Competitive Programming Tutorial

Ethan Arnold

March 2016

Contents

1 Introduction **1**

2 Implementation **2**

3 Brute force **4**

4 Backtracking **4**

5 Binary search **9**

6 Divide and conquer **9**

7 Greedy **9**

8 Math **9**
8.1 Introduction . . . 9
8.2 Seive of Eratosthenes . . . 9
8.3 Binary exponentiation . . . 9
8.4 Matrix exponentiation . . . 9
8.5 Recurrences . . . 9
8.6 Euclidean algorithm . . . 9
8.7 Extended Euclidean algorithm . . . 9
8.8 Chinese remainder theorem . . . 9
8.9 Catalan numbers . . . 9

9 Dynamic programming **9**
9.1 Introduction . . . 9
9.2 Fibonacci . . . 9
9.3 Coin change . . . 9
9.4 Longest increasing subarray . . . 9
9.5 Long increasing subsequence . . . 9
9.6 Longest common subsequence . . . 9
9.7 Binary knapsack . . . 9
9.8 Unbounded knapsack . . . 9
9.9 Largest square . . . 9
9.10 Practice problems . . . 9

10 Graphs **9**
10.1 Review . . . 9
10.2 BFS and DFS . . . 9
10.3 Dijkstra's algorithm . . . 9
10.4 Bellman-Ford algorithm . . . 9
10.5 Floyd-Warshall algorithm . . . 9
10.6 Graph coloring . . . 9

10.7 Minimum spanning tree . . . 9
10.8 Strongly connected components . . . 9
10.9 Topological sort . . . 9
10.10 Lowest common ancestor . . . 9
10.11 Practice problems . . . 9

11 Segment trees **9**
11.1 Introduction . . . 9
11.2 Range sum query . . . 9
11.3 Longest increasing subsequence . . . 9
11.4 Lowest common ancestor . . . 9
11.5 Practice problems . . . 9

12 Miscellaneous **9**
12.1 Sliding window range minimum query . . . 9
12.2 Largest rectangle under histogram . . . 9
12.3 Coordinate compression . . . 9
12.4 Sweep line . . . 9
12.5 Square root decomposition . . . 9
12.6 Meet in the middle . . . 9

13 Maximum flow **9**
13.1 Pipe network . . . 9
13.2 Maximum bipartite matching . . . 9
13.3 Minimum cost maximum flow . . . 9
13.4 More topics?? . . . 9
13.5 Practice problems . . . 9

14 2-SAT **9**
14.1 Binary timetable . . . 9
14.2 TBD . . . 9
14.3 Practice problems . . . 9

15 Nim **9**
15.1 Nim . . . 9
15.2 Nimbers (Grundy numbers) . . . 9
15.3 Practice problems . . . 9

16 Fast Fourier transform **9**
16.1 Introduction . . . 9
16.2 Distinct sums . . . 9
16.3 Hamming distance . . . 9
16.4 Practice problems . . . 9

17 Linear programming **9**
17.1 Introduction . . . 9
17.2 Minimum cost maximum flow . . . 9
17.3 ??? . . . 9
17.4 Practice problems . . . 9

1 Introduction

What is competitive programming?

Competitive programming is an activity in which competitors try to write efficient solutions to programming problems in a timed contest. A contest usually lasts between 2 and 5 hours, and there are usually between 5 and 10 problems to be solved. For each problem, contestants submit a program that reads input and produces output meeting the problem specification.

Some people praise competitive programming as a means of preparing for coding interviews. While it is true that good competitive programmers are almost always good interviewers, competitive programming has much more to offer than the relatively flat world of interviewing. For this reason, I will focus on the contest aspect.

How to read these tutorials

These tutorials provide what I would consider an accelerated introduction to competitive programming. If you read every word and look over all the solutions, you will end up with no more knowledge than you have now. You must participate actively in the learning process.

Each tutorial starts with a motivating problem. I encourage you to read this problem and think about how you would solve it. After working on it for 10-15 minutes, you can read the discussion that follows. In the discussion portion, I attempt to explain the technique and apply it to the motivating problem. As you read this part of the tutorial, you should continue thinking about how you would implement each part of the solution as it is explained. After the full solution is explained, try to code it yourself before looking at my solution.

These motivating problems are often difficult. For example, the problem I use in the backtracking tutorial is not as straightforward as one might expect given its position in the series of tutorials. This is not meant to discourage you, but to challenge you. The topics I cover range from trivial to quite difficult, and if you are going to try to read all of these tutorials in one sitting, you will become hopelessly lost somewhere in the middle. If you actually want to improve, you need to take the chapters one at a time. For each chapter, do all the practice problems listed. Even if you think you already know a topic, do the practice problems anyway–if you know the topic that well, they shouldn't take more than a few minutes!

How to practice

Reading is not enough to become a better competitive programmer. The practice problems listed in each chapter here are a good start: you should read the problem statement, come up with a solution, code the solution, and submit it. If you get wrong answer, don't look at the test case you failed! You won't get that luxury in a competition, and debugging with such limited information is an important skill to learn. Carefully reread your solution and think about edge cases. Write a few test cases and work them out by hand; if you're lucky, you'll find one that your solution fails.

If you can't determine the source of your problem after an hour of debugging, feel free to take a break. Come back in a day or two and see if you can find the bug.

On the other hand, if you read a problem and can't even figure out where to begin, don't panic. Keep thinking about it, and consider how you might apply the technique from the chapter. If you come up with an idea but you're unsure of its correctness, I encourage you to write an informal proof. Especially in greedy problems and at higher levels of competition, the solution is not obvious and a proof may be required to convince yourself that it is correct.

Now, solving all the linked problems is a good start. But if that's all the practice you do, you'll walk into a competition and have no idea where to start. You need to practice solving problems without the context of having just read a tutorial. For this, I recommend going on Codeforces and clicking on random problems (make sure you disable "show tags for unsolved problems" in your settings) and solving them. Other sources of practice problems include UVa online judge, Kattis, SPOJ, HackerRank, CodeChef, and Topcoder.

The last element of good practice is to actually compete. Codeforces holds regular competitions (although some of them are in the middle of the night) which generally have very good problems. HackerRank, CodeChef, and Topcoder are also popular contest platforms.

Acknowledgments

2 Implementation

Motivating problem

Description

Liam was hired by Moogle to help build the Gaps app. Moogle Gaps is an application for finding the length of the shortest path between two locations, using different modes of transportation. Liam has been assigned the part of the application for hot air balloons. Because hot air balloons are not constrained by roads, the shortest distance between two points via hot air balloon is the Euclidian distance between those two points (as X and Y coordinates).

Liam has spent his days watching TouYube videos, and the launch date for Gaps is fast approaching. He needs your help to implement the hot air balloon distance functionality. The app takes as input a list of N locations and Q queries. A location is given as a name (1-20 uppercase letters) and a pair of coordinates. A query asks for the shortest hot air balloon distance between two named locations.

Constraints

$$1 \leq T \leq 5$$
$$1 \leq N \leq 1,000$$
$$1 \leq Q \leq 5,000$$
$$-10,000 \leq X_i \leq 10,000$$
$$-10,000 \leq Y_i \leq 10,000$$

Input

The first line contains T, the number of test cases. T test cases follow.

The first line of each test case contains N and Q. The next N lines contain descriptions of the locations in the app. The ith line contains the name for the ith location as well as X_i and Y_i (which are integers). The next Q lines contain queries. The jth query contains two names of locations.

Output

For each test case, output Q lines. The jth line contains the hot air balloon distance between the two locations in the jth query. An answer is considered correct if its absolute or relative error is less than 10^{-6}.

Sample input

```
1
4 5
VOUNTAINMIEW 45 100
AOSLNGELES -1000 350
YEWNORK 4000 5008
AELTVIV 9999 -4046
VOUNTAINMIEW AOSLNGELES
YEWNORK AELTVIV
VOUNTAINMIEW AELTVIV
AOSLNGELES VOUNTAINMIEW
AOSLNGELES YEWNORK
```

Sample output

```
1074.4882502847577
10861.07347364891
10782.923165821037
1074.4882502847577
6833.517688570068
```

Discussion

An implementation problem is one in which the correct approach is fairly obvious, and the main difficulty is in implementing it. In many competitions, the first problem or two are implementation or brute force (covered in the next tutorial) problems.

Practice problems

- `http://codeforces.com/problemset/problem/614/A`

Further reading

3 Brute force

Motivating problem

Description

Constraints

Input

Output

Sample input

Sample output

Discussion

A brute force problem is one in which the obvious algorithm is good enough; sometimes a more efficient solution is possible but more complex than the brute force approach. When reading a problem, it is often helpful to come up with a brute force approach even if the problem's contraints call for something faster. The brute force solution can serve as a starting point for the final solution. Of course, when the constraints are small enough that the brute force solution should run in time, no more thinking is necessary. Once you identify a problem as brute force, it becomes an implementation problem.

The primary hint that indicates that a problem should be solved with a brute force approach is the input size.

Practice problems

- `http://codeforces.com/problemset/problem/629/A`
- `http://codeforces.com/problemset/problem/631/A`

- http://codeforces.com/problemset/problem/626/A
- http://codeforces.com/problemset/problem/464/B
- http://codeforces.com/problemset/problem/124/B

Further reading

4 Backtracking

Motivating problem

Description

Emma is a graduate student at the University of Texas at Austin, and she has been invited to give talks at N conferences around the world. All of the conferences are taking place at the same time (and are not ending until Emma finishes her talks), and Emma gets to choose when she gives her talk at each conference. Emma loves traveling, but she hates airplanes–unfortunately, the N conferences are spread among M countries, and in order to travel between countries she must fly in a plane. She wants to schedule her talks in an order that minimizes the amount of time she spends in an airplane. Emma can hang glide between conferences in the same country, and she doesn't mind hang gliding; furthermore, she will hang glide to the first conference she attends and back to UT after the last conference.

Because Emma is busy putting together her talks, she wants you to find an optimal order for the N conferences.

Constraints

$1 \leq T \leq 5$
$1 \leq N \leq 100,000$
$1 \leq M \leq 8$
$1 \leq A_i \leq M$
$1 \leq X_j \leq M$
$1 \leq Y_j \leq M$
$1 \leq B_j \leq 1,000$

Input

The first line of the input contains T, the number of test cases. T test cases follow.

The first line of each test case contains N and M. N lines follow. The ith line contains an integer A_i indicating in which country the ith conference is located. After these N lines, $M(M-1)/2$ lines follow. The jth of these lines contains three integers, X_j, Y_j, and B_j. Each unordered pair (X_j, Y_j) appears exactly once. B_j is the time it takes to travel from country X_j to Y_j or vice versa.

Output

Output N lines. The ith line contains the ith conference at which Emma should speak, in order to minimize her total time spent in an airplane. If there are multiple optimal orders, print the lexicographically smallest (where one sequence of numbers S is lexicographically smaller than another sequence of numbers T if the first index at which S and T differ is i and $S_i < T_i$).

Sample input

```
1
5 3
1
2
1
3
3
1 2 10
2 3 50
1 3 100
```

Sample output

```
1
3
2
4
5
```

Discussion

Backtracking is a technique for recursive algorithms that can best be learned by example.

A careful reading of the above problem statement should yield a few key insights:

- Emma wants to visit M countries in some order
- Emma can rearrange the conferences within a country without changing her total airplane time
- We want to choose the lexicographically smallest order of conferences that minimizes total airplane time

Let's see if we can decompose the problem into a few different parts. First, we must choose some order of countries to visit; then, we must choose the order of the conferences in each country. There are $M!$ different orders of countries, and $M \leq 8$, so we can actually just try all of these. Now, given some order of countries, what's the lexicographically smallest way to arrange the conferences? With a little bit of thinking, it should be clear that we should simply sort the conferences within each country. With this insight, we can easily generate the lexicographically smallest answer if we know the correct order of the M countries.

We're almost ready to begin solving the problem! There is one last case to consider: what if two orderings of the countries are tied? That is, what if two (or more) of the $M!$ orderings of countries have the same total airplane time? Obviously, the tiebreaker is lexicographic: we should choose the one that will produce the lexicographically smallest final answer. Keep in mind that we are not choosing the lexicographically smallest ordering of countries, but that of conferences. It is possible that, say, country 1 should be the last country visited because it contains conferences with high numbers.

So, our strategy is this: generate all $M!$ permutations of countries, and keep track of the best one seen so far. At the end (after looking at all permutations), we can expand that best ordering of countries into an ordering of conferences by replacing each country with the sorted list of conferences in that country. I encourage you to try to code this solution on your own.

I won't discuss the implementation in too much detail here, but there are a couple of important points. First, I use an approach called backtracking to generate the permutations recursively. In backtracking, you change the current state temporarily, make a recursive call, and then revert the change. Using this technique, we can avoid making copies of the list of countries. To apply backtracking to the problem of generating permutations, we define a state by the current permutation of countries and the number of countries that have been permuted so far. We can then set the next country in the permutation and recursively generate the permutations for the rest of the list.

For example, say we want to generate all permutations of the list `[1, 2, 3]`. Of course, the current position that we are considering is position 0 (in this recursive call, we are choosing an element to go in the 0th spot in the list). There are 3 things we can put there: 1, 2, or 3. First, let's put 1 there and then recurse on the rest of the list, then put 2 there and recurse, then put 3 there and recurse. In that first recursive call, our current list is still `[1, 2, 3]`, but our current position is now 1. Now we have 2 things we can put at index 1: 2 or 3. This pattern continues. When we reach the end of the list, we've finished a permutation. It will be helpful to draw the tree of recursive calls; you can see that every possible permutation is generated. The following code will generate this tree.

```
def gen_permutations(current_list, current_position):
    print " " * current_position + str(current_list) + " " + str(current_position)
    if current_position == len(current_list):
        print " " * current_position + str("finished permutation: ") + str(current_list)
    else:
        for index in range(current_position, len(current_list)):
            swap(current_list, current_position, index)
            gen_permutations(current_list, current_position + 1)
            swap(current_list, index, current_position)
```

Here's the output of that function on the list `[1, 2, 3]`:

```
[1, 2, 3] 0
 [1, 2, 3] 1
  [1, 2, 3] 2
   [1, 2, 3] 3
   finished permutation: [1, 2, 3]
  [1, 3, 2] 2
   [1, 3, 2] 3
   finished permutation: [1, 3, 2]
 [2, 1, 3] 1
  [2, 1, 3] 2
   [2, 1, 3] 3
   finished permutation: [2, 1, 3]
  [2, 3, 1] 2
   [2, 3, 1] 3
   finished permutation: [2, 3, 1]
 [3, 2, 1] 1
  [3, 2, 1] 2
   [3, 2, 1] 3
   finished permutation: [3, 2, 1]
  [3, 1, 2] 2
   [3, 1, 2] 3
   finished permutation: [3, 1, 2]
```

When we finish a permutation, though, we want to see if it's better than the best one seen so far (that is, whether it has a smaller total airplane time than the current best, or, if it has the same airplane time as the current best, whether it is lexicographically smaller). To do this, we'll keep track of the current permutation's airplane time, as well as the best permutation and its airplane time. The best permutation and its airplane time should be global variables, so that we can update them at one leaf of the call tree and let those changes be seen at a later leaf of the call tree. But everyone knows that global variables are poor style, so instead we'll use a trick to achieve the same effect as pass-by-reference: we'll pass in a list for each global variable, and then just change the elements of the list. I should note that in competitions I usually don't worry about tricks like this and instead just use global variables.

Finally, our recursive function needs to be able to get the minimum conference number for a country in order to decide whether a permutation is lexicographically smaller than another one. When comparing two permutations, we don't need to compare all the conferences, just the first (minimum) for each country. This is because if two permutations have the same minimum conference for a given position, then they have the same country at that position, and all the conferences in that country will match. So, we need to be able to get from a list of countries to a list of their corresponding minimum conferences. To do this, we'll use a list comprehension and a dictionary mapping countries to their minimum conferences (we'll pass this dictionary into our recursive function).

Now we have enough information to assemble our final recursive function.

```python
# Generate all permutations
def gen_permutations(countries, cur_time, min_conferences, best_order, best_time, position):
    # If we're at the end of the list, we've finished a permutation
    if position == len(countries):
        # See if the current permutation (stored in countries) is better than best_order
        if cur_time < best_time[0]:
            # Our airplane time is less than the previous best
            for i in range(len(best_order)):
                best_order[i] = countries[i]
            best_time[0] = cur_time
        elif cur_time == best_time[0]:
            # Our airplane time is the same as the previous best
            # We must check if we have a lex. smaller list of min conferences
            if lex_less([min_conferences[country] for country in countries],
                        [min_conferences[country] for country in best_order]):
                for i in range(len(best_order)):
                    best_order[i] = countries[i]
                best_time[0] = cur_time
    else:
        # This is a recursive way to generate permutations using backtracking
        # For each country we haven't placed yet, put it in the current position and
        #     generate permutations on the rest of the list
        for index in range(position, len(countries)):
            swap(countries, position, index)
            new_time = cur_time + times[(countries[position - 1], countries[position])]
            gen_permutations(countries, new_time, min_conferences,
                             best_order, best_time, position + 1)
            swap(countries, index, position)
```

This is the meat of the implementation. After calling `gen_permutations` with appropriate arguments, the best permutation of countries will be stored in the list object passed in as `best_order`. We can then expand `best_order` by printing the sorted conferences in each country. The full code and some tests can be found in the code directory.

Some readers will recognize this as the traveling salesman problem. The traveling salesman problem is one of the most famous NP-complete problems; there is no known polynomial time algorithm that solves it. There are faster algorithms that take $\mathcal{O}(M^2 \cdot 2^M)$ time instead of $\mathcal{O}(M \cdot M!)$, but those are based on dynamic programming and are still only feasible for input sizes up to about 20.

Practice problems

Further reading

- `https://en.wikibooks.org/wiki/Algorithms/Backtracking`
- `https://www.cis.upenn.edu/~matuszek/cit594-2012/Pages/backtracking.html`
- `http://moritz.faui2k3.org/en/backtracking`

These resources discuss backtracking as a means of searching for a goal state. In this problem, we use the same technique, but we use it to explore *all* states and take the best leaf node found.

5 Binary search

Motivating problem

Description

Constraints

Input

Output

Sample input

Sample output

Discussion

Practice problems

Further reading

6 Divide and conquer

Motivating problem

Description

Constraints

Input

Output

Sample input

Sample output

Discussion

Practice problems

Further reading

7 Greedy

Motivating problem

Description

Constraints

Input

Output