Statistics

Match Editorial

Archive
Printable view
Discuss this match

SRM 169
Tuesday, October 28, 2003

Match summary

The fact that 507 coders (or should I say top coders) have registered for this match is quite impressive. DIV I coders lived up to their high standard and the submission and success rates where pretty solid. But I think there was one other quite impressive thing if not a bit suprising and frustrating....

Lets look at the following scenario:
It is probably (if memory serves me correctly) the only time that I have seen complete silence in my room (Div II) and basically no submissions for the longest period of time.

Here is how it went:
We all started with the 250 as usual and we all submitted the easy problem within the normal time frame of about 5 to 10 mins. So far, so good. I check the submission times and I see that I am ok. I was about 4th in my room at the time when I submitted. So what do we all do next? But of course, we open and tackle the 500 problem. So we all opened the 500 and...

And I think that any spectator watching my room would have been quite taken aback and would have shaken his applet a bit in order to see why nothing was happening. And nothing seemed to be happening for the longest period of time. But in actuality we were all hard at work trying to figure out a strategy for getting this problem into submission. Time passed, and an occasional compilation or testing message would appear on the room history list. Finally after what must have been like 40 minutes submissions started to trickle in.

But alas! No time was left to even attempt the 1000 and in my room there were no submissions for the hard problem. What is even more interesting is the fact that most of the submissions for the DIV II 500 would fail either by a direct coder challenge or through a system test case. There was about a 20% success rate for this problem. Any questions? Yes, this write-up is dedicated to the coders of alternative submission lifestyle. But do read on, as we will have a look at why there was such a poor success rate for this problem and we will look at a strategy to maximize chances at scoring high points especially for those who would like to trade the green of their earthly pastures for the blue of the heavens.

The Problems

Swimmers

Rate It Discuss it

Used as: Division Two - Level One:

Value	250
Submission Rate	260 / 299 (86.96%)
Success Rate	196 / 260 (75.38%)
High Score	**swanksax2** for 238.48 points (6 mins 18 secs)
Average Score	191.46 (for 196 correct submissions)

This is a relatively simple problem but one has to be always on the lookout for special cases.
In a nutshell: A swimmer will plunge into a river at some point A and swim downstream a certain given distance to point B (i.e. with the river current) and then reverse and swim upstream back to point A (i.e. against the current) We are given the distance that swimmer will swim before they will reverse. We are also given the speed of the river current and the speed of the swimmer. We are to calculate the actual time that it took the swimmer to swim from point A to point B and back to point A. It is in fact a simple vector addition on the x-axis. One important thing to watch out for is the fact that the swimmer might not have enough speed to return. If the speed of the river current is greater than or equal to the swimmer's speed then the swimmer will never make it back. In such a case we return -1.

Simple formula:

The time to swim from point A to point B is given by
TimeFromAtoB = ABDistance/(speedOfSwimmer + speedOfCurrent)
Here the swimmer is taking advantage of the fact that his/her speed gets added
to the speed of the current.

The time to swim back from point B to point A (against the current) is given by
TimeFromBtoA = ABDistance/(speedOfSwimmer - speedOfCurrent)
Here the swimmer has to fight the current and in fact looses speed which
basically means that the current speed gets subtracted from the swimmer's speed.
Thus the total time that the swimmer takes to complete their round is going to be the addition of these two.

Contraints:
There were a number of things to watch out for though:

(1) The return value was to be rounded down (take the floor of the result) but one has to be careful not to do the following:

floor(ABDistance/(speedOfSwimmer + speedOfCurrent)) +
floor(ABDistance/(speedOfSwimmer - speedOfCurrent))

as this would result occasionally in too small a value.
We need to do the following:

floor((ABDistance/(speedOfSwimmer + speedOfCurrent)+
(ABDistance/(speedOfSwimmer - speedOfCurrent)
)

(2) The actual distance from A to B could be 0. This is a special case and we would return for such a swimmer the time of 0 regardless of what the swimmer's speed is or the river current speed. This takes precedence over case (3) below.
(3) If the actual speed of the swimmer is less than (or equal-to) the speed of the river current then the swimmer would never make it back. In such a case we return -1;
(4) Note that we can have a case where we would have division by 0 in our formula if the speed of the swimmer is the same as the speed of the river current. We need to ensure that we trap this case before it gets to our formula. Note that test of (equal-to) in case (3) takes care of that. The code would fail if we only checked for the less-than condition.

Sample java code:

```
int[] result = new int[speeds.length];

for(int i=0; i < speeds.length; i++)
{
   // case (2)
   if(distances[i] <= 0)
   {
      result[i] = 0;
      continue;
   }
   // case (3) note the (equal-to).
   if(speeds[i] <= current)
   {
      result[i] = -1;
   }
   // case (1)
   else
   {
      double temp = ((double)distances[i]/(speeds[i]+current));
      temp += ((double)distances[i]/(speeds[i]-current));
      result[i] = (int)(Math.floor(temp));
   }
}
return result;
```

Where did code fail:
Most code that failed the system tests or was successfully challenged did not ensure that case (2) takes precedence over case (3) as well as missing the (equal-to) condition in case (3)

Twain

Rate It Discuss it

Used as: Division Two - Level Two:

Value	500
Submission Rate	110 / 299 (36.79%)
Success Rate	23 / 110 (20.91%)
High Score	**SleepyOverlord** for 273.99 points (32 mins 1 secs)
Average Score	216.66 (for 23 correct submissions)

Oh boy, oh boy! This was the killer question for div II. On the surface this is a relatively simple problem where we are given a number of rules to follow that transform a given input string through a succession of well defined steps. There is a maximum of 7 transformations (and a minimum of 0 where the input string in unchanged) that can be applied to the input string. The key to solving this problem successfully is to build the solution in increments and test each transformation independently.Basically if you followed the rules you would have no problem solving this question. But then why did so many coders fail this question? And secondly why was the fastest submission time clocked at about 30 mins?

With this particular question you have to ensure that you read each rule carefully as mistakes in such questions are difficult to pinpoint and thus very time consuming to debug.
Lets look at two sample transformations necessary for this question:

Transformation (1) - year 1

- If a word starts with "x", replace the "x" with a "z".
- Change all remaining "x"s to "ks"s.

Solution pseudo-code:

find all the "x"'s in the input string and for each "x" do the following test:
(beginning of a word test)
if the x is the very first character in the string (beginning of a word special case) then
  replace this 'x' with 'z'
else if the character just before the current 'x' is a space character ' ' then
  replace this 'x' with 'z'
(inside the word)
    else
      replace the 'x' with "ks"

Faster Trick Solution:

Add to the beginning of the input String a space: input = " "+input. This will collapse the word test code above from two cases to one generic case.
Replace each occurrence of " x" with " z" (note the added space to test a beginning of a word)
Replace remaining occurrences of "x" with "ks"
Remove the extra at the beginning " ": input = input.substring(1, input.length());

Here is a java code snippet:

```
input = " " + input;
input = input.replaceAll(" x", " z");
input = input.replaceAll("x","ks" );
input = input.substring(1);
```

Transformation (5) year 5

- If a word starts with "sch", change the "sch" to a "sk".
- If a "ch" is directly followed by an "r", change the "ch" to a "k".
- After applying the above rules, change all "c"s that are not directly followed by an "h", to a "k".
(This includes all "c"s that are the last letter of a word.)

Solution pseudo-code:

Add to the beginning of the input String a space: input = " "+input;
Replace each occurrence of " sch" with " sk" (note the added space to test a beginning of a word)
Replace each occurrence of "chr" with " kr"
Append a space to the input string: input = input + " " (This is a nice trick to ensure that we can test for the last letter without falling off the edge)
Replace all occurrences of "c" not followed by an "h" to a "k":
(Find all 'c' and then check if the next character to the right is an 'h'
  if(not) then
    replace the 'c' with "k"
)
Remove the helper spaces from beginning and end.

Here is a java code snippet:

```
input = " " + input + " ";
input = input.replaceAll(" sch", " sk");
input = input.replaceAll("chr","kr" );
for(int i=0; i < input.length(); i++)
{
   if(input.charAt(i) == 'c' && input.charAt(i+1) != 'h')
   input = input.substring(0, i) + "k" +
          input.substring(i+1);
}
input = input.substring(1, input.length()-1);
```

As can be seen just going over two transformations this can be quite time consuming especially if you do not use any tricks and shortcuts. It is also very error prone.

Where did code fail:
Basically all over the place. It was easy to make mistakes here. In some cases the code had some problems with correctly figuring out

where the beginning of a word was.

Here is a suggested strategy that might be useful when you hit a rather difficult to write/understand problem:
(1) Read it carefully a couple of times (Always a good thing to do)
(2) Formulate a strategy for the solution before you write code
(3) Test your strategy against a couple sample test cases (no code yet)
(4) if you find that the question is very time consuming (i.e. for the 500 I would say if within 25 minutes you are not testing the code) you might consider skipping this question for now and peek if the next question is perhaps more to your liking. You can always come back (with a loss of time and thus points of course) if you find the other question even harder.

In SRM 169 it turns out that the 1000 point question for Div II was actually pretty accessible and doable. Skipping the 500 question would have been preferable (I think) to a lot of people.

FairWorkload Rate It Discuss it

Used as: Division Two - Level Three:

Value	1000
Submission Rate	19 / 299 (6.35%)
Success Rate	11 / 19 (57.89%)
High Score	**.** for 914.74 points (8 mins 50 secs)
Average Score	721.75 (for 11 correct submissions)

Used as: Division One - Level Two:

Value	500
Submission Rate	147 / 184 (79.89%)
Success Rate	132 / 147 (89.80%)
High Score	**antimatter** for 487.35 points (4 mins 36 secs)
Average Score	348.60 (for 132 correct submissions)

This is a very nice question, which can be solved in a couple ways. But first and foremost lets try to understand what the problem is asking us to do.
We are given a number of filing cabinets where each cabinet contains some number of folders. We are also given the number of available workers. What we would like to do is to partition the cabinets amongst the N workers in a such a way that
(1) each worker works with adjacent cabinets
(2) number of folders that each worker has to work on (cabinets can not be shared) is spread as evenly as possible.
We are to return the largest partition in our most evenly spread arrangement.

As an example: if we are given 9 cabinets:

10 20 30 40 50 60 70 80 90

and we have 3 workers, the best solution would be:

10 20 30 40 50 | 60 70 | 80 90

And the largest partition in this case is 170 (80+90)

Constraints:
The number of cabinets we will work with is [2..15]
The max number of workers we will have is [2..15]

This is a nutshell is a problem of generating all N (number of workers) subsets of the cabinets such that the following is always true:

(1) A subset must contain only adjacent cabinets
(2) Cabinets can not repeat
(3) Cabinets can not be omitted: The total number of cabinets in all subsets gives us the total set back
(4) Subsets cannot be empty

Here are a couple approaches to this problem:

(1) First we can generate all the possible valid subsets as follows:
We set a globalMax to be the total number of folders.
We start with the full set of cabinets and we divide this set into two subsets LeftSet and RightSet.
We then take the RightSet and again divide it by into two sets.
We repeat this process until we have generated the same number of sets as we have workers.
For each subset that we generate in this manner we calcuate the LocalMax.
We then compare this LocalMax to the GlobalMax and if GlobalMax is greater than the LocalMax then we set the new GlobalMax to be the value of the LocalMax.
Basically we have found at this point a solution that distributes the folders better.
Once we go through all these sets we return the current GlobalMax.

This can be done in a recursive manner as follows (after lanenal's solution from room 2)

```
int globalMax;
public int getMostWork(int[] f, int w)
{
   for(int i=0; i<f.length; i++) globalMax += f[i];
   genSubsets(f, -1, w, 0);
   return globalMax;
}
void genSubsets(int[] f, int from, int w, int localMax)
{
   int sum = 0;
   // Final subset
   if(w==1)
   {
     // get the folders for this subset
   for( int i=from+1; i<=f.length-w; i++) sum += f[i];
      // is this the most folders for the current subsets?
      // If yes then update the localMax
      if(localMax<sum) localMax = sum;
      // Is our localMax less than (better) the globalMax
      if(localMax < globalMax) globalMax = localMax;
      return;
   }

   for( int i=from+1; i<=f.length-w; i++)
   {
      // the LeftSubset folders sum
   sum += f[i];
   // Generate the RightSubset by dividing the current
   // subset in two with the pivot at i.
   // Propagate localMax to the RightSubset
   genSubsets(f, i, w-1, sum>localMax?sum:localMax);
   }
}
```

(2) Application of Binary Search. For a non-recursive example of this approach please look at Ukraine's code in room 1.

(3) We can also use Dynamic Programming. I will leave this as an exercise to the reader.

MineField

Rate It | Discuss it

Used as: Division One - Level One:

Value	250
Submission Rate	181 / 184 (98.37%)
Success Rate	177 / 181 (97.79%)
High Score	**ZorbaTHut** for 245.83 points (3 mins 42 secs)
Average Score	206.02 (for 177 correct submissions)

This is actually a very easy problem in my opinion.Basically given locations of randomly generated mines on a playing grid we are to return an array which for each location on the grid that is not occupied by a mine will give the count of all the mines that are neighboring this location.

Constraints:

9x9 grid
Mines input will be [0, 10]
Each grid location will neighbor at most 8 mines.

I will present here three steps to the solution:

(1) Create a grid char Array [10][10] initialized to 0
(The grid is extended by a 1 char buffer in every direction)
Parse the input string and place the mines in grid. (denoted with 'M')
(2) For each grid location in the range of [1..9][1..9]
  int count=0;
  Iterate through each neighbor of the current grid location
  if(the neighbor location contains 'M')
   count++;
  Set the current grid location to count;
(3) Convert the grid representation to the output string.

The reason why I have created the grid to be 10x10 is so that I have a buffer (which is empty and thus will not contribute to the count) which allows me not to have a boundary testing code when I am looking at the neighbors:

Here is a java snippet for (2) assuming the grid is made to be [10][10]

```
char[][] grid = new char[10][10];
```

```
int count;
for(int row=1; row <= 9; row++)
{
   for(int col=1; col <= 9; col++)
   {
      count=0;
      if(grid[row][col] = 'M') continue;
      // Test all neighbours
      // (Note no need to test if outside of grid)
      for(int r = row-1; r < = row+1; r++)
      {
         for(int c = col-1; c < = col+1; c++)
         {
            if(row==r && col==c) continue;
            if(grid[r][c] == 'M')
               count++;
         }
      }
      grid[row][col] = count;
   }
}
```

Where did code fail:
The success rate for this problem was almost 100% but a few solutions failed mostly on incorrectly going through the neighbors. Especially one has to be careful with the

```
if(row==r && col==c) continue
```

condition to ensure that one doesn't do this instead: (for example)

```
if(row==0 && col==0) continue;
```

GoldMine Rate It Discuss it

Used as: Division One - Level Three:

Value	900
Submission Rate	56 / 184 (30.43%)
Success Rate	39 / 56 (69.64%)
High Score	**WishingBone** for 757.59 points (12 mins 49 secs)
Average Score	472.93 (for 39 correct submissions)

This is a rather straight forward problem once you realize that is can be solved in a greedy manner.
At any given point a miner has a profit value associated with him depending on the state of each mine (i.e is the mine full? How much will the mine yield if we add this miner to it) We need to find the mine that will yield best profit if we add this miner to it.The basic premise has to do with maximizing resources. Given a number of gold mines and a formula for the amount of money that each mine would yield given a number of miners working in the mine we can use the following observation to solve this problem:

Take the next available miner and apply him into each of the mines.
Calculate the profit made by applying this miner to each mine and choose the best
profit for this miner. Apply this miner to the gold-mine that makes the most money from
having him.
We continue in this manner until all miners have been distributed.

How do we know that this will work? Think of it this way: any given gold mine will have profit based on the following function:

Profit(gold-mine , num-miners)
We could run now this function for every gold mine for all the possible num_miners [0..6]
that could fit into that mine.
This would build for us a profit table for each mine.
We could then sort this table based on profit and number of miners.

Then, once we are given the number of available miners we would simply look up the
most profitable entry in our table for the most miners we could fit in it. Then we repeat this
process for the remaining miners (after having removed the best entry from the profit table)
This is in essence a greedy algorithm, which is equivalent to the algorithm given before.

Constraints:
We can have at most six miners per gold mine.
All miners must be placed.

That's all, folks. Until the next SRM. keep on coding!

By **AleaActaEst**
TopCoder Member

Twitter

Follow

Recent Blog Posts Updated

Apr 23 @timmhicks – Tim Hicks Happy Hump Day topcoders! We are excited to announce that we will be releasing a new look for the very popular /tc by...Read More

Apr 23 Do you ever find yourself hitting “send” on an email and wondering if it’ll arrive in the recipient’s inbox? Sending email has become so ubiquitous, simple and...Read More

Apr 22 @ClintonBon – Clinton Bonner We know what you’re thinking. Great, another ‘puff piece’ on the ‘wisdom of crowds’ and how all we need to do is post...Read More

View More

About topcoder

The topcoder community gathers the world's experts in design, development and data science to work on interesting and challenging problems for fun and reward. We want to help topcoder members improve their skills, demonstrate and gain reward for their expertise, and provide the industry with objective insight on new and emerging technologies.

About Us

Get Connected

Your email address

Submit