Algorithm Tutorials

How To Find a Solution

By Dumitru
TopCoder Member

Introduction

With many TopCoder problems, the solutions may be found instantly just by reading their descriptions. This is possible thanks to a collection of common traits that problems with similar solutions often have. These traits serve as excellent hints for experienced problem solvers that are able to observe them. The main focus of this article is to teach the reader to be able to observe them too.

Straight-forward problems that don't require any special technique (e.g. simulation, searching, sorting etc.)

In most cases, these problems will ask you to perform some step by step, straight-forward tasks. Their constraints are not high, and not too low. In most cases the first problems (the easiest ones) in TopCoder Single Rounds Matches are of this kind. They test mostly how fast and properly you code, and not necessarily your algorithmic skills.

Most simple problems of this type are those that ask you just to execute all steps described in the statement.

> **BusinessTasks - SRM 236 Div 1:**
> N tasks are written down in the form of a circular list, so the first task is adjacent to the last one. A number n is also given. Starting with the first task, move clockwise (from element 1 in the list to element 2 in the list and so on), counting from 1 to n. When your count reaches n, remove that task from the list and start counting from the next available task. Repeat this procedure until one task remains. Return it.

For N<=1000 this problem is just a matter of coding, no special algorithm is needed - do this operation step by step until one item is left. Usually these types of problems have a much smaller N, and so we'll not consider cases where N is very big and for which complicated solution may be needed. Remember that in TopCoder competitions even around 100 millions sets of simple operations (i.e. some multiplications, attributions or if statements) will run in allowed time.

This category of problems also includes those that need some simple searches.

> **TallPeople - SRM 208 Div 1:**
> A group of people stands before you arranged in rows and columns. Looking from above, they form an R by C rectangle of people. Your job is to return 2 specific heights - the first is computed by finding the shortest person in each row, and then finding the tallest person among them (the "tallest-of-the-shortest"); and the second is computed by finding the tallest person in each column, and then finding the shortest person among them (the "shortest-of-the-tallest").

As you see this is a really simple search problem. What you have to do is just to follow the steps described in the statement and find those 2 needed heights. Other TC problems may ask you to sort a collection of items by respecting certain given rules. These problems may be also included in this category, because they too are straight-forward - just sort the items respecting the rules! You can do that with a simple O(N^2) sorting algorithm, or use standard sorting algorithm that exist in your coding language. It's just a matter of coding.

Other example(s):
MedalTable - SRM 209 Div 1.

Breadth First Search (BFS)

Problems that use BFS usually ask to find the fewest number of steps (or the shortest path) needed to reach a certain end point (state) from the starting one. Besides this, certain ways of passing from one point to another are offered, all of them having the same cost of 1 (sometimes it may be equal to another number). Often there is given a N x M table (formed of N lines and M columns) where certain cells are passable and others are impassable, and the target of the problem is to find the shortest time/path needed to reach the end point from the start one. Such tables may represent mazes, maps, cities, and other similar things. These may be considered as classical BFS problems. Because BFS

complexity is in most cases linear (sometimes quadratic, or N logN), constraints of N (or M) could be high - even up to 1 million.

> **SmartWordToy - SRM 233 Div 1:**
> A word composed of four Latin lowercase letters is given. With a single button click you can change any letter to the previous or next letter in alphabetical order (for example 'c' can be changed to 'b' or 'd'). The alphabet is circular, thus 'a' can become 'z', and 'z' can become 'a' with one click.
>
> A collection of constraints is also given, each defining a set of forbidden words. A constraint is composed of 4 strings of letters. A word is forbidden if each of its characters is contained in corresponding string of a single constraint, i.e. first letter is contained in the first string, the second letter - in the second string, and so on. For example, the constraint "lf a tc e" defines the words "late", "fate", "lace" and "face".
>
> You should find the minimum number of button presses required to reach the word finish from the word start without passing through forbidden words, or return -1 if this is not possible.

Problem hints:

- Words can be considered as states. There are at most 26^4 different words composed of 4 letters (thus a linear complexity will run in allowed time).
- There are some ways to pass from one state to another.
- The cost of passing from a state to another is always 1 (i.e. a single button click).
- You need to find the minimum number of steps required to reach the end state from start state.

Everything indicates us that it's a problem solved by the help of a BFS. Similar things can be found in any other BFS problems. Now let's see an interesting case of BFS problems.

> **CaptureThemAll - SRM 207 Div 2 (3rd problem):**
> Harry is playing a chess game. He has one knight, and his opponent Joe has a queen and a rook. Find the minimum number of steps that Harry's knight has to jump so that it captures both the queen and the rook.

Problem hints: At first sight this may seem like dynamic programming or backtracking. But as always, take a look into the text of the statement. After a while you should observe the following things:

- A table is given.
- The knight can jump from one cell to some of its neighbors.
- The cost of passing from a cell to another is always 1 (just one jump).
- You need to find the minimum number of steps (jumps).

Given this information we can see that the problem can be easily solved by the help of BFS. Don't get confused by the fact that connected points are represented by unconnected cells. Think of cells as points in a graph, or states (whatever you want) - and in order to pass from one point to another, the knight should be able to jump from the first to the second point.

Notice again that the most revealing hint about the BFS solution is the cost of 1 for any jump.

Train yourself in finding the hints of a BFS problem in following examples:

Other example(s):
RevolvingDoors - SRM 223 Div 1
WalkingHome - SRM 222 Div 1
TurntableService - SRM 219 Div 1

Flood Fill

Sometimes you may encounter problems that are solved by the help of Flood Fill, a technique that uses BFS to find all reachable points. The thing that makes them different from BFS problems described above is that a minimum path/cost is not needed.

For example, imagine a maze where 1 represents impassable cells and 0 passable cells. You need to find all cells that are reachable from the upper-left corner. The solution is very simple - take one-by-one a visited vertex, add its unvisited neighbors to the queue of visited vertices and proceed with the next one while the queue is still populated. Note that in most cases a DFS (Depth First Search) will not work for such problems due to stack overflows. Better use a BFS. For inexperienced users it may seem harder to implement, but after a little training it becomes a "piece of cake". A good example of such problem would be:

> **grafixMask - SRM 211 Div 1:**
> A 400 x 600 bitmap is given. A set of rectangles covers certain parts of this bitmap (the corners of rectangles have integer coordinates). You need to find all contiguous uncovered areas, including their sizes.

Problem hints: What do we have here?

- A map (table)
- Certain points are impassable (those covered by given rectangles)
- Contiguous areas need to be found

It is easy to understand that a problem with such a statement needs a Flood Fill. Usually problems using it are very easy to detect.

Brute Force and Backtracking

I have placed these 2 techniques in the same category because they are very similar. Both do the same thing - try all possible cases (situations) and choose the best one, or count only those that are needed (depending on the problem). Practically, Backtracking is just more advanced and optimized than Brute Force. It usually uses recursion and is applied to problems having low constraints (for example N<=20).

Brute Force

There are many problems that can be solved by the help of a simple brute force. Note that the limits

must not be high. How does a brute force algorithm work? Actually, it tries all possible situations and selects the best one. It's simple to construct and usually simple to implement. If there is a problem that asks to enumerate or find all possible ways (situations) of doing a certain thing, and that doesn't have high limits - then it's most probably a brute force problem.

> **GeneralChess - SRM 197 Div 1:**
> You are given some knights (at most 8), with their positions on the table (-10000<=x, y<=10000). You need to find all positions to place another one, so that it threatens all given pieces.

Problem hints: Well, this is one of the easiest examples. So which are the hints of this statement?

- You need to find all possible situations (positions) that satisfy a certain rule (threatens all given pieces).
- The limits are very low - only 8 knights are at most given.

It's a common Brute Force problem's statement. Note that x and y limits are not relevant, because you need only try all positions that threaten one of the knights. For each of these positions see if the knight placed at that position threatens all others too.

Another interesting problem would be:

> **LargestCircle - SRM 212 Div 2 (3rd problem):**
> Given a regular square grid, with some number of squares marked, find the largest circle you can draw on the grid that does not pass through any of the marked squares. The circle must be centered on a grid point (the corner of a square) and the radius must be an integer. Return the radius of the circle.
>
> The size of the grid is at most 50.

Problem hints: And again one of the most important hints is the low limit of the size of the grid - only 50. This problem is possible to be solved with the help of the Brute Force because for each cell you can try to find the circle whose center is situated in that cell and that respects the rules. Among all of these circles found, select the one that has the greatest radius.

Complexity analysis: there are at most 50x50 cells, a circle's radius is an integer and can be at most 25 units, and you need a linear time (depending on your implementation) for searching the cells situated on the border of the circle. Total complexity is low and thus you can apply a simple Brute Force here.

Other example(s):
Cafeteria - SRM 229 Div 1
WordFind - SRM 232 Div 1

Backtracking

This technique may be used in many types of problems. Just take a look at the limits (N, M and other main parameters). They serve as the main hint of a backtrack problem. If these are very small and you haven't found a solution that's easier to implement - then just don't waste your time on searching it and implement a straight-forward backtracking solution.

Usually problems of this kind ask you to find (similarly to Brute Force):

1. Every possible configuration (subset) of items. These configurations should respect some given rules.
2. The "best" configuration (subset) that respects some given rules.

> **BridgeCrossing - SRM 146 Div 2 (3rd problem):**
> A group of people is crossing an old bridge. The bridge cannot hold more than two people at once. It is dark, so they can't walk without a flashlight, and they only have one flashlight! Furthermore, the time needed to cross the bridge varies among the people in the group. When people walk together, they always walk at the speed of the slowest person. It is impossible to toss the flashlight across the bridge, so one person always has to go back with the flashlight to the others. What is the minimum amount of time needed to get all the people across the bridge?
>
> There are at most 6 people.

Problem hints:

- First look at the constraints - there are at most ONLY 6 people! It's enough for generating all possible permutations, sets etc.
- There are different possible ways to pass the people from one side to another and you need to find the best one.

This is of course a problem solved with a backtracking: at the beginning choose any 2 people to pass the bridge first, and after that at each step try to pass any of those that have been left on the start side. From all these passages select the one that needs the smallest amount of time. Note that among persons that have passed over the bridge, the one having the greatest speed should return (it's better

than returning one having a lower speed). This fact makes the code much easier to implement. After having realized these things - just code the solution. There may be others - but you will lose more time to find another than to code this one.

> **MNS - SRM 148 Div 1:**
> 9 numbers need to be arranged in a magic number square. A magic number square is a square of numbers that is arranged such that every row and column has the same sum. You are given 9 numbers that range from 0 to 9 inclusive. Return the number of distinct ways that they can be arranged in a magic number square. Two magic number squares are distinct if they differ in value at one or more positions.

Problem hints: Only 9 numbers are given at most; and every distinct way (configuration) to arrange the numbers so that they form a magic number square should be found. These are the main properties of a Backtracking problem. If you have observed them - think about the code. You can generate all permutations of numbers and for each of them check if it forms a magic square. If so - add it to the answer. Note that it must be unique. A possible way to do that - is to have a list of earlier found configurations, thus for each new magic square check if it exists in that list and if it doesn't - add it to the answer. There will not be many distinct magic squares, thus no additional problems will appear when applying this method.

Other example(s):
WeirdRooks - SRM 234 Div 1

Dynamic Programming

Quite a few problems are solved with the help of this technique. Knowing how to detect this type of problem can be very valuable. However in order to do so, one has to have some experience in dynamic programming. Usually a DP problem has some main integer variables (e.g. N) which are neither too small, nor too big - so that a usual DP complexity of N^2, N^3 etc. fits in time. Note that in the event that N is very small (for TC problems usually less than 30) - then it is likely the problem is not a DP one. Besides that there should exist states and one or more ways (rules) to reach one greater state from another lower one. In addition, greater states should depend only upon lower states. What is a so-called state? It's just a certain configuration or situation. To better understand dynamic programming, you may want to read this article.

> Let's analyze a simple classic DP problem:
>
> Given a list of N coins with their values (V1, V2, ... ,VN), and the total sum S. Find the minimum number of coins the sum of which is S (you can use as many coins of one type as you want), or report that it's not possible to select coins in such a way that they sum up to S.
>
> Let N <= 1,000 and S <= 1,000.

Problem hints:

- Two main integer variables are given (N and S). These are neither too small, nor are they too big (i.e. a complexity of N*S fits in time).
- A state can be defined as the minimum number of coins needed to reach a certain sum.
- A sum (state) i depends only on lower sums (states) j (j<i).
- By adding a coin to a certain sum - another greater sum is reached. This is the way to pass from one state to another.

Thus all properties of a DP problem are uncovered in this statement. Let's see another (slightly harder) DP problem

> **ZigZag - 2003 TCCC Semifinals 3:**
> A sequence of numbers is called a zig-zag sequence if the differences between successive numbers strictly alternate between positive and negative. The first difference (if one exists) may be either positive or negative. A sequence with fewer than two elements is trivially a zig-zag sequence. Given a sequence of integers, return the length of the longest subsequence that is a zig-zag sequence. A subsequence is obtained by deleting some number of elements (possibly zero) from the original sequence, leaving the remaining elements in their original order. Assume the sequence contains between 1 and 50 elements, inclusive.

Problem hints:

- There are N numbers given (1<=N<=50), thus N isn't too small, nor too big.
- A state (i,d) can be defined as the length of the longest zig-zag subsequence ending with the i-th number, for which the number before the last one is smaller than it for d=0, and bigger for d=1.
- A state i (i.e. a subsequence ending with the i-th number) depends only on lower states j (j<i).
- By adding a number to the end of a subsequence - another bigger (greater) subsequence is created. This is the way to pass from one state to another.

As you can see - this statement has almost the same traits (pattern) as in the previous problem. The most difficult part in identifying a DP problem statement is observing/seeing the states with the properties described above. Once you can do that, the next step is to construct the algorithm, which is out of the scope of this article.

Other example(s):
ChessMetric - 2003 TCCC Round 4
AvoidRoads - 2003 TCO Semifinals 4
FlowerGarden - 2004 TCCC Round 1
BadNeighbors - 2004 TCCC Round 4

Hard Drills:

Maximum Flow

In many cases it's hard to detect a problem whose solution uses maximum flow. Often you have to create/define graphs with capacities based on the problem statement.

Here are some signs of a Maximum Flow problem:

- Take a look at the constraints, they have to be appropriate for a O(N^3) or O(N^4) solution, i.e. N shouldn't be greater than 500 in extreme cases (usually it's less than 100).
- There should be a graph with edges having capacities given, or you should be able to define/create it from data given in the statement.
- You should be finding a maximum value of something.

> Sample problem:
>
> You are given a list of w ater pipes, each having a certain maximum w ater flow capacity. There are w ater pipes connected together at their extremities.
>
> You have to find the maximum amount of w ater that can flow from start junction to end junction in a unit of time.
>
> Let N<=100.

As you can see - it's a straight-forward maximum flow problem: water pipes represent edges of the graph, their junctions - vertices; and you have to find the maximum value of amount of water that can flow from start to end vertex in a unit of time.

Optimal Pair Matching:

These problems usually have a list of items (from a set A) for which other items (from a set B) should be assigned under some rules, so that all (or a maximum possible number of) items from set A have to each be assigned to a certain item from set B.

Mixed:

Some problems need other techniques in addition to network flows.

> **Parking - SRM 236 Div 1:**
> N cars and M parking lots are given. They are situated on a rectangular surface (represented by a table), w here certain cells are impassable. You should find a w ay to assign each car to a parking lot, so that the greatest of the shortest distances from each car to its assigned parking lot is as small as possible. Each parking lot can have at most one car assigned to it.

Problem hints: By reading this problem one can simply understand the main idea of the solution - it should be something similar to optimal pair matching, because each car (point from a set A) should be assigned to a parking lot (point from a set B) so that all are assigned and that there is at most one car assigned to a parking lot. Additionally, there can be cars that can't reach certain parking lots, thus some pairs of points (one point from A and the other from B) are not connected. However a graph should be created for optimal pair matching. The way to make it is clear - an edge exists between a car and a parking lot if only there is a path between them, and its cost is equal to the shortest distance needed for the car to reach the parking lot. The next step of the solution is a binary search on the longest edge. Although it may be out of the scope of this article, I will provide a short explanation: At each step delete those edges of the initial graph that have costs greater than a certain value C (Note that you'll have to save the initial graph's state in order to repeat this step again for other C values). If it's possible in this case to assign all the cars to parking lots - then take a smaller C, and repeat the same operation. If not - take a greater C. After a complete binary search, the smallest C for which a complete assignment is possible will be found. This will be the answer.

Linear Programming (Simplex)

Most of the common traits of problems solved with the help of the linear programming technique are:

- You are given collection of items having different costs/weights. There is a certain quantity of each item that must be achieved.
- A list of sets is given. These sets are composed of some of the available items, having certain quantities of each of them. Each set has a certain cost.
- The goal of the problem is to find an optimal combination (the cheapest one) of these sets so that the sum of quantities of each of the items they have is exactly the one needed to achieve.

At first it may seem confusing, but let's see an example:

> **Mixture - SRM 231 Div 1):**
> A precise mixture of a number of different chemicals, each having a certain amount, is needed. Some mixtures of chemicals may be purchased at a certain price (the chemical components for the mixture might not be available in pure form). Each of them contains certain amounts of some of the chemicals. You need not purchase the available mixtures in integral amounts. Hence if you purchase a 1.5 of a mixture having a price of 3 and amounts of "2 0 1", you'll pay 4.5 and get "3 0 1.5" amounts of chemicals. Your task is to determine the low est price that the desired mixture can be achieved.

Problem hints:

- A collection of items (chemicals).
- A list of sets (available mixtures), each containing certain amounts of each of the items, and having a certain cost.
- You need to find the lowest price of the desired collection of items achieved by the combination of

the available sets. More than that - you can take also non-integral amounts of mixtures.

These are exactly the traits described above.

Conclusion

If you have found this article interesting and you have learned new things from it - train yourself on any of the problems in the TopCoder Algorithm Arena. Try hard to see the hints and determine the type of the solution by carefully reading through the problem statement. Remember, there are still many problems that may not be included properly in any of the categories described above and may need a different approach.