Getting Started

This chapter introduces the basics of Marathon Matches.

Understanding the Marathon Competition Format

Problem

Individual matches in Marathon Competitions are called Marathon Matches (abbreviated to MMs), or simply Marathons. Prior to competing in one, one has to understand its structure and requirements.

Solution

TopCoder Marathon Competition has a specific format which is somewhat similar to other long-term on-line contests but still requires some insight.

Each MM presents the competitors only one problem. Unlike SRM problems, MM problems can not be solved exactly – either because there is no exact solution as such, or it is considered impossible to find within given time constraints. The task of each MM is to find a solution as close to the optimal one as possible; the quality of the solution is estimated using a certain (problem-specific) scoring system which is described as a part of problem statement. Score depends only on how good your solution performs and never on how many time has passed between reading the statement and submitting the solution.

From technical point of view, MM consists of two phases – Submission and System Testing. Submission phase is the time (a rather long one – usually two weeks, sometimes one or four weeks) to solve and submit the given problem. You can register for the match at any time during Submission phase; the problem statement is available even to unregistered members, so you can decide whether you will take part in a particular match after studying the problem.

After you read the problem statement, figure out a decent way to solve it and code the solution (all done off-line), it's time to submit your solution. You can do two types of submits: Example Test and Full Submission.

Example Test runs your code on a small (usually 10) predefined set of test cases and provides you detailed information on the results: the raw score (the individual score you've received on each test case, without aggregation in overall score), the runtime, error messages if the test case crashed and debug output if your solution generates it. This is somewhat similar to testing your SRM solution on examples before submitting it, except that you can't choose the test cases to run. You can do Example Test at most once per 15 minutes.

Full Submission (also referred to as provisional testing) runs your code on a larger (usually 100) predefined set of test cases and provides you only the overall

score of this submission, without per-test-case breakout or score. You can do Full Submission at most once per 2 hours.

Current rank-list of the contest is built based on last Full Submissions of all participants. A person is considered to be a participant of the match not if they have registered for it but if they have done Example Test or Full Submission. For some relative scoring systems overall scores of all participants are recalculated after each Full Submission. Example Tests don't affect standings; if you have done only Example Tests in the match, for the purposes of ratings calculations you are considered to be ranked lower than the people who scored zero on Full submissions.

After the Submission phase ends, System Testing (or final testing) starts. In this phase the latest Full Submissions of each competitor are tested against even larger set of test cases, and scores on these cases form final standings, used as match results, for rating calculations etc.

Discussion

Marathon Matches use a format common for long-term contests on hard problems. A few contests similar to typical Marathon tasks are: Al Zimmermann's Programming Contests and Google AI Challenge, and occasional data-mining contests are similar to special Marathon events.

Marathon Matches don't have a tight schedule like SRMs, so they needn't a refined strategy – basically it's just "solve and submit" at your own pace. However, there are some subtle things which should be stressed to avoid common but painful mistakes.

Example Test isn't an actual submit – if you do it but don't do Full Submission, you won't get a final score, so you'll be ranked even lower than people with zero score, with corresponding rating decrease.

Remember about the interval between Full Submissions, and plan last day of competition accordingly. Thus, it is strongly recommended not to postpone your last submission till last hours – a small bug which causes it to crash might leave you without time to resubmit and poor total score. You should leave a few hours for a desperate resubmission, as well as run Example Test before Full Submission – since the intervals between Example Tests and Full Submissions are independent, you'll save a lot of time by checking the validity of your new submission before fully submitting it.

The time management of the match depends heavily on schedule of your daily life during the match. You don't have to stop all activities to do good in a Marathon – sometimes it's enough to get a great idea and a few days to implement it. However, usually Marathon Matches are associated with long-term devotion to solving single problem.

It is important to read and re-read problem statement before starting to code, to make sure you understand what you have to do. Play with the visualizer if you prefer visual information over textual one. Submit in the contest only when you're 100% sure that you'll have enough time and inspiration to take part in the match.

To build a strategy for a particular match, you have to answer several questions as early in the match as possible:

- How much time you're willing to spend on this problem?
- What is the purpose of participation – win, increase your rating, win a t-shirt (if there is one for participation or top-something placement), have fun playing with the problem?
- Do you need to do any research before approaching the problem? If you do, how much time you'll spend on it?
- How many different approaches you'll try?
- What are fun parts of the problem? You might like the problem so much that you decide to take part regardless of other factors.
- What is the variance of the problem and the level of randomness in the results? In some problems the scoring or the input data is developed in a way which makes final rank-list differ a lot from provisional one, sometimes in a pretty random way. Some people prefer to avoid such problems, since bad luck can spoil the performance in final testing even for a good solution.

Sidebar: Understanding Specialized Marathon Matches

From time to time TopCoder hosts specialized Marathon Matches – matches which aim either to solve some real world problem or to test/promote new hardware or software platform. Such matches usually have sponsors (the company which needs this problem solved or platform played with) and monetary prizes, and sometimes they require specific skills or access to specific hardware. Note that sponsors and prizes alone are not a criterion – regular Marathons can have them as well, like NSA series.

Specialized matches come either as part of series or on their own. The series which took place so far were:

- *Intel Multi-Threading Competition* consisted of 12 matches which ran in 2006, one match per month. This series allowed usage of multi-threading and encouraged it, since the problems were suited for parallel programming. Submissions were tested on a dedicated Intel multi-processor server, and the only language allowed was Intel C++.

- *AMD Multicore Threadfest* consisted of 4 matches which ran in 2008. This was another multi-threading series, with simulation and image-processing related problems. The only language allowed was C++.
- *CUDA Superhero Challenge* consisted of Beta contest and 2 main matches and ran in 2009. Yet another multi-threading series used nVidia CUDA API to run code on a graphics processing unit. Submissions were tested on a dedicated server with powerful CUDA-enabled GPU.
- *NASA-TopCoder* is the only ongoing series, which aims to solve real-life problems in an innovative way. It started in November 2009 with SpaceMedkit problem, which required designing optimal medical kit for space missions. The results of the match were so impressive that in 2011 NASA Tournament Lab was created to run matches which solve NASA tasks. First problem was VehicleRecognition which focused on classifying aerial photos with respect to whether they contained vehicles on them. The second one was CraterDetection, also an image processing problem, which goal was to detect craters in a given set of orbital images taken under various conditions. This series didn't restrict the language used.
- *Harvard Business School* experimental series was a special one – it focused not only on solving the given task but also on studying individual against team problem solving. The contestants were divided in groups, and solved the task either individually or in teams depending on the group type. First match of the series was SequenceAlignment in April 2009, which solved common bio-informatics task of same name. Second match was SpaceMedkit, shared with NASA series.

All individual specialized matches focus on real-life tasks:

- *Predictive Marathon Competition 1* (June 2008) required to predict the outcomes of TopCoder Component competitions based on large set of real historical data for these competitions.
- *FundingPrediction* (February 2009) predicted outcome of loan funding based on training data set of real-life data about loans over a period of two years.
- *Linden Lab OpenJPEG* (February 2009) aimed to speed up decoding of "JPEG 2000" format in open source OpenJPEG library to use in the "Second Life" virtual world.
- *AgentMatching* (August 2009) was similar to FundingPrediction and predicted outcomes of real estate deals.
- *MessageDispatcher* (September 2009) simulated a real-life message dispatching system which processed huge quantities of messages in a most efficient way.
- *OrdinalTraitAssociationMapping* (August 2011) required to determine DNA markers associated with a trait, given DNA marker genotype data for a

large number of individuals

Dissecting a Problem Statement

Problem

TopCoder Marathon problems have a specific format of statement. Being able to grasp the essence of the problem fast is not so important as in SRMs, since Marathons allow to spend much more time on the problem. However, the statements tend to be larger than in SRM problems, and it's still important to understand the structure of the statement to be more comfortable with the problem.

Solution

The problem statement consists of a set of sections, given always in the same order. Unlike SRMs, some of the sections are optional and can be omitted depending on the specifics of the problem and the writer's preferences.

Statement

The main part of the problem statement. Since it can be quite long, most writers break it up into several smaller parts:

- Introduction is first paragraph or two give an informal introduction to the problem – its background, a general idea of what it's about and what the task will be.
- Implementation explains in detail what methods your code must implement in order to create a valid submission: the format and meaning of input parameters and return of each method, the purpose of each method and the order in which they will be called.
- Scoring explains how the return of your solution will be evaluated – both how scores on individual test cases are calculated and how the overall score is constructed from individual scores. See recipe "Understanding Absolute and Relative Scoring" for discussion of scoring schemes.
- Test Case Generation describes the process used to generate the test cases. Sometimes it describes generation of each input parameter of each method in detail, and sometimes it gives a high-level scheme of the process and refers to visualizer code for details of implementation. You can usually skip this section and still submit a valid solution, but reading it might give you an advantage of understanding some details which might be vital for success.
- Visualizer is a standalone program that allows you to test your solution locally at your computer and get the same scores that TopCoder server would produce. Java source code of the visualizer is also provided, and this

is extremely important because that is the exact and complete definition of the problem expressed in a formal language. If you have any doubt about the problem statement or you want to know minor details of the process which are omitted in the problem statement, the source code of the visualizer will give the answer. Visualizer manual is usually separated from problem statement.

Note that the division into subsections is optional – some writers don't use it, and some problem statements are too short for further breaking.

Definition

This part provides the signatures of class and methods you'll have to submit. Problem statement is shown using your default language, which can be set in the Arena or on the code submission page using the website. This part, as well as the next one, is auto-generated.

Available Libraries

Specifies the signatures of library methods you'll have to use in some problems. These are part of interactive problems, in which you are not just given the data to process but rather have to choose which data you'll need – the data you're given depends on your actions. Basically you'll have to call library methods, the tester will generate its return and give it to you for further processing. The methods themselves and the context of their usage are described in Introduction.

Notes

Things the writer wanted to mention but couldn't fit in Statement. This part also contains notes about the testing process – memory limit, time limit, code size limit, numbers of test cases etc.

Constraints

The constraints on the input parameters. This part is usually shorter than in SRMs and give less detail, since most things about input parameters are described in Test Case Generation section. Other than that, they are the same as in SRMs.

Examples

Provides the list of test cases which will be used in example testing, usually as seeds and some core parameters of the problem. You can generate these examples by using the visualizer with the seeds indicated in each example. Other than that, this section is quite useless, since the examples are not annotated and usually don't give full information about the test cases.

Discussion

Now let's have a closer look at how this structure works in a real problem Planarity. Introduction is very short and gives the context of the problem:

You are given a graph to be drawn on the plane. All edges of the graph are drawn as straight lines. Your task is to arrange the vertices of the graph so that the number of intersecting pairs of edges is minimized.

This problem is simple enough, so this paragraph gives not only the context but also the whole problem, even if in a rather informal way – more formal specification will follow. For more complex problems, like StreetSales, the introduction to the problem is longer and provides not only the context but also vital information about the trading scheme used in the problem. Generally after reading Introduction you will know what you have to do, and the rest of the problem will provide the details on how to do this.

Planarity is a fictional problem; sometimes problems are taken from real life – in such cases their real backgrounds can be given to provide contestants extra motivation for solving them; see sidebar "Understanding Specific Marathon Matches" for examples.

Next part is Implementation, with descriptions of methods, their parameters and return. Planarity has only one method to implement, so it's rather short as well:

Your code should implement one method **untangle**(int V, vector edges). The parameters of this method describe the graph in the following way: **V** is the number of vertices in the graph, and (**edges[2*j]**, **edges[2*j+1]**) are the indices of vertices which form j-th edge. You have to return a vector which contains the coordinates of the vertices. Elements $2i$ *and (2*i+1) of your return should contain the x- and y-coordinates of i-th vertex, respectively.

Note that Implementation matches Definition exactly, it just explains the methods in more details. Now you know how your solution must work from tester's perspective. For problems with more methods to implement (like StreetSales) or with library methods available (like ReliefMap) Implementation will be longer, but it will always match Definition and Available Libraries sections.

Next part is Test Case Generation; in Planarity it's full but relatively short, so it doesn't refer the reader to visualizer source. In some problems test cases are generated simply as a sequence of values sampled from specific probability distributions (usually uniform); for examples, see CellularAutomaton or EnclosingCircles. In most problems, however, there will be some more complex algorithm involved; see, for example, Textures, StreetSales, ReliefMap or nearly any other problem.

Scoring section follows; in Planarity it's trivial, but sometimes it can be the most massive part of the problem. For example, in CellularAutomaton all you have to do is return a new initial configuration of the automaton, but to score your return, the automaton evolution has to be followed for several steps. Same goes for BounceOff – you return a set of obstacles to be placed, and the score is based on the results of simulation of ball movement.

As usual in more recent problems, Planarity visualizer is moved outside of the

problem statement, to a separate page. It has links to visualizer .jar file and source code .java file, a pseudocode of what your program has to do to interact with it and a description of commands and options used to run visualizer.

In Notes you can find all kinds of things which didn't fit in the Statement:

- problem statement clarifications: KnightsMoveCipher, Epidemic;
- scoring of invalid return and details of test case generation (if not mentioned in corresponding sections): MegaParty;
- external sources used to generate test cases: KnightsMoveCipher, Epidemic, OneTimePad.

Compared to SRMs problems, Marathons statements tends to be less formal and to rely on visualizer for details; Notes are more about organization of the testing process than about problem clarifications, Constraints give less details, and Examples tend to be pretty useless without the visualizer.

A final recommendation: be sure to read the problem statement very carefully – it seems an obvious thing to do but this is an important step to solving the correct problem in a correct way. Usually revisiting the problem statement page in a later stage of the competition is a good idea, since something that looked like a minor detail at first could be used to improve your solution.

Submitting Your Solution

Problem

In the previous recipe we have reviewed the structure of Marathon problem statement. Once you've figured out what the problem is and how you are going to solve it, you have to actually code the solution and submit it. Same as SRMs, Marathons require the submitted code to be in specific format.

Solution

You can submit for a Marathon (as well as register for the match and check current standings) both on the website (Competitions -> Marathon Match -> Active Contests) and in Arena (Active Contests -> current match).

Marathons allow you to choose among the same four languages as SRMs – C++, Java, C# and VB.NET – and add Python as an option. For some ideas on which language to choose, see sidebar "Choosing the Right Programming Language". If you're using Arena, Definition and Available Libraries will change accordingly to your chosen language, so you will be able to see what data types to use and how the declare the required methods. On the website you can change your preferred language in the submission page.

With respect to solution format, Marathon problems are somewhat similar to SRM ones. You must submit implementation of one class, as specified in Definition part of problem statement. Unlike SRM problems, the implementation might require multiple public methods to function properly. In this case, problem statement will specify the order in which they will be called. Your code can also contain any accessory classes, but all code must be submitted in a single file, so, for example, in Java they can not be declared as public. Any supporting data structures you need can be included in the file either as global variables (if your language allows them) or as class variables. Note that each test case is tested in a separate process using one instance of the class, so you can use global variables and class variables for storing data between method calls within one test, but not between different tests.

Another difference from SRMs is that in some problems you have to use "libraries" provided by the problem environment – methods which give you extra information about the problem. Their signatures are also given in problem statement, in section "Available libraries".

The return from your solution is evaluated in a different way than in SRMs: instead of just checking it for being identical with a certain correct answer, it is evaluated with respect to how good it solves the given problem. In some cases this is done by comparing with "perfect" answer, and the score is the measure of similarity to it; however, in most cases the quality of your return is evaluated on its own.

The solution has limitations for its execution time and memory consumption, which vary from problem to problem and are specified in the statement. Your code doesn't have to be readable, but some problems impose a limitation on its size.

Discussion

Here are skeleton submissions for problemCellularAutomaton in all available languages. The basic solution for this problem is very simple – you are required to return some initial configuration of the automaton, and the easiest way to do this is to return the configuration you received as parameter.

C++

```
#include <string>
#include <vector>

using namespace std;

class CellularAutomaton {
public:
```

```
    vector<string> configure(vector<string> grid, string rules, int N, int K) {
        return grid;
    }
};
```

Java

```
public class CellularAutomaton {

    public String[] configure(String[] grid, String rules, int N, int K) {

        return grid;

    }

}
```

C#

```
using System;

public class CellularAutomaton {

    public string[] configure(string[] grid, string rules, int N, int K) {
        return grid;
    }

}
```

VB.NET

```
Public Class CellularAutomaton

    Public Function configure(ByVal grid As String(), ByVal rules As String, _
         ByVal N As Integer, ByVal K As Integer) As String()

        Return grid

    End Function

End Class
```

Python

```
class CellularAutomaton:
```

```
def configure(self, grid, rules, N, K):

    return grid
```

Sidebar: Choosing The Right Programming Language

TopCoder Marathon competitions support five programming languages: C++, C#, Java, Visual Basic.NET and Python. What language to choose as main language for competitions, and when it's worth to temporary switch language for particular marathon match?

First, let's look at the stats for last 25 marathon matches (Marathon Match 45 – Marathon Match 72, only regular marathon matches were considered, and matches 57, 66 and 70 were skipped because they have too many first-placed submissions).

	C++	Java	C#	VB.NET	Python
1-st places	14	7	3	1	0
Top 3	45	20	8	2	0
Top 10	160	62	25	3	0
F1 score	1589	644	251	41	0

The last row is score according to the recent Formula One World Championship points scoring system.

Of course, the results are biased towards more popular languages, but they probably gained their popularity for a reason.

Although generally Python is more popular (has more submissions) than VB.NET, it earns zero points. The main reason for that is that, unlike TopCoder SRM competitions, in Marathon Matches speed of a solution does play a huge role, and Python is the only interpreted language from five supported languages (besides, TopCoder doesn't support Psyco – a Python extension module which can greatly speed up the execution of a program). So even though Python is officially supported in Marathon competitions, it's really not an option if you want to win or take a good place (unless your algorithm is really superb, but there is no historical evidence of such case and you probably would have even bigger advantage using faster language). If you are Python expert, it might be handy for you to write a prototype program in Python and then rewrite it in a faster language.

In the above table C++ language has more points than all other languages added together. The main reason is speed again: C++ is considered the fastest of all five supported languages (and usually you can use MMX, SSE and inline assembler to speed up it even more). So in the long run C++ is probably the

language of choice if you want to win in TopCoder Marathon Matches.

Unlike SRMs, particular features of some language like arbitrary precision numbers or regular expression support in Java will not give a huge advantage, because Marathon Matches are long enough to implement needed piece of functionality in any language (although it can be handy to have it in standard library of a language). Java has another minor advantage – in most Marathon Matches pieces of visualizer code (provided as part of the problem) can be reused because it is written in Java.

On the other hand, Marathon Matches programs are much larger and much more complex compared to SRM solutions, and are more like real life programming. C# and Java are more high-level languages than C++, so they can help to manage complexity better because of higher level of abstraction (and TopCoder doesn't support popular Boost C++ Libraries that extend the functionality of C++). So some problems – the ones which require more complex implementation and are not very computation-heavy – might favor using Java or C# over C++.

As for non-standard, specific Marathon Matches, the above considerations also apply, but choices are usually more limited. Many specific Marathons require to use C/C++ only (Intel Multi-Threading Competition series, AMD Multicore Threadfest, Linden Lab OpenJPEG).

To summarize::

- you can't really use Python for your final submissions if you want to take high place;
- you can use C/C++ in every Marathon Match (including special format contests);
- you might want to use more high level languages like Java or C# for some problems that require complex implementation and are not very computation-heavy.

Also when considering different languages find in the rules or ask in the forums of the particular contest about what specific versions of compilers/interpreters server uses, in that environment and with what options, what libraries are allowed. TopCoder system doesn't support the latest versions of the allowed languages, as well as some popular libraries, and you have to take this into account when making your choice.

Understanding Marathon Scoring Systems

Problem

Marathon problems are typically impossible to solve exactly under the given constraints. The participants' submissions are scored based on their efficiency in solving the given problem. The scoring schemes used to do this vary from

match to match, and can sometimes be quite complicated. Understanding them is important to be able to focus on the best strategy for the match. This recipe explains the way scoring is done in Marathons, and examines the common scoring schemes.

Solution

For each problem there is a set of test cases on which all solutions are tested. The performance of the solution on each test case is estimated with a numeric value called "individual score". After the solution is tested on all test cases in the set, its individual scores are accumulated to get a measure of how good the solution is compared to solutions of other competitors – another numeric value called "overall score".

The methods of calculating both individual and overall scores for each problem are defined in "Scoring" section of the problem statement. Individual scores are a measure of how good is the solution itself on each test case, and the way they are calculated depends strongly on the nature of the problem. However, there are two basic types of ways to calculate overall scores:

1. Absolute scoring. The contribution of a test case to overall score of the solution depends only on its score for this test case. A typical example of absolute scoring is calculating overall score of the submission as a sum or average of its individual scores for all test cases.
2. Relative scoring. The contribution of a test case to overall score of the submission depends not only on its score for this test case, but also on the scores the other submissions got for this test case.

Discussion

Absolute scoring is usually used when individual scores are already normalized, i.e., the maximal possible score for each test case is known beforehand.

For example, in crypto matches XORPlusEncryption, OneTimePad and KnightsMoveCipher individual score is basically the percentage of correct characters in the decoded message, which is normalized to lie in [0..1] range naturally. In BrokenClayTile the score is the percentage of tile pixels guessed correctly. In Klondike individual score is 1 if the game was completed successfully, and 0 otherwise.

Another case of absolute scoring is applied when the maximal possible score for a test case is unknown beforehand, but the task is to minimize the score. In this case the absolute score is the sum of inverses of individual scores (possibly multiplied by a constant), like in SequenceAlignment and J2KDecode.

Relative scoring is usually used when there is no simple estimate of how large or how small the individual scores can get (see, for example, ContinuousSameGame

or FactoryManager). The usual purposes of relative scoring are to make all test cases have approximately even weights in the overall score and to prevent the overall score from growing too large.

The most popular cases of relative scoring define the contribution of individual test case in the overall score as either YOUR/MAX or MIN/YOUR, where YOUR is the solution's score on this test case, and MAX (MIN) is the maximal (minimal) score achieved by anyone on this test case.

Other scoring schemes that are appropriate for specific contests may be adopted, which can use more complicated math formulas which include mean and standard deviation of top 20 scores on this test case (see ContinuousSameGame). The most exotic method of relative scoring so far was used in problem Navigator, in which each test case added MIN/YOUR to overall score of submission, but only if this submission touched the maximal number of waypoints (or tied with some other submission to do this).

A special case of relative scoring is so-called ranking scoring, which calculates the contribution of a test case based on the number of submissions beaten by this one or tied with it on this test case (see TilesMatching or Planarity. It gives less information about the relative performance of your solution against others, since there is no way to figure out by how much on average you beat them, or how much better you have to do to catch up with leaders – you know just that you have to do better.

Note that 0 is usually a special score which represents some kind of submission failure (timeout, invalid format of return, crash etc.), so all scoring schemes disallow it to contribute towards the overall score.

With absolute scoring, it's your absolute improvement on each case that matters, while with relative scoring it's usually your percent improvement. Most people tend to like absolute scoring, mainly because with it the competitor's score depends only on his submission, and not on the other submissions, and it's much easier to track down your overall progress.

With relative scoring, overall scores of all competitors are recalculated after each submission. The non-evident part of this scheme is that only the last submission of each competitor is taken into account; maximal or minimal scores achieved by previous submissions don't matter, and the overall scores of previous submissions (can be seen in competitor's "Submission history") are not updated. This way your overall score changes frequently (unfortunately, mostly decreases) even if you don't submit a new solution. A good idea is either to compare your solutions locally (see recipes "Comparing Solutions" and "Keeping Track of Your Progress") or at least save your old submission's score before submitting the new one and compare the results immediately.

Absolute scoring is usually cleaner and easier to use as a measure of one's progress. In fact, the only people who really like relative scoring are problem writers, for the main reason that it doesn't require inventing a way to normalize

individual scores.