**Topic 1 : Introduction to Dynamic Programming**

**Dynamic Programming** is an algorithmic approach to solve some complex problems easily and save time and number of comparisons by storing the results of past computations.

*We can also see Dynamic Programming as dividing a particular problem into subproblems and then storing the result of these subproblems to calculate the result of the actual problem.*

Consider the problem of finding ***the N-th Fibonacci number***.  
  
We know that n-th fibonacci number fib(n) can be defined as:

fib(n) = fib(n-1) + fib(n-2), where n >= 2.  
  
and,  
  
fib(0) = 0   
fib(1) = 1

We can see that the above function fib() to find the nth fibonacci number is divided into two subproblems fib(n-1) and fib(n-2) each one of which will be further divided into subproblems and so on.  
  
The first few Fibonacci numbers are:

*1, 1, 2, 3, 5, 8, 13, 21, 34,*........

The recursive program to find N-th Fibonacci number is shown below:



1

2

3

4

5

6

7

8

9

int fib(int n)

{

if (n <= 1)

return n;

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

}

Below is the recursion tree for the recursive solution to find the N-th Fibonacci number:

fib(5)  
 / \  
 fib(4) fib(3)  
 / \ / \  
 fib(3) fib(2) fib(2) fib(1)  
 / \ / \ / \  
 fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)  
 / \  
fib(1) fib(0)

We can see that the function fib(3) is being called 2 times. If we would have stored the value of fib(3), then instead of computing it again, we could have reused the old stored value.  
  
The **time complexity** of the *recursive solution* is **exponential**. However, we can improve the time complexity by using Dynamic Programming approach and storing the results of the subproblems as shown below:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

int fib(int n)

{

// Declare an array to store Fibonacci numbers

int f[n+2]; // 1 extra to handle case, n = 0

int i;

// 0th and 1st number of the series are 0 and 1

f[0] = 0;

f[1] = 1;

for (i = 2; i <= n; i++)

{

// Add the previous 2 numbers in the series

// and store it

f[i] = f[i-1] + f[i-2];

}

return f[n];

}

The **time complexity** of the above solution is ***linear***.

**Topic 2 : Properties of Dynamic Programming Problem**

There are two main properties of any problem which identifies a problem that it can be solved using the dynamic programming approach:

1. Overlapping Subproblem Property
2. Optimal Substructure Property

Let us look at each one of these properties in details:

1. **Overlapping Subproblems:** Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of the same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don’t have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn’t have common subproblems. If we take an example of following the recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.



1

2

3

4

5

6

7

8

9

/\* simple recursive program for Fibonacci numbers \*/

int fib(int n)

{

if ( n <= 1 )

return n;

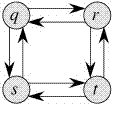
return fib(n-1) + fib(n-2);

}

Recursion tree for execution of fib(5):

fib(5)  
 / \  
 fib(4) fib(3)  
 / \ / \  
 fib(3) fib(2) fib(2) fib(1)  
 / \ / \ / \  
 fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)  
 / \  
fib(1) fib(0)

We can see that the function fib(3) is being called 2 times. If we would have stored the value of fib(3), then instead of computing it again, we could have reused the old stored value.

1. **Optimal Substructure:** A given problem has Optimal Substructure Property if an optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.  
     
   For example, the Shortest Path problem has the following optimal substructure property:  
   If a node **x** lies in the shortest path from a source node **u** to destination node **v** then the shortest path from **u to v** is combination of shortest path from **u to x** and shortest path from **x to v**. The standard All Pair Shortest Path algorithms like **Floyd–Warshall** and **Bellman-Ford** are typical examples of Dynamic Programming.  
     
   On the other hand, the Longest Path problem doesn’t have the Optimal Substructure property. Here, by Longest Path we mean the longest simple path (path without cycle) between any two nodes. Consider the following unweighted graph given in the CLRS book. There are two longest paths from q to t: **q->r->t** and **q->s->t**. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path **q->r->t** is not a combination of the longest path from q to r and longest path from r to t, because the longest path from q to r is **q->s->t->r** and the longest path from r to t is**r->q->s->t**.  
   

**Topic 3 : Overlapping Subproblems Property**

We had already discussed the basics of **Overlapping Subproblems property** of a problem that can be solved using the Dynamic Programming algorithm. Let us extend our previous example of Fibonacci Number to discuss the overlapping subproblems property in detail.



1

2

3

4

5

6

7

8

/\* simple recursive program for Fibonacci numbers \*/

int fib(int n)

{

if ( n <= 1 )

return n;

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

}

Recursion tree for execution of *fib(5)*

fib(5)

/ \

fib(4) fib(3)

/ \ / \

fib(3) fib(2) fib(2) fib(1)

/ \ / \ / \

fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)

/ \

fib(1) fib(0)

We already discussed how storing results of the subproblems can be effective in reducing the number of calculations or operations to obtain the final result. As in the above recursion tree, we can see that different values like fib(1), fib(0), fib(2) are being calculated more than once. There are two different ways to store the values so that these values can be reused:

1. Memoization (Top Down)
2. Tabulation (Bottom Up)

Let us look at each one of these in details:

1. **Memoization (Top Down):**The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need the solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.  
     
   Following is the memoized version for nth Fibonacci Number.

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

/\* C++ program for Memoized version

for nth Fibonacci number \*/

#include <bits/stdc++.h>

using namespace std;

#define NIL -1

#define MAX 100

int lookup[MAX];

/\* Function to initialize NIL

values in lookup table \*/

void \_initialize()

{

int i;

for (i = 0; i < MAX; i++)

lookup[i] = NIL;

}

/\* Function for nth Fibonacci number \*/

int fib(int n)

{

if (lookup[n] == NIL)

{

if (n <= 1)

lookup[n] = n;

else

lookup[n] = fib(n - 1) + fib(n - 2);

}

Run

Java



**Output:**

Fibonacci number is 102334155

1. **Tabulation (Bottom Up):**The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from the table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3) and so on. So literally, we are building the solutions of subproblems bottom-up.  
     
   Following is the tabulated version for nth Fibonacci Number.

C/C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

/\* C++ program for Tabulated version \*/

#include<bits/stdc++.h>

int fib(int n)

{

int f[n+1];

int i;

f[0] = 0; f[1] = 1;

for (i = 2; i <= n; i++)

f[i] = f[i-1] + f[i-2];

return f[n];

}

// Driver Code

int main ()

{

int n = 9;

printf("Fibonacci number is %d ", fib(n));

return 0;

}

Run

Java



**Output:**

Fibonacci number is 34

Both Tabulated and Memoized approaches store the solutions of subproblems. In the Memoized version, the table is filled on demand while in the Tabulated version, starting from the first entry, all entries are filled one by one. Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in the Memoized version.

**Topic 4 : Optimal Substructure Property**

A given problem has **Optimal Substructure Property** if the optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

*That is, say if a problem****x****is divided into subproblems****A****and****B****then the optimal solution of x can be obtained by summing up the optimal solutions to the subproblems A and B.*

For example, the Shortest Path problem has following optimal substructure property:  
If a node **x** lies in the shortest path from a source node **u** to destination node **v** then the shortest path from **u to v** is combination of shortest path from **u to x** and shortest path from **x to v**. The standard All Pair Shortest Path algorithms like Floyd - Warshall andBellman-Ford are typical examples of Dynamic Programming.

Let us consider a simple example of the 0**-1 Knapsack Problem**. The problem states that given values and weight associated with N items. The task is to put these items into a Knapsack of capacity **W, such** that the value of all items in the Knapsack is maximum possible. You can either include a complete element or do not include it, it is not allowed to add a fraction of an element.  
  
**For Example**:

**value[]** = {60, 100, 120}  
**weight[]** = {10, 20, 30}  
**W** = 50  
  
**Where**, *value[]* is the array containing values of elements,  
 *weight[]* is the array containing corresponding weights.  
and, *W* is the weight of Knapsack.  
  
The **answer** will be 220. We will pick the 2nd and 3rd elements   
and add them to the Knapsack for maximum value.

**Optimal Substructure**: To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.  
Therefore, the maximum value that can be obtained from N items is the max of the following two values.

1. Maximum value obtained by n-1 items and W weight (excluding nth item).
2. Value of nth item plus maximum value obtained by n-1 items and W minus the weight of the nth item (including nth item).

If the weight of the nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.  
  
**Overlapping Subproblems**: Let us first look at the recursive solution to the above problem:

// This function returns the maximum value that can   
// be put in a knapsack of capacity W   
int knapSack(int W, int weight[], int value[], int n)   
{   
 // Base Case   
 if (n == 0 || W == 0)   
 return 0;   
   
 // If the weight of the nth item is more than Knapsack   
 // capacity W, then this item cannot be included in   
 // the optimal solution   
 if (wt[n-1] > W)   
 return knapSack(W, wt, val, n-1);   
   
 // Return the maximum of two cases:   
 // (1) nth item included   
 // (2) not included   
 else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),   
 knapSack(W, wt, val, n-1)   
 );   
}

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree when the above recursive function is evaluated with the sample examples.

In the following recursion tree, K() refers to knapSack(). The two   
parameters indicated in the following recursion tree are n and W.   
The recursion tree is for following sample inputs.  
W = 2, wt[] = {1, 1, 1}, val[] = {10, 20, 30}  
  
 K(3, 2) ---------> K(n, W)  
 / \   
 / \   
 K(2, 2) K(2, 1)  
 / \ / \   
 / \ / \  
 K(1, 2) K(1, 1) K(1, 1) K(1, 0)  
 / \ / \ / \  
 / \ / \ / \  
K(0, 2) K(0, 1) K(0, 1) K(0, 0) K(0, 1) K(0, 0)  
  
Recursion tree for Knapsack capacity 2 units a  
and 3 items of 1 unit weight.

Since sub-problems are evaluated again, this problem has Overlapping Subproblems property. So the 0-1 Knapsack problem has both properties of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array K[][] in a bottom-up manner. Following is Dynamic Programming based implementation.

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// A Dynamic Programming based solution for

// 0-1 Knapsack problem

#include <bits/stdc++.h>

using namespace std;

// A utility function that returns maximum of two integers

int max(int a, int b) { return (a > b) ? a : b; }

// This function returns the maximum value that can be put

// in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

int i, w;

int K[n + 1][W + 1];

// Build table K[][] in bottom up manner

for (i = 0; i <= n; i++) {

for (w = 0; w <= W; w++) {

if (i == 0 || w == 0)

K[i][w] = 0;

else if (wt[i - 1] <= w)

K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);

else

K[i][w] = K[i - 1][w];

}

}

return K[n][W];

}

Run

Java



**Output:**

220

**Topic 5 : Solving a Dynamic Programming Problem**

**D**ynamic **P**rogramming (DP) is a technique that solves some particular type of problems in Polynomial Time. Dynamic Programming solutions are faster than exponential brute methods and can be easily proved for their correctness. Before we study how to think Dynamically for a problem, we need to learn:

1. Overlapping Subproblems
2. Optimal Substructure Property

**Steps to solve a DP**

1) Identify if it is a DP problem

2) Decide a state expression with

least parameters

3) Formulate state relationship

4) Do tabulation (or add memoization)

**Step 1 : How to classify a problem as a Dynamic Programming Problem?**

* Typically, all the problems that require to maximize or minimize certain quantity or counting problems that say to count the arrangements under certain conditions or certain probability problems can be solved by using Dynamic Programming.
* All dynamic programming problems satisfy the overlapping subproblems property and most of the classic dynamic problems also satisfy the optimal substructure property. Once, we observe these properties in a given problem, be sure that it can be solved using DP.

**Step 2 : Deciding the state** DP problems are all about state and their transition. This is the most basic step which must be done very carefully because the state transition depends on the choice of state definition you make. So, let's see what do we mean by the term "state".  
  
**State** A state can be defined as the set of parameters that can uniquely identify a certain position or standing in the given problem. This set of parameters should be as small as possible to reduce state space.  
  
For example: In our famous Knapsack Problem, we define our state by two parameters **index** and **weight** i.e DP[index][weight]. Here DP[index][weight] tells us the maximum profit it can make by taking items from range 0 to index having the capacity of sack to be weighed. Therefore, here the parameters **index** and **weight** together can uniquely identify a subproblem for the knapsack problem.  
  
So, our first step will be deciding a state for the problem after identifying that the problem is a DP problem.  
  
As we know DP is all about using calculated results to formulate the final result.  
So, our next step will be to find a relation between previous states to reach the current state.  
  
  
**Step 3 : Formulating a relation among the states** This part is the hardest part of for solving a DP problem and requires a lot of intuition, observation and practice. Let's understand it by considering a sample problem

**Given 3 numbers {1, 3, 5}, we need to tell  
the total number of ways we can form a number 'N'   
using the sum of the given three numbers.**  
(allowing repetitions and different arrangements).  
  
Total number of ways to form 6 is: 8  
1+1+1+1+1+1  
1+1+1+3  
1+1+3+1  
1+3+1+1  
3+1+1+1  
3+3  
1+5  
5+1

Let's think dynamically about this problem. So, first of all, we decide a state for the given problem. We will take a parameter n to decide the state as it can uniquely identify any subproblem. So, our state dp will look like state(n). Here, state(n) means the total number of arrangements to form n by using {1, 3, 5} as elements.  
  
Now, we need to compute state(n).  
  
**How to do it?**So here the intuition comes into action. As we can only use 1, 3 or 5 to form a given number. Let us assume that we know the result for n = 1,2,3,4,5,6 ; being termilogistic let us say we know the result for the  
state (n = 1), state (n = 2), state (n = 3) ......... state (n = 6)  
  
Now, we wish to know the result of the state (n = 7). See, we can only add 1, 3 and 5. Now we can get a sum total of 7 by the following 3 ways:  
  
**1) Adding 1 to all possible combinations of state (n = 6)** Eg : [ (1+1+1+1+1+1) + 1]  
[ (1+1+1+3) + 1]  
[ (1+1+3+1) + 1]  
[ (1+3+1+1) + 1]  
[ (3+1+1+1) + 1]  
[ (3+3) + 1]  
[ (1+5) + 1]  
[ (5+1) + 1]  
  
**2) Adding 3 to all possible combinations of state (n = 4);**  
Eg : [(1+1+1+1) + 3]  
[(1+3) + 3]  
[(3+1) + 3]  
  
**3) Adding 5 to all possible combinations of state(n = 2)** Eg : [ (1+1) + 5]  
  
Now, think carefully and satisfy yourself that the above three cases are covering all possible ways to form a sum total of 7;  
  
Therefore, we can say that result for  
state(7) = state (6) + state (4) + state (2)  
or  
state(7) = state (7-1) + state (7-3) + state (7-5)  
  
In general,  
**state(n) = state(n-1) + state(n-3) + state(n-5)**  
So, our code will look like:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

// Returns the number of arrangements to

// form 'n'

int solve(int n)

{

// base case

if (n < 0)

return 0;

if (n == 0)

return 1;

return solve(n-1) + solve(n-3) + solve(n-5);

}

The above code seems exponential as it is calculating the same state again and again. So, we just need to add a memoization.  
  
  
**Step 4 : Adding memoization or tabulation for the state** This is the easiest part of a dynamic programming solution. We just need to store the state answer so that next time that state is required, we can directly use it from our memory  
  
Adding memoization to the above code



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

// initialize to -1

int dp[MAXN];

// this function returns the number of

// arrangements to form 'n'

int solve(int n)

{

// base case

if (n < 0)

return 0;

if (n == 0)

return 1;

// checking if already calculated

if (dp[n]!=-1)

return dp[n];

// storing the result and returning

return dp[n] = solve(n-1) + solve(n-3) + solve(n-5);

}

Another way is to add tabulation and make the solution iterative.

**Topic 6 : Sample Problems on Dynamic Programming**

**Problem #1 : Binomial Coefficients**

**Description -** Following are common definition of Binomial Coefficients -

1. A binomial coefficient C(n, k) can be defined as the coefficient of X^k in the expansion of (1 + X)^n.

1. A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects; more formally, the number of k-element subsets (or k-combinations) of an n-element set.

Write a function that takes two parameters n and k and returns the value of Binomial Coefficient C(n, k). For example, your function should return 6 for n = 4 and k = 2, and it should return 10 for n = 5 and k = 2.  
  
**Optimal Substructure** The value of C(n, k) can be recursively calculated using the following standard formula for Binomial Coefficients.

C(n, k) = C(n-1, k-1) + C(n-1, k)

C(n, 0) = C(n, n) = 1

**Overlapping Subproblems** It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for n = 5 and k = 2. The function C(3, 1) is called two times. For large values of n, there will be many common subproblems.

C(5, 2)

/ \

C(4, 1) C(4, 2)

/ \ / \

C(3, 0) C(3, 1) C(3, 1) C(3, 2)

/ \ / \ / \

C(2, 0) C(2, 1) C(2, 0) C(2, 1) C(2, 1) C(2, 2)

/ \ / \ / \

C(1, 0) C(1, 1) C(1, 0) C(1, 1) C(1, 0) C(1, 1)

Since same suproblems are called again, this problem has Overlapping Subproblems property  
**Pseudo Code**

// Returns value of Binomial Coefficient C(n, k)  
int binomialCoeff(int n, int k)  
{  
 int C[n+1][k+1]  
  
 // Caculate value of Binomial Coefficient in bottom up manner  
 for (i = 0; i <= n; i++)  
 {  
 for (j = 0; j <= min(i, k); j++)  
 {  
 // Base Cases  
 if (j == 0 || j == i)  
 C[i][j] = 1  
  
 // Calculate value using previously stored values  
 else  
 C[i][j] = C[i-1][j-1] + C[i-1][j]  
 }  
 }  
 return C[n][k]  
}

**Problem #2 : Minimum number of jumps to reach end**

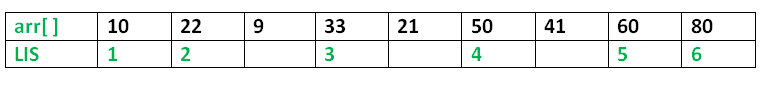
Given an array of integers where each element represents the max number of steps that can be made forward from that element. Write a function to return the minimum number of jumps to reach the end of the array (starting from the first element). If an element is 0, then cannot move through that element.  
**Example**

Input: arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}  
Output: 3 (1-> 3 -> 8 ->9)

First element is 1, so can only go to 3. Second element is 3, so can make at most 3 steps e.g. to 5 or 8 or 9.  
  
**Solution -**we build a jumps[ ] array from left to right such that jumps[ i ] indicates the minimum number of jumps needed to reach arr[ i ] from arr[ 0 ]. Finally, we return jumps[ n-1 ].  
**Pseudo Code**

// Returns minimum number of jumps   
// to reach arr[n-1] from arr[0]   
int minJumps(int arr[], int n)   
{   
 // jumps[n-1] will hold the result  
 int jumps[n]   
 if (n == 0 || arr[0] == 0)   
 return INT\_MAX;   
   
 jumps[0] = 0  
   
 // Find the minimum number of jumps to reach arr[i]   
 // from arr[0], and assign this value to jumps[i]   
 for (i = 1; i < n; i++)   
 {   
 jumps[i] = INT\_MAX  
 for (j = 0; j < i; j++)   
 {   
 if (i <= j + arr[j] && jumps[j] != INT\_MAX)   
 {   
 jumps[i] = min(jumps[i], jumps[j] + 1)  
 break  
 }   
 }   
 }   
 return jumps[n-1]  
}

**Problem #3 : Longest Increasing Subsequence**

**Description-** The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, the length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 and LIS is {10, 22, 33, 50, 60, 80}.  
  
  
More Examples:

Input : arr[] = {3, 10, 2, 1, 20}  
Output : Length of LIS = 3  
The longest increasing subsequence is 3, 10, 20  
  
Input : arr[] = {3, 2}  
Output : Length of LIS = 1  
The longest increasing subsequences are {3} and {2}  
  
Input : arr[] = {50, 3, 10, 7, 40, 80}  
Output : Length of LIS = 4  
The longest increasing subsequence is {3, 7, 40, 80}

**Optimal Substructure** Let arr[0..n-1] be the input array and L(i) be the length of the LIS ending at index i such that arr[i] is the last element of the LIS.  
Then, L(i) can be recursively written as:

L(i) = 1 + max( L(j) ) where 0 < j < i and arr[j] < arr[i]; or  
L(i) = 1, if no such j exists.

To find the LIS for a given array, we need to return max(L(i)) where 0 < i < n.  
Thus, we see the LIS problem satisfies the optimal substructure property as the main problem can be solved using solutions to subproblems.  
**Overlapping Subproblems** Considering the above implementation, following is recursion tree for an array of size 4. lis(n) gives us the length of LIS for arr[ ].

lis(4)

/ | \

lis(3) lis(2) lis(1)

/ \ /

lis(2) lis(1) lis(1)

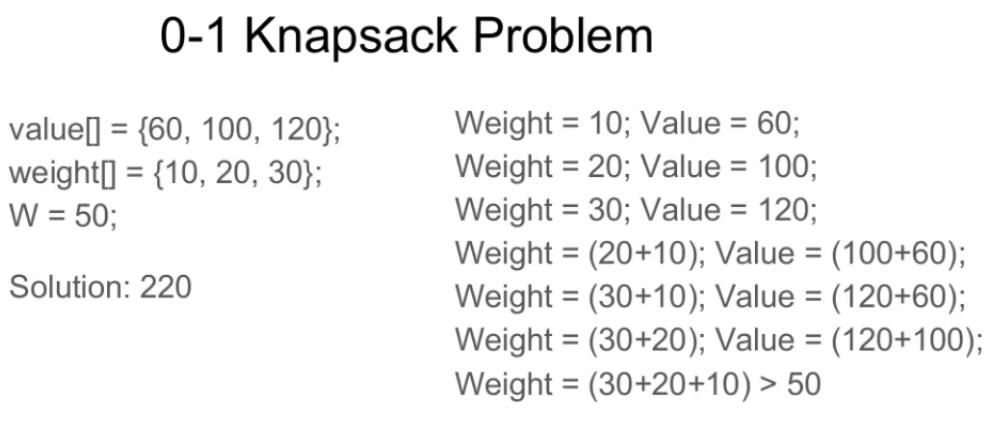
/

lis(1)

We can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of the same subproblems can be avoided by either using Memoization or Tabulation.  
**Pseudo Code**

/\* lis() returns the length of the longest increasing   
 subsequence in arr[ ] of size n \*/  
int lis( int arr[], int n )   
{   
 int lis[n]  
 lis[0] = 1   
 /\* Compute optimized LIS values in bottom up manner \*/  
 for (int i = 1; i < n; i++ )   
 {  
 lis[i] = 1;  
 for (int j = 0; j < i; j++ )   
 if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)   
 lis[i] = lis[j] + 1  
 }  
 // Return maximum value in lis[]  
 return \*max\_element(lis, lis+n)  
}

**Problem #4 : 0-1 Knapsack Problem**

**Description -**Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).  
  
  
**Optimal Substructure: To** consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.  
Therefore, the maximum value that can be obtained from n items is a max of the following two values.

1. Maximum value obtained by n-1 items and W weight (excluding nth item).
2. Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If the weight of the nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.  
  
**Overlapping Subproblems**

In the following recursion tree, K() refers to knapSack().

The two parameters indicated in the following recursion tree are n and W.

The recursion tree is for following sample inputs.

wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

K(3, 2) ---------> K(n, W)

/ \

/ \

K(2,2) K(2,1)

/ \ / \

/ \ / \

K(1,2) K(1,1) K(1,1) K(1,0)

/ \ / \ / \

/ \ / \ / \

K(0,2) K(0,1) K(0,1) K(0,0) K(0,1) K(0,0)

Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.

Since suproblems are evaluated again, this problem has Overlapping Subprolems property. So the 0-1 Knapsack problem has both properties -  
**Pseudo Code**

// Returns the maximum value that can be put in a knapsack of capacity W  
int knapSack(int W, int wt[], int val[], int n)  
{  
 int K[n+1][W+1]  
 // Build table K[][] in bottom up manner  
 for (i = 0; i <= n; i++)  
 {  
 for (w = 0; w <= W; w++)  
 {  
 if (i==0 || w==0)  
 K[i][w] = 0  
 else if (wt[i-1] <= w)  
 K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])  
 else  
 K[i][w] = K[i-1][w]  
 }  
 }  
 return K[n][W]  
}