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.

Working on the Problem

Measuring Time

Problem

Marathon matches typically have relatively large time limits for program's running time - between 10 and 30 seconds per test case. The problems are computationally hard and impossible to solve optimally even within these generous time limits, so it is often a good idea to design your solution to use up all available runtime trying to improve its return value.

Your program gets no time warnings from the server, so it has to manage its timing itself. To do this, you have to be able to measure the time elapsed accurately and in a way which is as close to server's way of time measurement as possible. In this recipe we show how this can be done and discuss some popular pitfalls that come along the way.

Solution

To measure elapsed time, you have to use one of the standard functions of your language that return current time. Call it once before running your algorithm, and a second time when you need to know the elapsed time. Take the difference between measurements to determine the running time of the algorithm (or its part in case of several measurements).

Following are the code snippets which show the usage of time measuring functions in languages allowed at TopCoder. Each of them defines a function getTime() that returns current time in seconds.

C++

```
#include <sys/time.h>

double getTime() {
   timeval tv;
   gettimeofday(&tv, 0);
   return tv.tv_sec + tv.tv_usec * 1e-6;
}
```

Java

```
double getTime() {
```

```
   return 0.001*System.currentTimeMillis();
}
```

C#

```
double getTime() {
   return 0.001*DateTime.Now.TotalMilliseconds;
}
```

VB.NET

```
Function getTime As Double
   Return 0.001*Date.Now.TotalMilliseconds
End Function
```

Discussion

There are some general advices to be taken into account if your solution relies heavily on time measurement.

- Remember that testing system measures total time elapsed while your code was running (so-called wall clock time), not CPU time used by it. Functions like clock() in C++ or time.clock() in Python ignore the time used by other processes, so using them will result in exceeding time limit.
- Submissions are tested in sandboxed environment, so all system calls (including the calls of time measuring functions) are inspected by the testing system. This makes them slow - much slower than on your system - so calling them too often can slow down your solution significantly and rob the algorithmic part of the precious time. Thus, for example, if your solution does a lot of small optimizations in a loop (for example, in hill climbing or simulated annealing algorithms), you might consider calling getTime() not each iteration but every 10th or 100th iteration (depending on how long one iteration takes).
- All time measurement functions have a certain accuracy which differs from their precision. Thus, System.currentTimeMillis() in Java gives accuracy of about 10 milliseconds. Don't rely too much on time measurement being precise, though; remember that running your solution on TopCoder server has overhead you can't measure, like instantiating the class and passing parameters to the method. In recent matches the time spent on reading your return is not calculated towards the total runtime of your solution, but some of the overhead is still there. My personal preference is to play safe by leaving 0.5s of unmeasured time for such things. After all, changes which are done in last 20% or less of the time are usually minor, and can't

cover the harm of possibly hitting time limit and losing the score for that test case totally.

Local testing of solutions which rely heavily on time measurement has some extra pitfalls in it. First, TopCoder servers measure wall clock time, but they are focused on running the solution alone. Your computer most likely has other things running simultaneously with your solution - background music, instant messengers, open browser and IDE etc. This means you'd better measure CPU time used by your process alone.

Another pitfall could be the operating system you're using, if it differs from the one used by TopCoder. The servers which test C++, Java and Python solutions use Linux, while the servers which test C# and VB.NET use Windows. If you're using Windows for competing in C++, for example, gettimeofday() won't work for you locally. C++ allows to handle this in a way which will work both on TC servers and on your system without modifying the code: use #define and #ifdef compiler directives to distinguish between local run and TC testing system.

```
#ifdef LOCAL

#include <time.h>

#else

#include <sys/time.h>

#endif


double getTime() {

#ifdef LOCAL

   return (clock()/CLK_TCK);             //CPU time on Windows

#else

   timeval t;
   gettimeofday(&t,NULL);
   return t.tv_sec + t.tv_usec * 1e-6;  //wall clock time on Linux

#endif

}
```

Of course, there are other time measuring routines you might prefer. Thus, C# has Stopwatch class:

```
using System.Diagnostics;

Stopwatch sw = new Stopwatch();

sw.Start();                 //start countdown

//...

time = sw.ElapsedMilliseconds;  //get the time elapsed since the call of Start()
```

In Java you can use System.nanoTime() which gives nanoseconds precision (but not accuracy!), though I don't think anybody would need that kind of precision in a Marathon match.

In C++ you can use inline ASM to measure time via RDTSC (read time-stamp counter) instruction, but it's a low-level utility and the results might vary depending on the processor frequency of your computer vs TC system. You'll have to do some kind of scaling if you want to use it:

```
double getTime() {

   unsigned long long time;

   __asm__ volatile ("rdtsc" : "=A" (time));

#ifdef LOCAL

   return time / LOCAL_SCALE;

#else

   return time / TC_SCALE;

#endif

}
```

Running the Visualizer with Your Solution

Problem

You have coded a solution for a Marathon problem, and you want to test it locally using the provided tester before submitting it to the server.

Solution

Typically each Marathon problem provides an offline tester - a tool which allows to generate test cases, run your solution on them and score the results without submitting it to TopCoder servers. If the problem allows, the tester provides a visualization of the input data and the return of your solution - an image or a dynamic drawing, sometimes interactive; that's why the tester is usually referred to as the visualizer. Using the tester you can gain more insight in the problem and get more information about the performance of your solution.

The solution you submit to the server contains only the implementation of the required class. To run it with the visualizer, you have to add the code which will interact with the visualizer, namely read the parameters generated by the visualizer from standard input, create an instance of the class, pass the parameters to the corresponding method of the class, receive the results and write them to standard output.

In some problems the scheme is more complicated: the required methods of the class are called iteratively until a certain condition is satisfied. Since this condition is usually checked at visualizer side, it's enough for your code to read the parameters from stdin and write the results of processing them to stdout in an infinite loop. Once the visualizer has finished receiving data from your solution, it will halt the solution.

Some problems provide library methods available on the server, and using them locally requires additional code which imitates calling these methods and receiving their return via standard output and standard input of your solution.

The exact order of actions is described in pseudocode in the visualizer manual, but you have to implement them in language of your choice.

Discussion

To run the visualizer with your solution, you should usually run:

java -jar *jarname* -exec *command* -seed *seed*

Here *jarname* is the name of provided .jar file which contains the visualizer, seed is seed for test case generation, and *command* is the command which runs your solution.

If your solution is written in Java and stored in file YourSol.java, you have to compile it to YourSol.class, and use "java YourSol" as *command*. For other languages, compile your solution into an executable file and use its name as *command*.

Running the tester on a certain seed will always generate the same test case, so setting it is a convenient way to check the efficiency of different approaches on the same data. The only exception to this is seed 0, which generates a new test case each time. Test cases generated with seeds 1 through 10 are typically used as example tests (this will be given in the problem statement), so you can use them to compare the results of local testing and testing on the servers. For

massive local testing you can choose any set of seeds and stick to it whenever you test new solution.

Some visualizers provide additional options which tune the way the visualization is done, but these options depend on the specific problem and are listed in the visualizer manual. Usually it is possible to turn visualization off; for some game-based problems the visualizer might allow to play the game yourself, i.e., provide the data for it via visual interface.

There are a few things which should be stressed before and apart from showing them in the examples:

- flush streams each time you have finished writing data to them, otherwise the visualizer doesn't start reading from them and becomes not responsive.
- when you read data from input stream, use a method which reads the end-of-line character from the stream, otherwise you might get some weird values for next variables.
- don't leave any unread data in standard input of your solution, even if you don't need it at the moment, otherwise the visualizer might be unable to read the contents of its standard output.
- generally, if you're not yet comfortable with using the visualizer, perform the data exchange between it and your solution exactly as described in the manual.

There are three basic ways of visualizer-solution interaction.

1. The visualizer calls the solution's method once.

In this case your interaction code has to read the parameters from stdin, process them, print the result to stdout and flush stdout. Example of such problem is BrokenClayTile. The pseudocode given in the visualizer description for this problem is typical and easy to follow:

```
S = int(readLine())

N = int(readLine())

P = int(readLine())

for (i=0; i<P; i++)

    pieces[i] = readLine()

ret = reconstruct(S, N, pieces)

for (i=0; i<S; i++)

    printLine(ret[i])
```

```
flush(stdout)
```

2. The visualizer calls the solution's method (or several methods) several times.

This is typical for game-based or simulation-based problems, in which the task is done in several steps, and information necessary for taking a decision about the next step is known only after the previous step is completed. The number of calls is usually unknown beforehand and is limited either with a constant or with some condition which must be fulfilled in order to stop the simulation. Thus, for example, in problem ChessPuzzle method click is called for as long as there are valid moves left, at most KRC times.

Since it's the visualizer who decides to stop simulation, for your interaction code it's enough to run an infinite loop which reads the parameters, processes them and prints the result. Once the simulation is over, the visualizer will halt your running solution.

Usually this approach is combined with a single call to another method which gives initial parameters of simulation or parameters that stay the same during all simulation (in ChesssPuzzle it's start). This initial call should be handled as described in previous paragraph. If one of several methods is called depending on the outcome of the completed step, the visualizer will usually let know which one is called (via standard input as well), so your loop will have to recognize this and call the correct method.

The pseudocode for problem ChessPuzzle is as follows:

```
K = int(readLine())

R = int(readLine())

for (i=0; i<R; i++)

    board[i] = readLine()

printLine(start(K, board))

flush(stdout)

while (true)

    revealed = readLine()

    printLine(click(revealed))

    flush(stdout)
```

3. The visualizer calls the solution's method once and provides a library method which solution should call to get necessary information.

This interaction method is the trickiest to implement in the visualizer. It is used only if the other two methods are too unnatural for the problem. In this case, once the visualizer has provided the initial parameters, it waits for the solution's return. If it is a predefined constant (usually "?"), the visualizer interprets the next portion of data as the parameters of the library method to be called; otherwise it assumes that this is the final return of the solution. This is kind of a reverse of the second way - it makes the solution call the visualizer's method, instead of having the visualizer call solution's method.

Now let's have a closer look at how to write interaction code, using problem ReliefMap as an example. It provides a library method Relief.measure(x,y) and thus uses third way of visualizer-solution interaction. The interaction pseudocode for the problem looks like this:

```
double measure(x, y)

{   printLine('?')

    printLine(x)

    printLine(y)

    flush(stdout)

    return double(readLine())

}

main

{   H = int(readLine())

    for (i=0; i<H; i++)

        contourMap[i] = readLine()

    ret = getMap(contourMap)

    printLine('!')

    for (i=0; i<W*H; i++)

        printLine(ret[i])
```

```
        flush(stdout)

    }
```

Here are the codes which allow the solution in any language be tested with the visualizer. "The solution itself" is the solution which can be submitted to TopCoder's server with little or no modification (this is convenient, since such modifications are a source of errors). Two other parts perform the data exchange with the visualizer. "Imitation of provided library class" implements a simulation of the library class with a method your solution will call; this method writes the call with its parameters to standard out and reads the visualizer's response from standard in. "Main interaction code" is code which calls the solution's method and returns its result to the visualizer.

C++

```
/* ----- imitation of provided library class ----- */

#include <iostream>

using namespace std;

class Relief {

   public:

   static double measure(int x, int y) {

        cout<<"?"<<endl<<x<<endl<<y<<endl;

        cout.flush();

        double ret;

        cin>>ret;

        return ret;

   }

};

/* ----- the solution itself ----- */

#include <vector>
```

```
#include <string>

using namespace std;

class ReliefMap {

   public:

   vector<double> getMap(vector<string> contourMap) {

       /* your code here */

   }

};

/* ----- main interaction code ----- */

int main(int argc, char* argv[])

{   vector<string> contourMap;

   int H;

   cin>>H;

   for (int i=0; i<H; i++)

   {   string t;

       cin>>t;

       contourMap.push_back(t);

   }

   ReliefMap rm;

   vector<double> ret = rm.getMap(contourMap);

   cout<<"!"<<endl;

   for (int i=0; i<ret.size(); i++)

       cout<<ret[i]<<endl;
```

```
    cout.flush();
    return 0;
}
```

Java

```
/* ----- imitation of provided library class ----- */
import java.io.*;
class Relief {
    public static double measure(int x, int y) {
      try {
        System.out.println("?\n"+x+"\n"+y);
        System.out.flush();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        return Double.parseDouble(br.readLine());
      }
      catch (Exception e) {}
    }
}
/* ----- the solution itself ----- */
public class ReliefMap {
    public double[] getMap(String[] contourMap) {
        /* your code here */
    }
/* ----- main interaction code - can be added to the solution class ----- */
```

```
    public static void main(String[] args) {
      try {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int H = Integer.parseInt(br.readLine());
        String[] contourMap = new String[H];
        for (int i=0; i<H; i++)
            contourMap[i] = br.readLine();
        ReliefMap rm = new ReliefMap();
        double[] ret = rm.getMap(contourMap);
        System.out.println("!");
        for (int i=0; i<ret.length; i++)
            System.out.println(ret[i]);
        System.out.flush();
      }
      catch (Exception e) {}
    }
}
```

C#

```
// ----- imitation of provided library class -----
using System;
public class Relief {
    public static double measure(int x, int y)
    {   Console.WriteLine("?{2}{0}{2}{1}", x, y, Environment.NewLine);
```

```
        Console.Out.Flush();
        return double.Parse(Console.ReadLine());
    }
}
// ----- the solution itself -----
public class ReliefMap {
    public double[] getMap(string[] contourMap) {
        // your code here
    }
// ----- main interaction code - can be added to the solution class -----
    private static void Main()
    {   int lineCount = int.Parse(Console.ReadLine());
        string[] contourMap = new string[lineCount];
        for(int i = 0; i < lineCount; i++)
            contourMap[i] = Console.ReadLine();
        double[] result = new ReliefMap().getMap(contourMap);
        Console.WriteLine("!");
        foreach(double d in result)
            Console.WriteLine(d);
        Console.Out.Flush();
    }
}
```

Python

```
# ----- imitation of provided library class -----

import sys

class Relief:

   @staticmethod

   def measure(x, y):

       print '?'

       print x

       print y

       sys.stdout.flush()

       return float(sys.stdin.readline())

# ----- the solution itself -----

class ReliefMap:

   def getMap(self, contourMap):

       # your code here

       pass

# ----- main interaction code -----

H = int(sys.stdin.readline())

contourMap = []

for i in range(0, H):

   contourMap.append(sys.stdin.readline()[:-1])

rm = ReliefMap()

ret = rm.getMap(contourMap)

print '!'
```

```
for line in ret:
   print line
   sys.stdout.flush()
```

VB.NET

```
Module Module1
   ' ----- imitation of provided library class -----
   Public Class Relief
       Public Shared Function measure(ByVal x As Integer, ByVal y As Integer) As Double
           Try
               Console.WriteLine("?")
               Console.WriteLine(x.ToString())
               Console.WriteLine(y.ToString())
               Console.Out.Flush()
               Dim ret As Double
               ret = Val(Console.ReadLine())
               Return ret
           Catch
               Return 0
           End Try
       End Function
   End Class
   ' ----- the solution itself -----
   Public Class ReliefMap
```

```
    Public Function getMap(ByVal contourMap As String()) As Double()
        ' your code here
    End Function
End Class
' ----- main interaction code -----
Sub Main()
    Dim H As Integer
    Try
        H = Val(Console.ReadLine())
    Catch
    End Try
    Dim contourMap(H) As String
    Try
        For i As Integer = 0 To H - 1
            contourMap(i) = Console.ReadLine()
        Next i
    Catch
    End Try
    Dim rm As ReliefMap
    rm = New ReliefMap
    Dim ret As Double()
    ret = rm.getMap(contourMap)
    For i As Integer = 0 To ret.Length() - 1
```

```
            Console.WriteLine(ret(i))

        Next i

        Console.Out.Flush()

    End Sub

End Module
```

Finally, note that even if your solution correctly interacts with the visualizer, the results of testing your solution locally and on the server might still differ:

- check for visualizer updates which can happen when the match has already started. Sometimes the first version of the visualizer might contain some bugs, which will be found and fixed in later versions.
- the visualizer doesn't implement the check for time limit violation. So a solution which finishes successfully locally can get "Time Limit Exceeded" when tested on the server.
- on the other hand, if your solution uses certain iterative technique and checks its running time itself, the results can differ because on different systems different number of iterations is done.
- floating point calculations can cause minor differences in your solution's behaviour on different systems, and in problems like BounceOff this might cause major differences in results.

** Rewriting the Visualizer

Problem

Nowadays most Marathon problems come with visualizers, which simplify local testing. But sometimes the official visualizer lacks some functionality you need or is simply uncomfortable to use. In these cases you can modify the visualizer or even write your own. This recipe will address several issues you have to keep in mind when doing this.

Solution

If you're using Java, modifying the visualizer is simple: you just take the official one and fix whatever you need, or move pieces of code into new one. Java is extremely convenient because you can keep the scheme of parameters generation provided by the visualizer, which is important for being able to reproduce the exact test case that will be generated for a specific test case.

For other languages, you'll have to take the official one as the base and translate it. Depending on the problem, this can be quite simple (for example, if only

math without any language-specific classes is involved). However, you won't be able to generate exactly the same test case (at least not in a simple way), since random numbers generators differ for different languages. The official visualizer typically uses the same SecureRandom class with SHA1PRNG algorithm, the same as server tester does. To have exactly the same test cases generated, you have two basic options:

- write your own port of this algorithm. It is possible, but quite time-consuming, so it's probably a thing you want to do between matches and not during one.
- use the official visualizer once to generate input data for all test cases you will be using, save it to a file (or a set of files) and use your visualizer to load data from them, test your solution on this data and process the results. This doesn't include much extra work - after all, you've probably written the code which reads the parameters directly from the visualizer, uses them to drive your solution and passes the results back (as described in "Running Visualizer with Your Solution" recipe). You'll just have to modify this code to read parameters from file instead of standard input stream, or simply redirect input stream to be taken from file. To capture input data generated by the visualizer, you can modify it to write it to a file, or write a dummy solution which will read it, dump it to a file and return something that visualizer will accept as a valid return.

Note that you can use the second approach directly only for problems which generate all parameters immediately, before first call of the solution, for example, Permute, Planarity, CellularAutomaton etc. If the problem is a multi-step one, with some parameters being generated on the go, like in StreetSales or TilesMatching, you might not be able to pre-generate and capture them easily. However, most problems which look like that, in reality pre-generate some state randomly and then obtain parameters to be passed to the solution from this state in a deterministic way; see, for example, ChessPuzzle, BlackBox or ReliefMap. For them, you can modify the visualizer to output this hidden state, and build your visualizer over it.

The most important part of developing an alternate visualizer is ensuring that the results it produces are the same as the ones produced by official visualizer - otherwise you might spend time on improving completely wrong thing. If your visualizer generates parameters from the scratch, remember to compare them to the ones generated by official visualizer; and in any case spend some time testing the scores given by your visualizer with ones given by official one - they have to be identical when obtained for the same solution.

Discussion

Why could someone want to spend time on modifying the visualizer or inventing it from scratch? There can be quite a lot of reasons:

- visualizer efficiency. The official visualizer usually copies the logic of the server code which tests your solutions. It is not optimized in any way, so in some cases it might be less efficient than you want it to be. For example, in BounceOff the simulation of ball movement could have been optimized, and a lot of people did it to win some execution time.
- interaction efficiency. The official visualizer interacts with the solution via standard input/output, because it needs to be compatible with solutions in all languages allowed in the competition. In some matches the data passed in this way is quite large, so passing parameters for a single test case can take several minutes. For example, in SpaceMedkit 2 of 5 input parameters were huge string arrays (300 KB and 43 MB respectively), and they didn't vary from run to run. Rewriting the visualizer so that it passed only varying parameters, and constant parameters were taken by the solution directly from file, and doing a batch of test cases in row (with one file read across all test cases) decreased time of local testing run a lot.
- need for more information. The things you might want to know about processing of your solution can be anything - from hidden parameters of the test case to visualization of some extra elements of the problem. This is the reason why learning enough Java to be able to understand visualizer source code and slip a few minor modifications into it is a must.
- tracing and debugging possibilities. In some cases it is useful to be able to do step-by-step tracing of either your code or visualizer's code on a particular test case. You can't do this using standard visualizer, but once you have rewritten it so that the visualizer calls the solution directly, you're free to use any debugging tools your IDE provides.

Even if your visualizer is correct, you may not get the exact same results when comparing the official visualizer to your new one. Floating point rounding issues could creep in, and in some problems this can cause very different results. In these cases, creative intervention is required to verify the operation of your runner. If you suspect this and are using C++, try compiling both your visualizer and solution without optimizations. This avoids register optimization passes from the compiler which can cause some rounding errors.

Implementing the Limitations Locally

Comparing Your Solutions**

Problem

You have two or more different solutions, and neither of them is the best on all test cases. You want to choose the best of them - either for your final submission or for further improvement.

Solution

The most evident solution is just to make a full submission for each of your solutions on TopCoder server. This will give you an idea of how your solutions perform against other competitors' solutions, and certainly you can simply choose the highest-scoring of your submissions. However, this way you'll get no information on performance on individual tests, and nothing about how your solutions score compared to each other. Making example submissions gives detailed information on each test case, but there are only a few tests available - definitely not enough to make a decision about one solution being really better than the other. To get decent information for comparison, you'll have to test your solutions locally.

Use the visualizer or the tester you've written yourself to run each solution on a massive batch of tests - at least a hundred, the more the better. Keep track of the individual scores of each solution, and calculate their scores against each other using the same method of calculating the overall score as the one used by TopCoder server (the one described in problem statement).

Discussion

When you estimate the results of local testing, individual scores can be transformed into overall solution's score in multiple ways. It's important to use the real scoring formula for this, since other ones might distort the results significantly. For example, if the problem statement defines the overall score as "the sum of individual scores, divided by 100", you can use the sum of raw scores without loss of score quality. But if the problem uses any form of relative scoring, the things are different, and you would be misled by checking just the sum of raw scores. For example, if each test case's contribution to the overall score is its individual score, divided by the maximal score anybody achieved on this test case, then it's not the absolute value of the individual score that matters, but the relative value within a test case.

Note that the suggested method of comparing solutions is not a panacea. Often you'll find out that all your solutions do a bad job on a particular class of tests, which have been solved much better by someone else. If the problem uses relative scoring, these cases' contributions to overall scores will differ from your local estimates a lot. A good practice is to compare locally the scores of as many your solutions as possible - including the most trivial ones, like returning empty array or a constant.

It is possible to gather information from doing full submission for several different solutions and compare their scores from TopCoder tester with locally predicted ones. Note, though, that the interval between full submissions might distort this information, if the holder of top scores on some test cases resubmits between two submissions of yours.

Your scoring program should track as many information about your solutions

as possible: the quantity and seeds of failed test cases, the best and the worst seeds etc. This is easy to do and will save you a lot of time when tracking down why the solution fails, why it scored lower than another one against your expectations, and what you should do to improve it.

Deciding How to Improve Your Solution

Problem

There comes a time in each match where you need to decide which aspects of your solution need improving. This may be a particular set of test cases that you need to improve on, or a certain component within your approach that needs refinement, and you may or may not have a good idea exactly how much improvement is needed.

Solution

While there are no hard and fast rules for how to improve your solution, there are a few tricks that are often applicable:

- Get an approximation of the best possible score for each case. Sometimes it's possible to get an upper (or lower if the score must be minimized) bound with a few heuristics and to use this as a gold standard against which to compare your solutions. This can help you to determine which test cases you have the most to gain on, and exactly how much potential there is for gain on each of them.
- Write a "cheating" solution and run it locally. In many problems there are hidden parameters that the tester is aware of, but your solution is not (typically some parameters of test case generation process). If your solution guesses these parameters and relies on them in further calculations, try to modify your tester to pass these parameters directly to your solution and modify your solution to use them instead of your guess. This can help you decide whether or not it's worth the extra effort to improve your parameter estimation process.
- Extend the time limit. This is particularly helpful in optimization problems (for example, based on hill climbing or simulated annealing techniques), where you are testing a large number of potential returns against a scoring function. If you're trying to decide how much a 10% speed improvement can help, first allow your submission to use 10% more time and see whether things improve enough to make it worthwhile. Letting a solution run long is also another great way to get estimates of the best scores possible for a test case.
- Separate analysis of each component of your solution. There are a lot of Marathon problems where your solution naturally breaks into a few

different components, perhaps stages in a multi-stage approach. Putting some effort into determining which pieces of your solution are the weakest can have huge benefits. The analysis can range from simply thinking about the different pieces and how important each seems (or where you've been neglecting potential improvements) all the way to devising specific quality tests for each piece in isolation.

- Watch the visualizer. You have to be careful with this one, because the visualizer can be as much of a distraction as it can be a useful tool, but very often the best way to gain insight into your solution's weaknesses is watch it in action. Pay careful attention to the decisions it makes, and how you would correct those decisions if you could intervene, and then see if you can modify the code to intervene for you.

Discussion

It's important to remember that Marathon Matches are as much about optimizing the time you spend on the problem as they are about optimizing your solution. A little bit of extra time spent carefully considering where your effort is best applied can make the time spent applying that effort incredibly more useful. With time and practice you'll also find that you're able to skip some of the more analytical steps and rely a bit more on your intuition to guide you, but whenever you find that improvements are hard to come by, take a step back and make sure that you're working on the right things.

Now let's have a look at the problems in which the listed tricks can be helpful.

- The best possible score often is evident from the problem statement, especially for problems which require achieving some result as closely as possible. For example, in DigitsPattern and CellularAutomaton you are required to get a pattern as close to the given one as possible; in ChessPuzzle you have to remove as many tiles as possible, ideally all of them, etc.

However, even in problems of this kind the evident best score sometimes can't be reached. In Planarity, you need to get as few pairs of intersecting edges as possible - but for non-planar graphs you won't be able to get rid of all intersections, so the optimal score will be strictly greater than 0. To make the estimate more realistic depending on the graph given, one could use crossing number inequality, which states that for any graph with V vertices and E edges the number of edge intersections of this graph is >= E-3*V. Given that in this problem E was chosen between 2*V and 5*V-1, this estimate was useful for approximately two thirds of test cases.

In problems in which the best possible score is not evident right away, it still can be roughly estimated sometimes. In PolymerPacking the score is the number of segments in the polymer, divided by the area occupied by it. The maximal score is not given explicitly, but can be estimated (very roughly) as 2 in a following

way: given an unlimited number of mirror operations, a chain without 0 elements can be turned into (1,1, -1,-1, 1,1, -1,-1, ...), in which each new pair of segments adds one unit of area, making area roughly equal to L/2.

- Cases of problems in which hidden parameters need to be estimated as part of the solution and not as the final answer are not so frequent, but still exist. Thus, in DensityImaging test cases were generated using so-called "blur factor" - the number of times the blurring process was applied during test case generation. A lot of good solutions estimated it and used for final adjustment of the return. A "cheating" tester could pass it directly to the solution, and you could either check that the estimate was correct or watch the effect of getting it wrong.
- Making the solution run faster is almost always a good idea - simply because it can't hurt in any way except for taking extra implementation time. 10% more time won't make a critical improvement if your solution consists of one round of hill climbing, since most likely the result will have established by then. But if you restart your hill climbing several times and take the best result, or use any other approach which needs several batches of action, a 10% improvement might give you time to run another batch and thus affect the result significantly.
- Almost all good solutions consist of several stages, and more than once after the match people realize that they've spent days on pruning one stage, when they should have spent an hour on another one. The only thing which can be recommended here is to do a pause now and then and to have a fresh look at the solution.
- Note that watching the visualizer can be misleading. For example, in ChessPuzzle the intuitive approach is to remove the central pieces first and the border and corner ones later, since there are several types of tiles which lead only to the border of the board. However, the top solutions used calculations instead of human intuition, and their moves seem completely counter-intuitive to the watcher until later in the game.

Improving Your Solution in the Home Stretch**

Problem

You are in the "home stretch" (usually starting Sunday evening for those who work typical day jobs), so you cannot invest time in large refactorings, different algorithms, or things which might not pay off, but still want to make some improvements.

Solution

Use one or several of the tried and true methods for squeezing a few extra points out of a solution:

1. Loop until you run out of time repeating your algorithm if it uses a pseudo-random generator, and halt when nearing the time limit, returning the best solution found so far.
2. Attempt solutions which use multiple algorithms (already coded ones) and return the best result.
3. Splice multiple solutions together based on local testing data and input parameters.
4. Write a script to search for optimal values of any "tweakable" parameters.

Discussion

It's late Sunday evening and, after peering at your screen through bleary eyes, taking the trip to the bathroom muttering strange things to yourself and provoking stares from your children or cats, you've just discovered The Idea. It requires a scrap and rewrite, but it will completely dominate the scoreboard. You'll be published after this! "This will rule them all," you cackle rubbing your hands together in that sleep-deprived, maniacal way.

Well, probably not. It's important to make a reasonable estimate of how much time you have left and how you can use it. At this stage in the game you have one or two evenings left, and one social event, desperate grocery stop, or call from your child's principal could easily cut that in half or eliminate it. Often those last-minute ideas just can't pay off (of course, there's always that one that might...)

So what can you do?

Loop until you run out of time

If you aren't using all available time and your algorithm uses a pseudo-random generator or can be easily adapted to use one in such a way that it creates solutions approximately as good, on average, as it used to, but with some deviation, then this is an easy tactic. Simply repeat your algorithm with different seeds and record the best score (as well as the return value which produced it).

This is applicable for problems with full information, in which you can predict the score of a return value before actually returning it. More specific examples include problems solvable with hill climbing, in which you pick the initial state and/or the change of the current state randomly. In general case running the hill climbing for 20 seconds gives not much improvement over running it for 5 seconds, and restarting it four times with different initial states increases the chances of finding another local maximum, with better value.

Run multiple algorithms in series

If your primary algorithm doesn't dominate the other algorithms you've scored locally and you have time to run more than one, this might be an alternative to looping until you run out of time. Run each algorithm and return the result which produces the best score. This also applies to problems with full information, since you need to calculate the score of the return as well.

Splice together multiple solutions

If you can't run multiple solutions in series, but you have a second algorithm which scores better on some subset of cases that you can detect with some accuracy from the input, include both algorithms and decide which one to use based on the input. To do this, you will need to analyze your local testing data.

Usually it is possible to write a quick brute force algorithm to solve small test cases, so you can use different algorithms depending on the size of the input. Alternatively, the problem might contain some non-size parameter which affects the nature of the test case itself. For example, in Epidemic there were three possible values of K - the variable which defined the lag between people getting infected and you getting information about this. It was very natural to process values of K = 0 (immediate gain of knowledge) and K = 2 (two days' delay) using different algorithms - in the first case you'd inoculate friends of the infected people, while in the second one you'll have to figure out how far had the infection spread already.

Find optimal values for "tweakable" parameters

A lot of problems require you to simplify them by plucking somewhat arbitrary numbers out of the air - such as the grid size for a packing problem, or how many elements to process at once, or a probabilistic fudge factor to account for anticipated improvements by some sort of later optimization pass.

If you've set up your local tester, you can write a script to inject these values into your solution and run your test set. In C++, you can do this with the preprocessor, passing a "-D" option to the compiler to set the value for each run. In Java, you can import a constant from another class locally, and generate and compile that class with a little bit of shell. You'll also need a script for automated estimate of results - you don't want to choose your values by looking through thousands of individual scores.

Since we are talking about a last minute hack that you are probably going to kick off before you hit the sack on Sunday or Monday evening (way later than you should have, of course), heavy strategies like coding a genetic algorithm for all the different parameters in shell is probably not as good an idea as it sounds (and it does sound good, doesn't it?). You'll probably only have time to write either a binary search for one parameter with no exit condition, or some kind of naive hill climbing. Well, just trying a lot of values would work as well, but that's too boring to be put here.

After you kick it off, make sure that your SSH server is up and running and you can get to your tester box - you know, running on that odd port so that you can

get through the firewall hole that MIS probably knows about but doesn't have time to fix. You're going to want to check on it once in the morning to make sure it didn't go completely haywire, and again at lunch just in case you can start the search for the next parameter.

Methods of Solving the Problem

Identifying Principal Types of Problems

Problem

Marathon problems are more versatile than Algorithm ones, so they are harder to classify with respect to solving methods, and such classification will be rather generic. However, all Marathon problems can be classified with respect to completeness of provided information.

Solution

The principal problem types are the following:

- full-information problems provide all information necessary to evaluate each possible return at once.
- multi-step problems provide the information in portions, and each portion might depend on the outcome of previous steps.
- hidden-information problems provide part of the information, and keep hidden another part of it which is important for scoring.

Discussion

Both Algorithm and Marathon tracks have a problem archive - a list of all problems which ever appeared in the matches, plus an option to solve them outside of the competition. Practicing (solving past problems to prepare to solving future problems) is less useful in Marathons than in Algorithm competitions. A lot of Algorithm problems use a relatively small set of algorithms, so once you know them all and can recognize and implement them fast, you're ready for future competitions, and most likely will do well.

Marathon problems are not like this. Each of them is unique, and requires a unique solution which is seldom similar to any solution to previous problems. Besides, solving an Algorithm problem takes at most a few hours, while a Marathon problem can take weeks. That's why people might spend some more time with the problem they started after the match is over, to finish some idea or to try a new trick, but seldom solve a new problem from archive from the scratch.

However, even a rough classification of problems can give a hint about possible techniques of solving it. Let's have a closer look at the suggested classification.

Full-information problems are the common and probably the most popular type among participants. They provide all information about the subject of the problem, with nothing kept secret. The competitor just has to construct the solution based on the known information and return it, and he can predict just how well will this solution score on each test case. This allows to try constructing several solutions and pick the best of them, or to start with one solution and improve it step by step, etc.

For this kind of problem the space of solutions is usually too large for straightforward search, but there are plenty of standard techniques like hill climbing and simulated annealing (discussed in next recipes) which can be applied in more or less evident way. Examples of full-information problems include Planarity, PolymerPacking, BookSelection, EnclosingCircles, QualityPolygons and lots of others.

Multi-step problems are also quite common and include game-based and measurement-based problems.

In **game-based problems** you play a game against AI opponent or environment, and you have to make a small decision on each step. After your decision is implemented, you receive a portion of new information about the state of the game. This way the information you've got changes with each move, but you never know everything, and have to take decisions with only partial information. Examples of game-based problems include ChessPuzzle, TilesMatching, Klondike etc.

Measurement-based problems usually involve a certain object with unknown parameters. You are allowed a limited number of information requests ("measurements") to uncover these parameters, and have to reconstruct the object the best you can from the pieces of information you learn. The number of measurements is usually limited to a number which is not enough to get full information, so you have to choose them wisely and approximate the unknown information from known fragments. Scoring is done by comparison of the actual object to your reconstruction, much like in hidden-information problems, but here you can control which information above initial one you get. Examples of measurement-based problems include BlackBox, ReliefMap, GraphicalAuthentication, ImageScanner etc.

Hidden-information problems are rather rare nowadays, but it's still an important class of problems. They have some information that is never given away to the competitor and is used only for scoring purposes. These are usually reconstruction problems and deciphering: you know something about an object, and have to reconstruct it using only this knowledge. Hidden information describes the actual object, and scoring is done based on comparison of the actual object and your reconstruction of it. Unlike measurement problems, hidden-information provide you no way to extend the initial information - you have to analyze it at best you can. Recent examples of hidden-information problems include BrokenClayTile, Enigma, GrilleReconstruction and earlier

crypto-problems.

Subtype of hidden-information problems is **randomized problems**. In them you have all information about some process, but it is simulated using certain random parameters which you can't predict. The task is to provide the parameters of simulation which yield a desired result or at least a close one. The scoring is done based on the results of actual simulation. Examples include WatermarkSequence, FlockingBehaviour etc.

Of course, from time to time there appear problems which are hard to classify, but realizing this basic classification can help a lot by limiting the space of problem solutions to be searched to techniques and tricks best applicable to problems of each particular class. For example, seeing a full-information problem, one can immediately start thinking about applying some form of hill-climbing to it (though it might not be the optimal solution).

Generating Ideas for Your Solution

Problem

You've read and analyzed the problem statement and maybe even launched the visualizer. You understand the problem, but don't know how to start solving it yet.

Solution

The first basic way of generating ideas is to rely on existing human knowledge about this class of problems (or this exact problem, if you're lucky). Almost all Marathon problems rely on something from the writer's experience - a game they played, an article they read, a task they had to manage in their daily life. Thus, it's quite probable to search and find something related to the problem or at least similar to it.

The second way is to play with the problem yourself to study it in detail. Some problems contain non-evident patterns which can give a starting point for the solution.

Discussion

The good thing about Marathons is the amount of time you have: the typical two weeks gives you enough time to think about the problem, dig through Internet in search of similar problems and applicable approaches, try various ideas, reject inefficient ones and finally pick and polish the best one. Sometimes it's totally necessary to try several ideas to get a decent feel of the problem.

Let's have a closer look at the strategies described in Solution and how they can be applied to existing Marathon problems.

1. Rely on existing knowledge of the problem

Most Marathon problems have some real-life prototype, which may have drawn enough attention from other programmers or researchers to have documents that describe heuristics for solving this type of problem. The prototype can be either a known serious problem or a game which was formalized and studied.

The strategy for solving this kind of problems is: pick out the keywords of the problem and try to search for them in various wordings on the Web.

Examples of problems which allow using this approach are:

- Planarity is based on a game of same name, which is simpler (it handles only planar graphs) but is in turn based on rigorous research. The task is to find the layout of the graph which produces its rectilinear crossing number (or gets as close to is as possible), which in turn has been subjected to a lot of investigation.
- BookSelection is similar to lots of serious cutting and packing problems - strip or bin packing, container loading, knapsack etc. All of them are easy to find, so one just has to pick the most similar one (they differ in subtle shades of problem statement) and study its solutions (which are non-trivial but still give a starting point of thought).
- TilesPuzzle is based on Eternity II puzzle, and at least one solution was based on explanation of one of Eternity II solvers.
- Klondike is based on a card game of same name, which is surprisingly well-studied. I got an idea of my solution from an article which describes ways of solving this game with some minor changes in the input data.
- Permute is actually linear ordering problem, subject to plenty of studies as well.

Another version of this approach is reading "Post your approach" for similar old problems or even studying the solutions themselves, though one has to be really desperate for this :-) A perfect example of this approach are the crypto-marathons which ran in 2008: in OneTimePad Psyho used a very clever method of data compression to fit the necessary data into the size limit of submission, and in following matches it was successfully adopted by many competitors.

2. Play with the problem

Recent visualizers for game-based problems tend to provide "manual" mode - a mode in which the user interacts with the visualizer directly, without writing a solution. Older visualizers didn't have this feature, so one had to look for a non-TopCoder implementation of this game.

Regardless of the tool you use, the idea is to try solving the problem by hand and to observe the patterns which you follow when doing it. Automating these patterns can give you at least a draft of a solution. Examples of problems which allow using this approach are:

- ChessPuzzle. After a few games it becomes quite evident that it's better to clear one layer of tiles completely before starting next layer. Another observation (though a one not necessary for a good strategy) is that there are more ways to move to the border and corner cells of the board, so they should be left uncleared for as long as possible when clearing the layer.
- TilesMatching. Playing it can hint that it's important to leave as many places to fit the next tile as possible, regardless of what type it comes in.
- ContinousSameGame. A basic strategy derived from playing the game by hand can be "remove the blocks of all colors except for one, and hope that the blocks of this color will form a large group in the end".

Approaching Single-Player Game-Based Problems

Problem

Quite a lot of Marathon problems are game-based and simulate more-or-less known board or card games, usually single-player, but sometimes multi-player, with server playing for the opponents. They require developing a strategy to win the game, or to do good in it.

Solution

The game-based problems follow the common pattern: on each move you are given the current state of the game, and you have to return your move. After the move you usually get some information about new state of the game which was not available before. In each state there exists a limited number of valid moves, and the general task can be divided in two sub-tasks: identifying valid moves and choosing the best one of them.

Valid moves are identified using the game rules, exactly as described in problem statement. Usually the number of technically possible moves is limited, and it is possible to iterate over all of them and check each one for validity. Alternatively, you can generate all valid moves based on the rules.

Thus, in TilesMatching you can iterate over all cells of the board and check whether placing the current tile there is possible accordingly to the rules of the game. In ChessPuzzle, on the contrary, it might be easier to generate all cells which are allowed to click in the current state of game, since they are defined by the type and position of cell that was clicked last. Checking the validity of moves in Klondike is a bit trickier because each step offers several types of moves: advancing the position in the deck and moving cards from the deck to the stack/pile, from the stack to the stack/pile, or from the pile to the stack. Each possibility must be checked separately.

Identifying valid moves is a necessary step to implementing your solution, a fairly mechanical process. The real problem-solving begins at the stage of choosing

the best move on each step in a way which will lead to the best game outcome possible.

One of the possible techniques is: assign each game state a score which describes how "good" the state is. The score function is usually constructed heuristically, by playing the game, noting features of the game state which seem important for human decision and trying to formalize their contribution to score (as a bonus for good features and a penalty for bad ones).

For example, in Klondike the score of the state can be based on:

- number of cards in completed piles (the most natural part of the score, which shows how close to the end the game is);
- number of valid moves from this state (more mobile states are scored higher, since we don't want any dead ends);
- number of cards in the deck;
- number of face-down cards in the stacks (larger numbers are scored lower, since they mean that we have a lot of unknown and unmanageable cards);
- difference of sizes of greatest and smallest completed piles (the smaller, the better) etc.

In TilesMatching the score of the state can be based on:

- number of tiles on the board;
- number of tiles in the row/column in which the last tile was placed;
- for each possible tile - the number of positions where it can be placed (with a penalty if a tile can be only discarded);
- for each empty cell - the number of tiles which can be placed on it (with a penalty if only wild-card can be placed) etc.

In ChessPuzzle the score of next click can be based on:

- its position on the board (non-border cells are generally trickier to get to, so they are scored higher),
- number of valid moves after this click (once again, more mobile states are scored higher),
- number of layers left in this cell (the more layers are left, the higher is the score) etc.

Once the score function is constructed, some form of greedy search with limited depth is possible. A sample pseudo-code follows:

```
function estimate_state(state, depth)
{   if (depth < search_depth)
   {   for each valid move from state
           estimate_state(state after this move, depth+1)
```

```
      return the score of the best of the searched states
   and remember the move which leads to it
   }
   return the score of state
}

function main(state)
{   for each valid move from state
      estimate_state(state after this move, 0)
   return the move which leads to the best of the searched states
}
```

The search of depth 0 is just greedily picking the move which leads to the best score on the current move. Increasing the depth of the search usually improves the quality of the result, but requires more implementation effort (to organize the search itself and storage of optimal moves) and runs longer, so it has to be done carefully. Some popular optimizations include:

- checking for total time spent (depth of search varies depending on time left),
- discarding some states at deep levels of search,
- storing the sequence of best moves found during the search. Thus, a search of depth 4 will find the best sequence of 5 moves which can be played next. The first move will be returned immediately, and the next 4 can be stored and returned at next 4 queries without repeating the search.

Discussion

Most problems allow to borrow the validity check from visualizer source code (reusing visualizer code is allowed ever since it became available without decompiling the visualizer). This gives a minor advantage to Java/C++/C# users (this part of visualizer is written in a way which doesn't use any Java specifics, and basic C++/C# syntax is quite similar to Java).

Note that this recipe describes only one of the possible approaches to this kind of problems, and not necessarily the best or universal; in some problems it might turn out to be quite inefficient. Thus, in TilesMatching the winning approach was fundamentally different.

Another method is to score the move itself instead of the game state it leads to. Thus, in TilesMatching it is evident that the moves which clear row AND column simultaneously are more valuable than the moves which clear row OR column, which in turn are more valuable than the moves which don't clear anything. This fact can't be described in terms of state scoring, since the state of the board after such move has no evidence of cleared rows and columns, and thus of the move which was just done.

A version of this approach is to create a priority of move types (instead of priority of individual moves), try out the technically possible moves of each priority in turn and use the first valid move encountered (or one of the valid moves of first valid type encountered). This approach works nicely in Klondike, where moves are divided in types naturally, and prioritizing the types is easy.

Note that Marathon game-based problems are different from Algorithm game-based problems: in Algorithm, you have full information about the game, while in Marathons some information stays hidden. Besides, Marathon games are usually large compared to Algorithm games, thus the whole space of states of the game can't be searched, and you need to invent some heuristics.

In the problems which give you new information about state of the game after some moves you can't run a real search with full information. However, this can be fixed by making some problem-specific assumptions about what can be the score of the state after the move. Thus, in Klondike you get the value of face down card only when it is open, but this can be accounted in the score by assuming that opening a face down card is always valuable to us. In ChessPuzzle it is profitable to try to remove all cells of topmost layer before starting next layer, so you can solve the problem as if there was only one layer; this way unknown information matters only between layers, and the search within the layer can be deep enough. In TilesMatching the state can be scored by using probabilities of getting each type of tile next.

Finally, almost every version of described approaches requires not only in-depth analysis of the game, but also quite a lot of tuning. The set of state features, exact values of bonuses and penalties, depth and other parameters of search on each move - all these things should be fine-tuned only when you're sure that you're not able to invent a fundamentally different algorithmically and significantly better approach. These are the things you leave till the last days of match and tune as last attempt on improving your score.