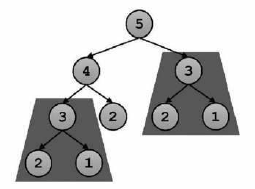
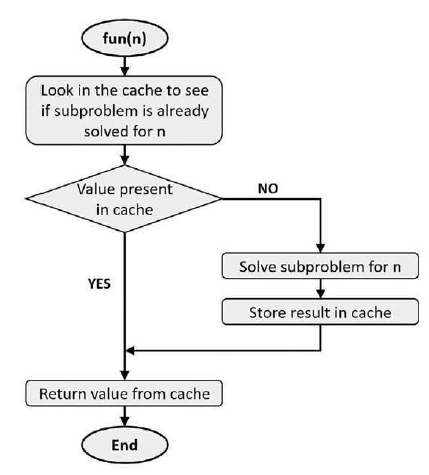
# Dynamic Programming

* **Recursion is a top-down approach of problem solving.**
* **Memoization is also top-down, but it is an improvement over recursion where we cache the result when a sub problem is solved, when same sub problem is encountered again, we use the results from cache rather than computing it again. It has the drawbacks of recursion with an improvement that one problem is solved only once.**
* **Dynamic Programming attempts to solve the problem in a bottom-up manner avoiding the overhead of recursion altogether.**
* **In almost all cases, bottom-up is better than top-down.**
* The best way to visualize DP problem is using recursion.
* DP is a bottom-up approach to problem solving where one problem is solved only once.
* A recursive solution takes more time and memory than the corresponding iterative solution.
* If iterative and recursive are equally easy to code then always use the iterative solution.
* Fibonacci using recursion without memoization:
  + int fib(int n) {
  + if (n == 1 || n == 2)
  + return 1;
  + else
  + return fib(n-1) + fib(n-2);
  + }
* Image showing the problem with only using recursion
  + 
* Fibonacci using recursion with memoization:
  + int[] memo = new int[N];
  + int fib(int n) {
  + if (memo[n] != 0)
  + return memo[n];
  + if (n==1 || n==2)
  + memo[n] = 1;
  + else
  + memo[n] = fib(n-1) + fib(n-2);
  + return memo[n];
  + }
* Image showing how recursion is memoization is done
  + 
* But at the end of the day, memoization with recursion is also recursion which takes huge space and time.
  + In fact memoization without overlapping sub problems is same as recursion.
  + That is why the bottom-up approach will be introduced now.
  + **Memoization is also dynamic programming**. Some authors in fact use the term "Memoized Dynamic Programming" or Top-Down Dynamic Programming" for memoization.
  + They use bottom-up dynamic programming to describe what we are calling Dynamic Programming here.
* Both Memoization and Dynamic Programming solves individual sub problems only once.
  + Memoization uses recursion and work top-down, whereas dynamic programming moves in opposite direction solving the problem bottom-up.
* Example of dynamic programming:
  + int fib(int n) {
  + if (n==1 || n==2)
  + return 1;
  + int a = 1;
  + int b = 1;
  + int c;
  + for (int i = 0; i <=n; i++)
  + {
  + c = a + b;
  + a = b;
  + b = c;
  + }
  + return c;
  + }
* Example of Dynamic Programming:
  + Calculate minimum cost for travelling from station 0 to station 1, station 2, station 3 and so on.
  + The train can only move in the positive direction:
  + These minimum costs will be saved in a one-dimensional array:
    - minCost[N];
  + Minimum cost to reach station-0 is 0 because we are already there:
    - minCost[0] = 0;
  + Minimum cost to reach Station-1 is cost[0][1], because that is the only way to reach Station-1:
    - minCost[1] = cost[0][1];
  + Minimum cost to reach Station-2 is minimum of below two values. Either go directly to Station-2 or take a break at Station-1.
    - minCost[0] + cost[0][2]
    - minCost[1] + cost[1][2]
  + Minimum cost to reach Station-3 is minimum of below three values:
    - minCost[0] + cost[0][3]
    - minCost[1] + cost[1][3]
    - minCost[2] + cost[2][3]
    - Here when we are breaking at station-2, we are using the already computed min cost of reaching station-2 and adding the actual cost going directly from station-2 to station-3.
  + Similarly, minimum cost to reach Station-3 is minimum of below 4 values:
    - minCost[0] + cost[0][4]
    - minCost[1] + cost[1][4]
    - minCost[2] + cost[2][4]
    - minCost[3] + cost[3][4]
  + and so on...
  + The Code is as follows:
    - int calculateMinCost(int cost[N][N]){
    - // minCost[i] = min cost from station-0 to station-i
    - int[] minCost = new int[N];
    - minCost[0] = 0;
    - minCost[1] = cost[0][1];
    - for (int i = 2; i < N; i++)
    - {
    - minCost[i] = cost[0][i];
    - for (int j = 1; j < i; j++)
    - {
    - if (minCost[i] > minCost[j] + cost[i][j])
    - minCost[i] = minCost[j] + cost[i][j];
    - }
    - }
    - return minCost[N-1];
    - }
* **Recursion is a top-down approach of problem solving.**
* **Memoization is also top-down, but it is an improvement over recursion where we cache the result when a sub problem is solved, when same sub problem is encountered again, we use the results from cache rather than computing it again. It has the drawbacks of recursion with an improvement that one problem is solved only once.**
* **Dynamic Programming attempts to solve the problem in a bottom-up manner avoiding the overhead of recursion altogether.**
* **In almost all cases, bottom-up is better than top-down.**
  + The one negative of bottom-up that is hardly ever seen is as follows:
  + In top-down approach, we do not solve all sub problems, we solve only those problems that need to be solved to get the solution of main problem.
  + In bottom-up dynamic programming, all the sub problems are solved before getting to the main problem.

# Dynamic Programming Problems

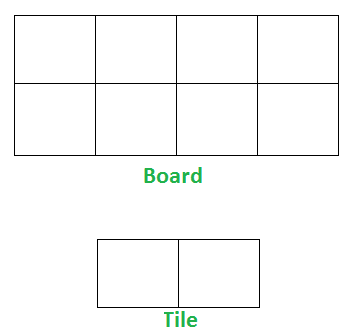
* **Longest Increasing subsequence**
  + In this we have to find the longest increasing subsequence.
  + Ex: {10, 22, 9, 33, 21, 50, 41, 60, 80}
  + Now the increasing subsequences are: {10}, {22}, {9}, {10, 22}, {10, 22, 33}, {10, 22, 33, 50}, {9, 21, 41, 60, 80} etc.
  + However, the longest subsequence is {10, 22, 33, 50, 60, 80} or {10, 22, 33, 41, 60, 80}
  + This is solved using DP as follows:
    - Create a memoization array of the same size
    - You will have to run nested for loops. The outer loop will start from 1 and inner loop will start from 0 and end at the iteration outer loop is at.
    - Initialize the all elements of this array as 1 because all elements in themselves are a subsequence of length 1.
    - Now the idea is that at every iteration, check all the elements before in the array. If the element in original array is greater than any element before it then do the following:
      * Find the maximum number in the memoization array before it and add one there.
      * Let’s say at 33, everything before it is smaller bit number at 22 is the maximum so you will take that number and add 1.
    - The biggest number in the array at the end is the answer.
    - You can keep track of this biggest number by making a maximum variable at the start and whenever memorization array is updates with a number bigger than the maximum then you update the maximum.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 22 | 9 | 33 | 21 | 50 | 41 | 60 | 80 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 2 | 1 | 3 | 0 | 0 | 0 | 0 | 0 |
| 1 | 2 | 1 | 3 | 2 | 0 | 0 | 0 | 0 |
| 1 | 2 | 1 | 3 | 2 | 4 | 0 | 0 | 0 |
| 1 | 2 | 1 | 3 | 2 | 4 | 4 | 0 | 0 |
| 1 | 2 | 1 | 3 | 2 | 4 | 4 | 5 | 0 |
| 1 | 2 | 1 | 3 | 2 | 4 | 4 | 5 | 6 |

* + Similar questions are:
    - Maximum Sum Increasing Subsequence
    - Maximum Length Chain of Pairs
    - Longest subsequence such that difference between adjacent is one
* **Knapsack Problem**
  + This requires to make a 2d table where you first need to know if the limit is 1kg then what is the maximum value you can have. If the limit is 2kg then how much value you can have. And in this way you keep going on until you reach the required maximum weight.
  + The 2d table requires the weights to be as the rows and the all the weights from 0 to maximum weight on the column as shown below:
  + In the example the maximum weight is 5 and the items have weight 1,2,3,4,5,6 and value 2,3,8,6,2,4.
  + So first, when only item of weight 1 is available then what is the maximum value possible for different weights. For weight 1, only 1 of item one can be used so maximum is value is 2. In fact, for all weights, if only item 1 is available then only that can be used. So, the first row will be filled with 2.
  + When the weight of the item is less than the total weight that can be accommodated, then the remaining weights are filled with the previous row. Let's say for the item with 3 kg. When 4 kg item was going to be filled then that leaves 1 kg. So, we go to previous row and see what the maximum weight with 1kg limit can be accommodated. However, if the number in the previous row is bigger then use that which means that the previous items are creating a greater value without including the present item.
  + This is done because only one copy of all items is available. So, you see what happens when one more item is made available.
  + If infinite copies would have been available then rows would have been the weight from 0 to weight required and column would be all the items weight. And in that case it would be seen what happens when one more kg space is allocated.
  + If the weight of item is more than what is allowed then just copy the number from above which means that this item cannot fit so all the items that have occurred till now should be taken as solution.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | **0** | **0** | **0** | **0** | **0** | **0** |
| 1(2) | **0** | **2** | **2** | **2** | **2** | **2** |
| 2(3) | **0** | **2** | **3** | **5** | **5** | **5** |
| 3(8) | **0** | **2** | **3** | **8** | **10** | **11** |
| 4(6) | **0** | **2** | **3** | **8** | **10** | **11** |
| 5(2) | **0** | **2** | **3** | **8** | **10** | **11** |
| 6(4) | **0** | **2** | **3** | **8** | **10** | **11** |

* **2 x n Tiling Problem**



* + The idea is that in how many ways can you assemble the 2x1 tile in 2xn board.
  + The problem can be divided into two problems
    - The tile can either be in vertical position in which case this becomes a problem that in how many ways can you assemble 2x(n-1) board.
    - The tile can be in horizontal position in which case the problem becomes that the only way this will work is if there are two horizontal tiles on top of each other and the problem becomes is how many ways can you assemble 2 x(n-2) board.
  + So the problem becomes:
    - How many ways can you get 2x(n-1) tiles.
    - +
    - How many ways can you get 2x(n-2) tiles.
  + Similar Problems:
    - Count number of binary strings without consecutive 1’s