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++

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

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

Java

```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.