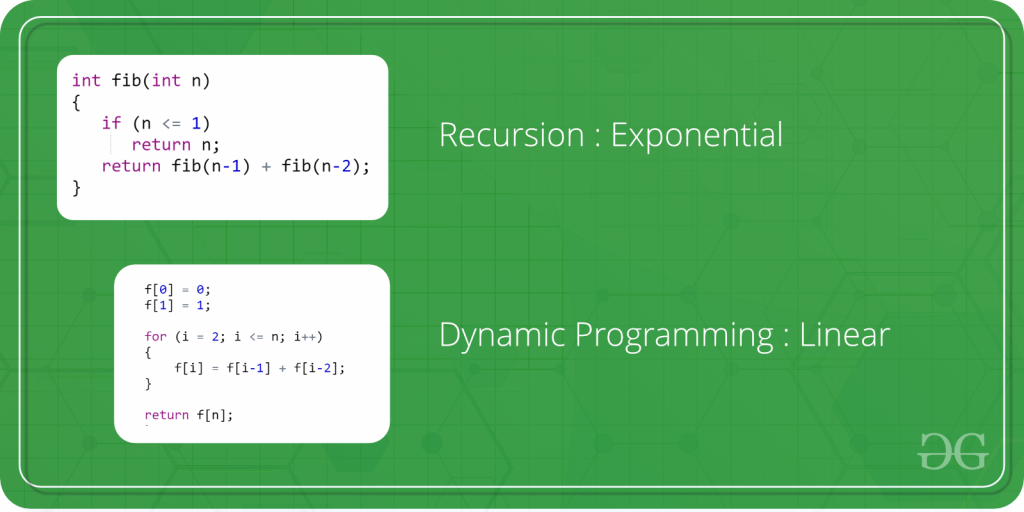
Dynamic Programming

Dynamic Programming is mainly an optimization over plain [recursion](https://www.geeksforgeeks.org/recursion/). Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial. For example, if we write simple recursive solution for [Fibonacci Numbers](https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/), we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.



# How to solve a Dynamic Programming Problem ?

* **Difficulty Level :** [Medium](https://www.geeksforgeeks.org/medium/)
* **Last Updated :** 13 Aug, 2021

**D**ynamic **P**rogramming (DP) is a technique that solves some particular type of problems in Polynomial Time. Dynamic Programming solutions are faster than the exponential brute method 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](https://www.geeksforgeeks.org/dynamic-programming-set-1/)
2. [Optimal Substructure Property](https://www.geeksforgeeks.org/dynamic-programming-set-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 maximizing or minimize certain quantities 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.

((AB)(CD))

(A((BC)D))

**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](https://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-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 weight. 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 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 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:

## C++

|  |  |
| --- | --- |
| // 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 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 C++  |  | | --- | | // 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 solution iterative. Please refer [tabulation and memoization](https://www.geeksforgeeks.org/tabulation-vs-memoizatation/) for more details. |

**Tabulation vs Memoization**

Prerequisite – [Dynamic Programming](https://www.geeksforgeeks.org/dynamic-programming-set-1/), [How to solve Dynamic Programming problems?](https://www.geeksforgeeks.org/solve-dynamic-programming-problem/)   
There are two different ways to store the values so that the values of a sub-problem can be reused. Here, will discuss two patterns of solving dynamic programming (DP) problem: 

1. **Tabulation:** Bottom Up
2. **Memoization:** Top Down

**Tabulation Method – Bottom Up Dynamic Programming**

As the name itself suggests starting from the bottom and accumulating answers to the top. Let’s discuss in terms of state transition.

Let’s describe a state for our DP problem to be dp[x] with dp[0] as base state and dp[n] as our destination state. So,  we need to find the value of destination state i.e dp[n].   
If we start our transition from our base state i.e dp[0] and follow our state transition relation to reach our destination state dp[n], we call it Bottom Up approach as it is quite clear that we started our transition from the bottom base state and reached the top most desired state.

**Now, Why do we call it tabulation method?**

To know this let’s first write some code to calculate the factorial of a number using bottom up approach. Once, again as our general procedure to solve a DP we first define a state. In this case, we define a state as dp[x], where dp[x] is to find the factorial of x.

Now, it is quite obvious that dp[x+1] = dp[x] \* (x+1) 

// Tabulated version to find factorial x.

int dp[MAXN];

// base case

int dp[0] = 1;

for (int i = 1; i< =n; i++)

{

dp[i] = dp[i-1] \* i;

}

The above code clearly follows the bottom-up approach as it starts its transition from the bottom-most base case dp[0] and reaches its destination state dp[n]. Here, we may notice that the dp table is being populated sequentially and we are directly accessing the calculated states from the table itself and hence, we call it tabulation method.

**Memoization Method – Top Down Dynamic Programming**

Once, again let’s describe it in terms of state transition. If we need to find the value for some state say dp[n] and instead of starting from the base state that i.e dp[0] we ask our answer from the states that can reach the destination state dp[n] following the state transition relation, then it is the top-down fashion of DP.

Here, we start our journey from the top most destination state and compute its answer by taking in count the values of states that can reach the destination state, till we reach the bottom most base state.

Once again, let’s write the code for the factorial problem in the top-down fashion 

// Memoized version to find factorial x.

// To speed up we store the values

// of calculated states

// initialized to -1

int dp[MAXN]

// return fact x!

int solve(int x)

{

if (x==0)

return 1;

if (dp[x]!=-1)

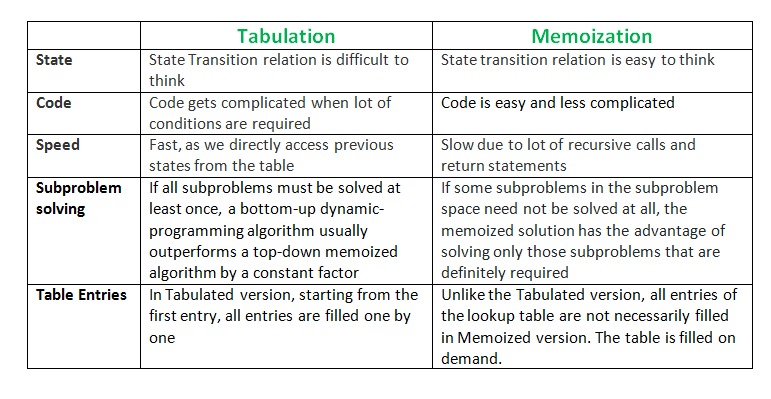
return dp[x];

return (dp[x] = x \* solve(x-1));

}

As we can see we are storing the most recent cache up to a limit so that if next time we got a call from the same state we simply return it from the memory. So, this is why we call it memoization as we are storing the most recent state values.

In this case the memory layout is linear that’s why it may seem that the memory is being filled in a sequential manner like the tabulation method, but you may consider any other top down DP having 2D memory layout like [Min Cost Path](https://www.geeksforgeeks.org/dynamic-programming-set-6-min-cost-path/), here the memory is not filled in a sequential manner. 



**Overlapping Subproblems Property in Dynamic Programming | DP-1**

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again. Following are the two main properties of a problem that suggests that the given problem can be solved using Dynamic programming.

In this post, we will discuss the first property (Overlapping Subproblems) in detail. The second property of Dynamic programming is discussed in the next post i.e. [Set 2](https://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/).  
1)**Overlapping Subproblems**  
2) **Optimal Substructure**

**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](https://www.geeksforgeeks.org/binary-search/) doesn’t have common subproblems. If we take an example of following recursive program for Fibonacci Numbers, there are many subproblems that are solved again and again.

* C

|  |
| --- |
| /\* a 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. There are following two different ways to store the values so that these values can be reused:   
a) Memoization (Top Down)   
b) Tabulation (Bottom Up)

**a) Memoization (Top Down):**The memoized program for a problem is similar to the recursive version with a small modification that 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++

|  |
| --- |
| /\* 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);      }        return lookup[n];  }    // Driver code  int main()  {      int n = 40;      \_initialize();      cout << "Fibonacci number is " << fib(n);      return 0;  }    // This is code is contributed by rathbhupendra |

**Output**

Fibonacci number is 102334155

**b) 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 program for Tabulated version \*/  #include <stdio.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];  }    int main()  {      int n = 9;      printf("Fibonacci number is %d ", fib(n));      return 0;  } |

**Output**

Fibonacci number is 34

Both Tabulated and Memoized store the solutions of subproblems. In Memoized version, 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 Memoized version. For example, [Memoized solution](https://www.ics.uci.edu/~eppstein/161/960229.html" \t "_blank)of the [LCS problem](http://en.wikipedia.org/wiki/Longest_common_subsequence_problem)doesn’t necessarily fill all entries.

To see the optimization achieved by Memoized and Tabulated solutions over the basic Recursive solution, see the time taken by following runs for calculating 40th Fibonacci number:  
[Recursive solution](https://ide.geeksforgeeks.org/vHt6ly)   
[Memoized solution](https://ide.geeksforgeeks.org/Z94jYR)   
[Tabulated solution](https://ide.geeksforgeeks.org/12C5bP)  
Time taken by the Recursion method is much more than the two Dynamic Programming techniques mentioned above – Memorization and Tabulation!

Also, see method 2 of [Ugly Number post](https://www.geeksforgeeks.org/ugly-numbers/) for one more simple example where we have overlapping subproblems and we store the results of subproblems.

We will be covering Optimal Substructure Property and some more example problems in future posts on Dynamic Programming.

Try the following questions as an exercise of this post.   
1) Write a Memoized solution for LCS problem. Note that the Tabular solution is given in the CLRS book.   
2) How would you choose between Memorization and Tabulation?

**Optimal Substructure Property in Dynamic Programming | DP-2**

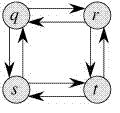
As we discussed in [Set 1](https://www.geeksforgeeks.org/dynamic-programming-set-1/), following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming:   
1) Overlapping Subproblems   
2) Optimal Substructure

We have already discussed Overlapping Subproblem property in the [Set 1](https://www.geeksforgeeks.org/dynamic-programming-set-1/). Let us discuss Optimal Substructure property here.

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

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 algorithm like [Floyd–Warshall](https://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/) and Single Source Shortest path algorithm for negative weight edges like [Bellman–Ford](https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/)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 longest simple path (path without cycle) between two nodes. Consider the following unweighted graph given in the [CLRS book](http://ressources.unisciel.fr/algoprog/s00aaroot/aa00module1/res/%5BCormen-AL2011%5DIntroduction_To_Algorithms-A3.pdf). 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 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.



**Optimal Binary Search Tree | DP-24**

Given a sorted array key *[0.. n-1]* of search keys and an array *freq[0.. n-1]* of frequency counts, where *freq[i]* is the number of searches for *keys[i]*. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.  
Let us first define the cost of a BST. The cost of a BST node is the level of that node multiplied by its frequency. The level of the root is 1.

**Examples:**

Input: keys[] = {10, 12}, freq[] = {34, 50}

There can be following two possible BSTs

10 12

\ /

12 10

I II

Frequency of searches of 10 and 12 are 34 and 50 respectively.

The cost of tree I is 34\*1 + 50\*2 = 134

The cost of tree II is 50\*1 + 34\*2 = 118

Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

There can be following possible BSTs

10 12 20 10 20

\ / \ / \ /

12 10 20 12 20 10

\ / / \

20 10 12 12

I II III IV V

Among all possible BSTs, cost of the fifth BST is minimum.

Cost of the fifth BST is 1\*50 + 2\*34 + 3\*8 = 142

**1) Optimal Substructure:**   
The optimal cost for freq[i..j] can be recursively calculated using the following formula.   
  
We need to calculate ***optCost(0, n-1)*** to find the result.   
The idea of above formula is simple, we one by one try all nodes as root (r varies from i to j in second term). When we make *rth* node as root, we recursively calculate optimal cost from i to r-1 and r+1 to j.   
We add sum of frequencies from i to j (see first term in the above formula)

**The reason for adding the sum of frequencies from i to j:**

This can be divided into 2 parts one is the freq[r]+sum of frequencies of all elements from i to j except r. The term freq[r] is added because it is going to be root and that means level of 1, so freq[r]\*1=freq[r]. Now the actual part comes, we are adding the frequencies of remaining elements because as we take r as root then all the elements other than that are going 1 level down than that is calculated in the subproblem. Let me put it in a more clear way, for calculating optcost(i,j) we assume that the r is taken as root and calculate min of opt(i,r-1)+opt(r+1,j) for all i<=r<=j. Here for every subproblem we are  choosing one node as a root. But in reality the level of subproblem root and all its descendant nodes will be 1 greater than the level of the parent problem root. Therefore the frequency of all the nodes except r should be added which accounts to the descend in their level compared to level assumed in subproblem.  
**2) Overlapping Subproblems**   
Following is recursive implementation that simply follows the recursive structure mentioned above. 

* C++

|  |
| --- |
| // A naive recursive implementation of  // optimal binary search tree problem  #include <bits/stdc++.h>  using namespace std;    // A utility function to get sum of  // array elements freq[i] to freq[j]  int sum(int freq[], int i, int j);    // A recursive function to calculate  // cost of optimal binary search tree  int optCost(int freq[], int i, int j)  {      // Base cases      if (j < i)  // no elements in this subarray          return 0;      if (j == i) // one element in this subarray          return freq[i];        // Get sum of freq[i], freq[i+1], ... freq[j]      int fsum = sum(freq, i, j);        // Initialize minimum value      int min = INT\_MAX;        // One by one consider all elements      // as root and recursively find cost      // of the BST, compare the cost with      // min and update min if needed      for (int r = i; r <= j; ++r)      {          int cost = optCost(freq, i, r - 1) +                     optCost(freq, r + 1, j);          if (cost < min)              min = cost;      }        // Return minimum value      return min + fsum;  }    // The main function that calculates  // minimum cost of a Binary Search Tree.  // It mainly uses optCost() to find  // the optimal cost.  int optimalSearchTree(int keys[],                        int freq[], int n)  {      // Here array keys[] is assumed to be      // sorted in increasing order. If keys[]      // is not sorted, then add code to sort      // keys, and rearrange freq[] accordingly.      return optCost(freq, 0, n - 1);  }    // A utility function to get sum of  // array elements freq[i] to freq[j]  int sum(int freq[], int i, int j)  {      int s = 0;      for (int k = i; k <= j; k++)      s += freq[k];      return s;  }    // Driver Code  int main()  {      int keys[] = {10, 12, 20};      int freq[] = {34, 8, 50};      int n = sizeof(keys) / sizeof(keys[0]);      cout << "Cost of Optimal BST is "           << optimalSearchTree(keys, freq, n);      return 0;  }    // This is code is contributed  // by rathbhupendra |

**Output:**

Cost of Optimal BST is 142

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. We can see many subproblems being repeated in the following recursion tree for freq[1..4]. 

https://media.geeksforgeeks.org/wp-content/uploads/matrixchainmultiplication.png

Since same subproblems are called again, this problem has Overlapping Subproblems property. So optimal BST problem has both properties (see [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems,](https://www.geeksforgeeks.org/archives/tag/dynamic-programming) recomputations of same subproblems can be avoided by constructing a temporary array cost[][] in bottom up manner.  
**Dynamic Programming Solution**   
Following is C/C++ implementation for optimal BST problem using Dynamic Programming. We use an auxiliary array cost[n][n] to store the solutions of subproblems. cost[0][n-1] will hold the final result. The challenge in implementation is, all diagonal values must be filled first, then the values which lie on the line just above the diagonal. In other words, we must first fill all cost[i][i] values, then all cost[i][i+1] values, then all cost[i][i+2] values. So how to fill the 2D array in such manner> The idea used in the implementation is same as [Matrix Chain Multiplication problem](https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/), we use a variable ‘L’ for chain length and increment ‘L’, one by one. We calculate column number ‘j’ using the values of ‘i’ and ‘L’. 

* C++

|  |
| --- |
| // Dynamic Programming code for Optimal Binary Search  // Tree Problem  #include <bits/stdc++.h>  using namespace std;    // A utility function to get sum of array elements  // freq[i] to freq[j]  int sum(int freq[], int i, int j);    /\* A Dynamic Programming based function that calculates  minimum cost of a Binary Search Tree. \*/  int optimalSearchTree(int keys[], int freq[], int n)  {      /\* Create an auxiliary 2D matrix to store results      of subproblems \*/      int cost[n][n];        /\* cost[i][j] = Optimal cost of binary search tree      that can be formed from keys[i] to keys[j].      cost[0][n-1] will store the resultant cost \*/        // For a single key, cost is equal to frequency of the key      for (int i = 0; i < n; i++)          cost[i][i] = freq[i];        // Now we need to consider chains of length 2, 3, ... .      // L is chain length.      for (int L = 2; L <= n; L++)      {          // i is row number in cost[][]          for (int i = 0; i <= n-L+1; i++)          {              // Get column number j from row number i and              // chain length L              int j = i+L-1;              cost[i][j] = INT\_MAX;                // Try making all keys in interval keys[i..j] as root              for (int r = i; r <= j; r++)              {              // c = cost when keys[r] becomes root of this subtree              int c = ((r > i)? cost[i][r-1]:0) +                      ((r < j)? cost[r+1][j]:0) +                      sum(freq, i, j);              if (c < cost[i][j])                  cost[i][j] = c;              }          }      }      return cost[0][n-1];  }    // A utility function to get sum of array elements  // freq[i] to freq[j]  int sum(int freq[], int i, int j)  {      int s = 0;      for (int k = i; k <= j; k++)      s += freq[k];      return s;  }    // Driver code  int main()  {      int keys[] = {10, 12, 20};      int freq[] = {34, 8, 50};      int n = sizeof(keys)/sizeof(keys[0]);      cout << "Cost of Optimal BST is " << optimalSearchTree(keys, freq, n);      return 0;  }    // This code is contributed by rathbhupendra |

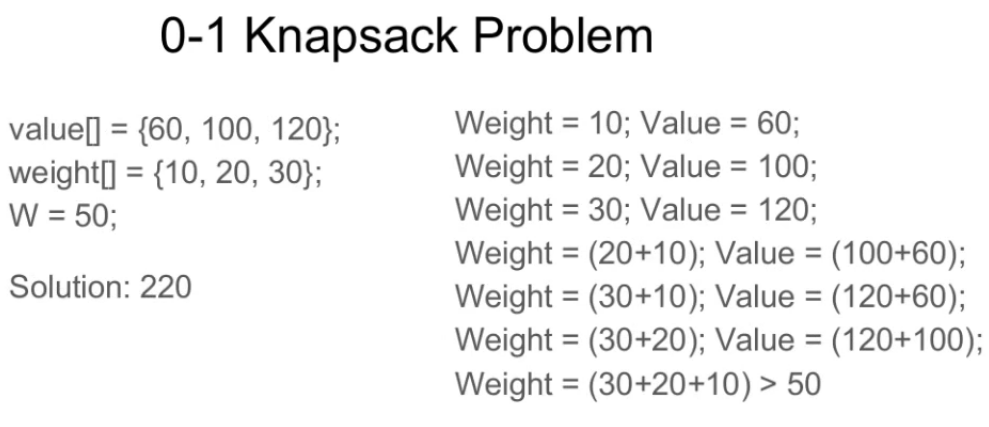
**Output:** 

Cost of Optimal BST is 142

**Notes**   
**1)** The time complexity of the above solution is O(n^4). The time complexity can be easily reduced to O(n^3) by pre-calculating sum of frequencies instead of calling sum() again and again.  
**2)** In the above solutions, we have computed optimal cost only. The solutions can be easily modified to store the structure of BSTs also. We can create another auxiliary array of size n to store the structure of tree. All we need to do is, store the chosen ‘r’ in the innermost loop.  
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

* 1. **Knapsack Problem | DP-10**

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).



**Method 1:** Recursion by Brute-Force algorithm OR Exhaustive Search.  
**Approach:** A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.  
***Optimal Sub-structure*:** To consider all subsets of items, there can be two cases for every item.

1. **Case 1:** The item is included in the optimal subset.
2. **Case 2:** The item is 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 ‘nth’ item is greater than ‘W’, then the nth item cannot be included and **Case 1** is the only possibility.

Below is the implementation of the above approach:

* C++

|  |
| --- |
| /\* A Naive recursive implementation of   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; }    // Returns the maximum value that  // can be put in a knapsack of capacity W  int knapSack(int W, int wt[], int val[], int n)  {        // Base Case      if (n == 0 || W == 0)          return 0;        // If 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));  }    // Driver code  int main()  {      int val[] = { 60, 100, 120 };      int wt[] = { 10, 20, 30 };      int W = 50;      int n = sizeof(val) / sizeof(val[0]);      cout << knapSack(W, wt, val, n);      return 0;  }    // This code is contributed by rathbhupendra |

**Output**

220

It should be noted that the above function computes the same sub-problems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. The time complexity of this naive recursive solution is exponential (2^n).

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(n, W)

K(3, 2)

/ \

/ \

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.

**Complexity Analysis:**

* **Time Complexity:** O(2n).   
  As there are redundant subproblems.
* **Auxiliary Space :**O(1).   
  As no extra data structure has been used for storing values.

Since subproblems are evaluated again, this problem has Overlapping Sub-problems property. So the 0-1 Knapsack problem has both properties (see [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem.

**Method 2:** Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), re-computation of same subproblems can be avoided by constructing a temporary array K[][] in bottom-up manner. Following is Dynamic Programming based implementation.

**Approach:** In the Dynamic programming we will work considering the same cases as mentioned in the recursive approach. In a DP[][] table let’s consider all the possible weights from ‘1’ to ‘W’ as the columns and weights that can be kept as the rows.   
The state DP[i][j] will denote maximum value of ‘j-weight’ considering all values from ‘1 to ith’. So if we consider ‘wi’ (weight in ‘ith’ row) we can fill it in all columns which have ‘weight values > wi’. Now two possibilities can take place:

* Fill ‘wi’ in the given column.
* Do not fill ‘wi’ in the given column.

Now we have to take a maximum of these two possibilities, formally if we do not fill ‘ith’ weight in ‘jth’ column then DP[i][j] state will be same as DP[i-1][j] but if we fill the weight, DP[i][j] will be equal to the value of ‘wi’+ value of the column weighing ‘j-wi’ in the previous row. So we take the maximum of these two possibilities to fill the current state. This visualization will make the concept clear:

Let weight elements = {1, 2, 3}

Let weight values = {10, 15, 40}

Capacity=6

0 1 2 3 4 5 6

0 0 0 0 0 0 0 0

1 0 10 10 10 10 10 10

2 0 10 15 25 25 25 25

3 0

**Explanation:**

For filling 'weight = 2' we come

across 'j = 3' in which

we take maximum of

(10, 15 + DP[1][3-2]) = 25

| |

'2' '2 filled'

not filled

0 1 2 3 4 5 6

0 0 0 0 0 0 0 0

1 0 10 10 10 10 10 10

2 0 10 15 25 25 25 25

3 0 10 15 40 50 55 65

**Explanation:**

For filling 'weight=3',

we come across 'j=4' in which

we take maximum of (25, 40 + DP[2][4-3])

= 50

For filling 'weight=3'

we come across 'j=5' in which

we take maximum of (25, 40 + DP[2][5-3])

= 55

For filling 'weight=3'

we come across 'j=6' in which

we take maximum of (25, 40 + DP[2][6-3])

= 65

* C++

|  |
| --- |
| // 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;  }    // 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;        vector<vector<int>> K(n + 1, vector<int>(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];  }    // Driver Code  int main()  {      int val[] = { 60, 100, 120 };      int wt[] = { 10, 20, 30 };      int W = 50;      int n = sizeof(val) / sizeof(val[0]);        cout << knapSack(W, wt, val, n);        return 0;  }    // This code is contributed by Debojyoti Mandal |

**Output**

220

**Complexity Analysis:**

* **Time Complexity:** O(N\*W).   
  where ‘N’ is the number of weight element and ‘W’ is capacity. As for every weight element we traverse through all weight capacities 1<=w<=W.
* **Auxiliary Space:** O(N\*W).   
  The use of 2-D array of size ‘N\*W’.

**Method 3:** This method uses Memoization Technique (an extension of recursive approach).  
This method is basically an extension to the recursive approach so that we can overcome the problem of calculating redundant cases and thus increased complexity. We can solve this problem by simply creating a 2-D array that can store a particular state (n, w) if we get it the first time. Now if we come across the same state (n, w) again instead of calculating it in exponential complexity we can directly return its result stored in the table in constant time. This method gives an edge over the recursive approach in this aspect.

* C++

|  |
| --- |
| // Here is the top-down approach of  // dynamic programming  #include <bits/stdc++.h>  using namespace std;    // Returns the value of maximum profit  int knapSackRec(int W, int wt[],                  int val[], int i,                  int\*\* dp)  {      // base condition      if (i < 0)          return 0;      if (dp[i][W] != -1)          return dp[i][W];        if (wt[i] > W) {            // Store the value of function call          // stack in table before return          dp[i][W] = knapSackRec(W, wt,                                 val, i - 1,                                 dp);          return dp[i][W];      }      else {          // Store value in a table before return          dp[i][W] = max(val[i]                        + knapSackRec(W - wt[i],                                     wt, val,                                     i - 1, dp),                         knapSackRec(W, wt, val,                                     i - 1, dp));            // Return value of table after storing          return dp[i][W];      }  }    int knapSack(int W, int wt[], int val[], int n)  {      // double pointer to declare the      // table dynamically      int\*\* dp;      dp = new int\*[n];        // loop to create the table dynamically      for (int i = 0; i < n; i++)          dp[i] = new int[W + 1];        // loop to initially filled the      // table with -1      for (int i = 0; i < n; i++)          for (int j = 0; j < W + 1; j++)              dp[i][j] = -1;      return knapSackRec(W, wt, val, n - 1, dp);  }    // Driver Code  int main()  {      int val[] = { 60, 100, 120 };      int wt[] = { 10, 20, 30 };      int W = 50;      int n = sizeof(val) / sizeof(val[0]);      cout << knapSack(W, wt, val, n);      return 0;  } |

**Output**

220

**Complexity Analysis:**

* **Time Complexity:** O(N\*W).   
  As redundant calculations of states are avoided.
* **Auxiliary Space:** O(N\*W).   
  The use of 2D array data structure for storing intermediate states-:

**[Note: For 32bit integer use long instead of int.]**  
**References:**

* <http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf>
* <http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf>

https://youtu.be/T4bY72lCQac?list=PLqM7alHXFySGMu2CSdW\_6d2u1o6WFTIO-   
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Method 4 :-** We use the dynamic programming approach but with optimized space complexity .

* Python3
* C++

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  int knapSack(int W, int wt[], int val[], int n)  {      // making and initializing dp array      int dp[W + 1];      memset(dp, 0, sizeof(dp));        for (int i = 1; i < n + 1; i++) {          for (int w = W; w >= 0; w--) {                if (wt[i - 1] <= w)                  // finding the maximum value                  dp[w] = max(dp[w],                              dp[w - wt[i - 1]] + val[i - 1]);          }      }      return dp[W]; // returning the maximum value of knapsack  }  int main()  {      int val[] = { 60, 100, 120 };      int wt[] = { 10, 20, 30 };      int W = 50;      int n = sizeof(val) / sizeof(val[0]);      cout << knapSack(W, wt, val, n);      return 0;  } |

**Output**

220

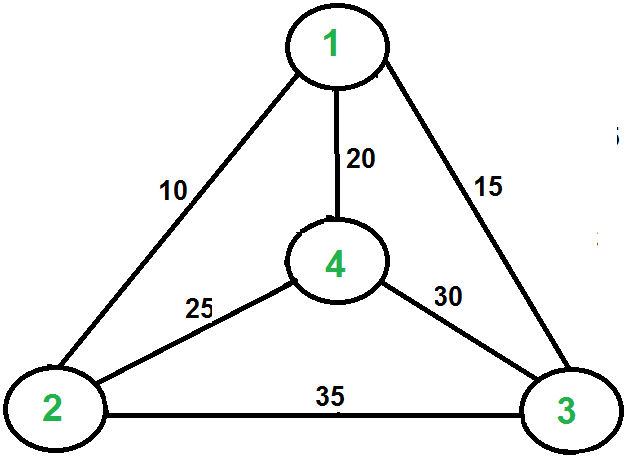
**Complexity Analysis**:

**Time Complexity**: O(N\*W). As redundant calculations of states are avoided.

**Auxiliary Space**: O(W) As we are using 1-D array instead of 2-D array.

**Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming)**

**Travelling Salesman Problem (TSP):** Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.  
Note the difference between [Hamiltonian Cycle](https://www.geeksforgeeks.org/backtracking-set-7-hamiltonian-cycle/) and TSP. The Hamiltoninan cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/Euler12.png)

For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The problem is a famous [NP hard](https://www.geeksforgeeks.org/np-completeness-set-1/)problem. There is no polynomial time know solution for this problem.

Following are different solutions for the traveling salesman problem.

**Naive Solution:**  
1) Consider city 1 as the starting and ending point.  
2) Generate all (n-1)! [Permutations](https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/)of cities.  
3) Calculate cost of every permutation and keep track of minimum cost permutation.  
4) Return the permutation with minimum cost.

Time Complexity: Θ(n!)

**Dynamic Programming:**  
Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far. Now the question is how to get cost(i)?  
To calculate cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term *C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i*.  
We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

If size of S is 2, then S must be {1, i},

C(S, i) = dist(1, i)

Else if size of S is greater than 2.

C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.

For a set of size n, we consider n-2 subsets each of size n-1 such that all subsets don’t have nth in them.  
Using the above recurrence relation, we can write dynamic programming based solution. There are at most O(n\*2n) subproblems, and each one takes linear time to solve. The total running time is therefore O(n2\*2n). The time complexity is much less than O(n!), but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.