**Question 8.11: Make Change -** How many ways can you have ‘amount’ cents using pennies, nickels, dimes and quarters?

|  |
| --- |
| **Execution Example:**  How many ways can you collect 27 cents with Quarters, Dimes, Nickels and Pennies?  Let's Choose 0 Quarters. How many ways can you collect 27 cents with Dimes, Nickels and Pennies?  Let's Choose 0 Dimes. How many ways can you collect 27 cents with Nickels and Pennies?  Let's Choose 0 Nickels. How many ways can you collect 27 cents with Pennies? ONLY ONE!  Let's Choose 1 Nickels. How many ways can you collect 22 cents with Pennies? ONLY ONE!  Let's Choose 2 Nickels. How many ways can you collect 17 cents with Pennies? ONLY ONE!  Let's Choose 3 Nickels. How many ways can you collect 12 cents with Pennies? ONLY ONE!  Let's Choose 4 Nickels. How many ways can you collect 7 cents with Pennies? ONLY ONE!  Let's Choose 5 Nickels. How many ways can you collect 2 cents with Pennies? ONLY ONE!  Let's Choose 1 Dimes. How many ways can you collect 17 cents with Nickels and Pennies?  Let's Choose 0 Nickels. How many ways can you collect 17 cents with Pennies? ONLY ONE!  Let's Choose 1 Nickels. How many ways can you collect 12 cents with Pennies? ONLY ONE!  Let's Choose 2 Nickels. How many ways can you collect 7 cents with Pennies? ONLY ONE!  Let's Choose 3 Nickels. How many ways can you collect 2 cents with Pennies? ONLY ONE!  Let's Choose 2 Dimes. How many ways can you collect 7 cents with Nickels and Pennies?  Let's Choose 0 Nickels. How many ways can you collect 7 cents with Pennies? ONLY ONE!  Let's Choose 1 Nickels. How many ways can you collect 2 cents with Pennies? ONLY ONE!  Let's Choose 1 Quarters. How many ways can you collect 2 cents with Dimes, Nickels and Pennies?  Let's Choose 0 Dimes. How many ways can you collect 2 cents with Nickels and Pennies?  Let's Choose 0 Nickels. How many ways can you collect 2 cents with Pennies? ONLY ONE!  Total Ways = All the (ONLY ONE!) statements = 13.  In this example, we have 27 cents and we can use quarters, dimes, nickels, and pennies. We iterate over all the possible choices (0 or 1 quarter) and we recurs once we made a decision because the problem terms into a different smaller problem. Eventually, we have to collect N cents with only pennies which there is always only one way to do. The solution is the number of times it reaches the base case which in this case is 13. |

**How to Identify These Types of Problems:** You are given a collection of item types. For example {apples, oranges, strawberries, bananas}, {Lawyers, Software Engineers, Business Analysts, HR}, {Quarters, Dimes, Nickels, Pennies} etc. You want to know given these item types, how many different ways (distribution of these types) are there that satisfy a certain condition. For example, how many different ways (distribution of coins) can you make 95 cents? Or, how many different ways (distribution of fruits) can you have a bag of 15 fruits? Or how many different ways (distribution of employees) can you run your start up business if you’re asked to hire 15 professionals and need at least 2 HR, 1 Lawyers, and 5 Engineers and 3 Business Analysts? They may ask you to count all the ways or print out all the ways. These problems are combinations with repetition allowed, order does not matter and there are conditions for which combinations are acceptable and which are not.

**Some Things You Need to Think About:** In each of these, you have 4 consecutive decisions to make. I.E. Choose the amount of quarters, then dimes, then nickels, then pennies. Or choose the amount of apples, then oranges, then strawberries, then bananas. When you make a decision, your “state” changes. You have less empty spots form employees, less spots for fruits or you have a lower amount to fill with change. Finally each decision has a possible range. You could only have zero or 1 quarters for 27 cents. You could only have 5 – 9 software engineers in the start-up problem. The way you need to think about it is that you have to make D1 – D4 decisions and each decision has a possible range which depends on the rules/constraints of the question and the current state (at D3 you already made D1 and D2). What you need to do is start iterate over all the possible decision values, make the decision (by affecting the state) and then recusing.

**Method Signature:** The first thing you need to think about is the method signature. The name could be something generic like “count ways” or “print ways” which will work for anything. Then you need to think about the parameters. You should have three parameters.

1. The first parameter is the constraints array. This array quantifies how each item affects the constraint. For example, in the change problem the array is {25, 10, 5, and 1}.This tells you how adding an extra quarter reduces the amount by 25 cents. For the start-up problem, the array is {5, 3, 2, 1} which is the minimums for each professional. This is probably the toughest parameter of the three. Never add in a string array with the item types like {Lawyers, Software Engineers, Business Analysts, HR}, {Quarters, Dimes, Nickels, Pennies} etc. These are useless. They are good to write down maybe to know the notation is the first index is the quarter and the last index of the constraints array is the penny.
2. The second parameter is an integer for the current decision.
3. The third parameter is the state variable. In these problems, when you make a decision on one item, it affects your next decision somehow. You either have less resources now (less employee and fruit spots available) or a lower amount to fill. You need this parameter because each time you make a decision, you have to update this variable and pass it to the next recursive call.

|  |
| --- |
| **public** **static** **int** makeChange(**int**[] coins, **int** currentDecision, **int** amount) {  **if** (currentDecision == coins.length - 1){ //if its last decision  **return** 1; //usually there is one feasible choice to meet conditions  }  **int** ways = 0;  **int** minimumDecision = 0;//figure out what are the posible values for current decision  **int** maximumDecision = *getMaximumDecision*(coins, currentDecision,amount);    **for** (**int** i = minimumDecision; i <= maximumDecision; i++) {//for each possible decision  amount = amount - (i \* coins[currentDecision]);//make the decision  ways += *makeChange*(coins, currentDecision + 1, amount);//recurs,update c.d.& state  amount = amount + (i \* coins[currentDecision]);//undo the decision  }  **return** ways;  }  //figure out what are the possible/feasible decision values for the current decision  //For example, if the amount is 27cents, and the current decision is 1, there could be  //the feasible values is 0,1 or 2 dimes.  **public** **static** **int** getMaxDecision(**int**[] coins, **int** currentDecision, **int** amount){  **int** maximumDecision = 0;  **while**( (maximumDecision + 1) \* coins[currentDecision] <= amount){  maximumDecision++;  }  **return** maximumDecision;  } |

|  |
| --- |
| **public** **static** **int** countWays(**int**[] minimums, **int** currentDecision, **int** employeesLeft){  **if**(currentDecision == minimums.length - 1){  **return** 1;  }    **int** ways = 0;  **int** minimumDecision = minimums[currentDecision];  **int** maximumDecision = *getMaximumDecision*(minimums, currentDecision, employeesLeft);    **for**(**int** i = minimumDecision; i <= maximumDecision; i++){//for each possible decision  employeesLeft -= i;//make the decision  ways += *countWays*(minimums, currentDecision + 1, employeesLeft);//rec,update c.d.& state  employeesLeft += i;//undo the decision  }    **return** ways;  }    **public** **static** **int** getMaximumDecision(**int**[] minimums, **int** currentDecision, **int** employeesLeft){  **int** maximumDecision = employeesLeft;  **for**(**int** i = currentDecision + 1; i < minimums.length; i++){  maximumDecision -= minimums[i];  }    **return** maximumDecision;  } |

Note: The for loop iterates over all the possible options for a current decision, the recursion makes that decision and proceeds down the tree to see all the combinations with that decision, the constraints specify the possible options in the for loop.

**Printing all the ways to make change:** Pass a parameter called print which is the sequence of decisions you make (for example ‘1 quarter 0 dimes 0 nickels 2 pennies’). Each time you make a decision (i.e. choose 1 quarter) add the decision string (i.e. ‘1 quarter’) to the print variable which is the list of all steps. When it gets to the base case (the leaves of the tree) it prints the print variable which is all the sequential decisions it has made. Also, there is a method called “get coin” which returns “quarter, dime, nickel, penny”, depending on what the current decision is.

|  |
| --- |
| **public** **static** **int** makeChange(**int**[] coins, **int** currentDecision, **int** amount, String print){  **if** (currentDecision == coins.length - 1){  String decisionString = " " + amount + *getCoin*(currentDecision);  print += decisionString; ; //add the decision to print which sequence of decisions  System.*out*.println(print);  **return** 1;  }  **int** ways = 0;  **int** minimumDecision = 0;  **int** maximumDecision = *getMaxDecision*(coins, currentDecision,amount);    String decisionString;  **for** (**int** i = minimumDecision; i <= maximumDecision; i++){  amount = amount - (i \* coins[currentDecision]);  decisionString = " " + i + *getCoin*(currentDecision);  print += decisionString; //add the decision to print which sequence of decisions  ways += *makeChange*(coins, currentDecision + 1, amount, print);//recurs  amount = amount + (i \* coins[currentDecision]);  print = print.substring(0, (print.length() – decisionString.length()));//undo print  }  **return** ways;  }  **OutPut:**  0 quarters 0 dimes 0 nickels 27 pennies  0 quarters 0 dimes 1 nickels 22 pennies  0 quarters 0 dimes 2 nickels 17 pennies  0 quarters 0 dimes 3 nickels 12 pennies  0 quarters 0 dimes 4 nickels 7 pennies  0 quarters 0 dimes 5 nickels 2 pennies  0 quarters 1 dimes 0 nickels 17 pennies  0 quarters 1 dimes 1 nickels 12 pennies  0 quarters 1 dimes 2 nickels 7 pennies  0 quarters 1 dimes 3 nickels 2 pennies  0 quarters 2 dimes 0 nickels 7 pennies  0 quarters 2 dimes 1 nickels 2 pennies  1 quarters 0 dimes 0 nickels 2 pennies  Total Different Combinations: 13 |

**Potential Traps:** Whatever variables you modify when you make a decision, undo them after. Make sure you don’t forget to modify any. In the example above, you constantly re-initialize decision string. If you however made it a local variable at the top, then you would have to undo it as well.

|  |
| --- |
| **public** **static** **int** configurationCountsHelper(**int** channels,**int** nodes,**int** currentDecision,String print){  **if**(currentDecision == (channels - 1)){  print += " " + nodes;  System.*out*.println(print);  **return** 1;  }    **int** ways = 0;  **int** minimumDecision = 0;  **int** maximumDecision = nodes;  String decisionString;  **for**(**int** i = minimumDecision; i <= maximumDecision; i++){//For each option  nodes -= i; //make that decision  decisionString = " " + i;  print += decisionString;  ways += *configurationCountsHelper*(channels, nodes, currentDecision + 1, print);  print = print.substring(0, (print.length() – decisionString.length()));  nodes += i;  }    **return** ways;  }    **public** **static** **int** configurationCounts(**int** channels, **int** nodes){  **return** *configurationCountsHelper*(channels, nodes, 0, "");  }    **public** **static** **void** main(String[] args){  **int** results = *configurationCounts*(3,3);  System.*out*.println("\nTotal: " + results);  }  Execution:  0 0 3  0 1 2  0 2 1  0 3 0  1 0 2  1 1 1  1 2 0  2 0 1  2 1 0  3 0 0  Total Configurations: 10 |

**Triple Step:** A child is running up a staircase in n steps and can hop either 1 step, 2 steps or 3 steps at a time. Implement a method to count how many ways the child can run up the stairs.

Normal Solution:

|  |
| --- |
| **public** **static** **int** countWays(**int** n) {  **if** (n < 0) {  **return** 0; //path doesn't count since we passed the top  } **else** **if** (n == 0) {  **return** 1; //we are at the top so path counts  }  **int** minimumDecision = 1; //we can take a single step  **int** maximumDecision = 3; // double step or trip step  **int** ways = 0;  **for**(**int** i = minimumDecision; i <= maximumDecision; i++){  n -= i; //make the step  ways += *countWays*(n); //recurse  n += i; //undo the step  }  **return** ways;  }  If the minimum decision and maximum decision is only two apart, then it becomes better to avoid the for-loop and simply do the below which is logically equivalent. The above allows to make adjustments easily (i.e. adding quadruple steps would only require increment the maximum decision).  **public** **static** **int** countWays(**int** n) {  **if** (n < 0) {  **return** 0;  } **else** **if** (n == 0) {  **return** 1;  }  **return** *countWays*(n - 1) + *countWays*(n - 2) + *countWays*(n - 3);  }  If the minimum decision and maximum decision are far apart, you could cap the maximum decision at either 3 or steps away from the top whichever is smaller. This allows the case where n < 0 to not be possible. It can never go past the top this way.  **public** **static** **int** countWays(**int** n) {  **if** (n == 0) {  **return** 1; //we are at the top so path counts  }  **int** minimumDecision = 1; //we can take a single step  **int** maximumDecision = Math.*min*(n,3); // double step or trip step  **int** ways = 0;  **for**(**int** i = minimumDecision; i <= maximumDecision; i++){  n -= i; //make the step  ways += *countWays*(n); //recurse  n += i; //undo the step  }  **return** ways;  }  Note: The way you can decide which recursive form to take is based on the distance betweem the minimum decision and the maximum decision. If this is larger than three, go with the longer form. If this is less than three, go with the shorter version. Also, if you have to define minimum and maximum decision, then remove that possibility that you go over the steps. |

|  |
| --- |
| Memoization:  **public** **static** **int** countWays(**int** n) {  **int**[] memo = **new** **int**[n + 1];  Arrays.*fill*(memo, -1);  **return** *countWaysWithMemo*(n, memo);  }    **public** **static** **int** countWaysWithMemo(**int** n, **int**[] memo) {  **if** (n == 0) {  **return** 1;  } **else** **if** (memo[n] > -1) {  **return** memo[n]; //If this was solved before  }    **int** minimumDecision = 1; //we can take a single step  **int** maximumDecision = Math.*min*(n,3); // double step or trip step  **int** ways = 0;  **for**(**int** i = minimumDecision; i <= maximumDecision; i++){  n -= i; //make the step  ways += *countWays*(n); //recurse  n += i; //undo the step  }    memo[n] = ways;//set the memo table n so you don't have to re-compute this  **return** ways;  }  To add memoization, simply include a memo array in the paramter. Whenever you solve countWays(n) you will save the value at memo(n). Then later on, you will check if you solve this sub problem before to not waste extra time. Watch Gayle’s hackerrank video about memoization to understand what it does. |

|  |
| --- |
| **Make Change with memoization:**  **public** **static** **int** makeChange(**int**[] coins,**int** currentDecision,**int** amount,**int**[][] memo){  **if**(currentDecision == coins.length - 1){  **return** 1;  }**else** **if**(memo[amount][currentDecision] > 0){  **return** memo[amount][currentDecision]; //If this was solved before  }    **int** minimumDecision = 0;  **int** maximumDecision = *getMaximumDecision*(coins, currentDecision,amount);    **int** ways = 0;  **for**(**int** i = minimumDecision; i <= maximumDecision; i++){  amount = amount - (i \* coins[currentDecision]);//make the decision  ways += *makeChange*(coins, currentDecision + 1, amount, memo);//recurs,update  amount = amount + (i \* coins[currentDecision]);//undo the decision  }    memo[amount][currentDecision] = ways; ;//set the memo table  **return** ways;  }  **public** **static** **int** makeChange(**int** amount, **int**[] coins){  **int**[][] memo = **new** **int**[amount + 1][coins.length];  **int** ways = (*makeChange*(coins,0,amount, memo));  **return** ways;  }  Notes: Since it was a potentially large double array, we did not bother to initialize the values to -1, we just let java initialize it to zero. This works out because all change values at least yield one result (amount in pennies). We needed a double array because we have to save the value of an amount with which coins. For example, total ways of 60 cents with quarters, dimes, nickels and pennies is a different sub problem than total ways of 60 cents with only pennies. |