’ is picked or it is not picked.
5. T(W, j) = max { T(W-w[j], j-1) + v[j], T(W, j-1)}
   1. Either item j is picked, then T(W-w[j],j-1)+v[j]
   2. Item at j is not picked: T(W,j-1)
6. There is only one item picked here at each iteration.

# Fractional Knapsack

C:\Users\kg\Desktop\interviews\dynamic\_programming\subsetsum-knapsack-difference.pdf

# Longest increasing subsequence

### Problem Definition: Given a sequence of values, say 14, 2, 5, 19, 44, 24 52, find the largest increasing subsequence in this, e.g. it can be 2,5,19,44, 52

L(n) = max(1+L(1…i)) where I < n, A[i] < A[n]

Or

L(i) = max(L(i) , 1+L(1…i-1)) where A[i-1] < A[i] and 0<i<n

Or

L(n) = 1+max(L(1), L(2), L(3), L(4), … L(n-1)) where

A[n] > A[n-1]

L(2), L(3)… L(n-1) are also recursive.

<https://courses.engr.illinois.edu/cs473/fa2011/lec/08_notes.pdf>

<http://www.8bitavenue.com/2011/11/dynamic-programming-longest-increasing-sub-sequence/>

# Edit Distance Problem

For X(i) and Y(j), the minimum edit distance is one of

1. Insert at i+1, then : Cost(x(i), y(j-1))+1
2. Delete at x(i): Cost(x(i-1), y(j))+1
3. Replace (if x(i) != y(j)): x(i-1), y(j-1)+1
4. If (x(i) == y(j)): x(i-1), y(j-1)
5. If x is empty, then there are j insertions in x or j deletions in y.
6. If y is empty, there are i deletions in x or i insertions in y

* <http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/DynProg/edit-distance.html>
  + C:\Users\kg\Desktop\interview\_files\dynamic\_programming\WorkedExamples\edit-distance-workedexample.html
* Vazirani book

# Longest Common Subsequence

* C:\Users\kg\Desktop\interview\_files\dynamic\_programming\WorkedExamples \LongestComonSubsequence

# Making Change

Given a set of coins, e.g. Coins = {1, 3, 5, 9, 15}, make change for an amount (say 66) for which number of coins is minimum.

**Technique**:

* Always go by the last prefix, in this case, prefix is the last coin to be subtracted from the total, and it can be any of the coins.
* T(n) = 1+min{T(n-C[i])} for all i in array Coins{1,3,5,9,15}. Where C[i] <= n
* T(6) = 1+ min(T(5), T(3), T(1))
* Use a memo array where each element in the array is an amount, e.g. amt[n+1] where amt[i] is # of coins for amount i.
* Solution is in amt[n]
* C:\Users\kg\Desktop\interview\_files\dynamic\_programming\WorkedExamples\money-change

# Independent Set Problem

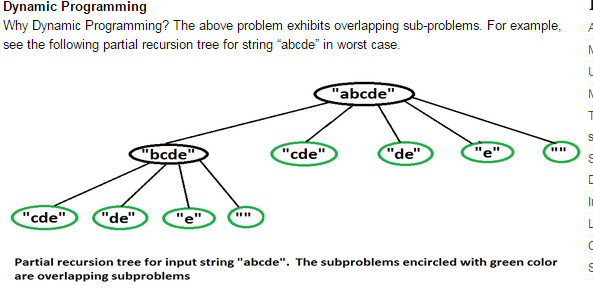
<http://cseweb.ucsd.edu/classes/wi12/cse202-a/lecture7-final.pdf>

# String Reconstruction

<http://cseweb.ucsd.edu/classes/wi12/cse202-a/lecture6-final.pdf>

<http://www.fas.harvard.edu/~libcs124/CS/lec9.pdf>

Hint:



# Shortest/Longest path in a DAG

d(v) = F { d(u) } + 1, where F is min or max

where (u,v) are edges incident to v

* Do topological sort of all vertices
* For shortest path from vertex v1 to vk, set d(v1) = 0
* Call relax on all vertices in the topological sorted stack.
* Use parent pointer to print paths.
* <http://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/>
* C:\Users\kg\Desktop\interview\_files\graphs\longest-path-in-dag.pdf

# Shortest Path Problem (

**Negative Weight Cycles Problem**:

# Links

* <https://www.cs.cmu.edu/~ckingsf/class/02713-s13/lectures/lec15-subsetsum.pdf>
* <http://www.fas.harvard.edu/~libcs124/CS/lec9.pdf>
* <http://cseweb.ucsd.edu/classes/wi12/cse282-a/>
* C:\Users\kg\Desktop\interviews\dynamic\_programming\WorkedExamples
* <http://cseweb.ucsd.edu/classes/wi12/cse202-a/lecture6-final.pdf>
* <http://cseweb.ucsd.edu/classes/wi12/cse202-a/lecture7-final.pdf>
* http://faculty.kfupm.edu.sa/ics/darwish/stuff/ics353handouts/Ch4Ch5.pdf