properties in the problem, attacking the problem from a non obvious angle, etc) so that you (or your team) will be able to derive the required solution to a hard/original type C problem in IOI or ICPC Regionals/World Finals and do so *within* the duration of the contest.

UVa	Title	Problem Type	Hint
10360	Rat Attack	Complete Search or DP	Section 3.2
10341	Solve It		Section 3.3
11292	Dragon of Loowater		Section 3.4
11450	Wedding Shopping		Section 3.5
10911	Forming Quiz Teams	DP with bitmask	Section 8.3.1
11635	Hotel Booking		Section 8.4
11506	Angry Programmer		Section 4.6
10243	Fire! Fire!! Fire!!!		Section 4.7.1
10717	Mint		Section 8.4
11512	GATTACA		Section 6.6
10065	Useless Tile Packers		Section 7.3.7

Table 1.3: Exercise: Classify These UVa Problems

Exercise 1.2.1: Read the UVa [47] problems shown in Table 1.3 and determine their problem types. Two of them have been identified for you. Filling this table is easy after mastering this book—all the techniques required to solve these problems are discussed in this book.

1.2.3 Tip 3: Do Algorithm Analysis

Once you have designed an algorithm to solve a particular problem in a programming contest, you must then ask this question: Given the maximum input bound (usually given in a good problem description), can the currently developed algorithm, with its time/space complexity, pass the time/memory limit given for that particular problem?

Sometimes, there are more than one way to attack a problem. Some approaches may be incorrect, others not fast enough, and yet others 'overkill'. A good strategy is to brainstorm for many possible algorithms and then pick the **simplest solution that works** (i.e. is fast enough to pass the time and memory limit and yet still produce the correct answer)[4]!

Modern computers are quite fast and can process[5] up to $\approx 100M$ (or 10^8; $1M = 1,000,000$) operations in a few seconds. You can use this information to determine if your algorithm will run in time. For example, if the maximum input size n is $100K$ (or 10^5; $1K = 1,000$), and your current algorithm has a time complexity of $O(n^2)$, common sense (or your calculator) will inform you that $(100K)^2$ or 10^{10} is a very large number that indicates that your algorithm will require (on the order of) hundreds of seconds to run. You will thus need to devise a faster (and also correct) algorithm to solve the problem. Suppose you find one that runs with a time complexity of $O(n \log_2 n)$. Now, your calculator will inform you that $10^5 \log_2 10^5$ is just 1.7×10^6 and common sense dictates that the algorithm (which should now run in under a second) will likely be able to pass the time limit.

[4]Discussion: It is true that in programming contests, picking the simplest algorithm that works is crucial for doing well in that programming contest. However, during *training sessions*, where time constraints are not an issue, it can be beneficial to spend more time trying to solve a certain problem using the *best possible algorithm*. We are better prepared this way. If we encounter a harder version of the problem in the future, we will have a greater chance of obtaining and implementing the correct solution!

[5]Treat this as a rule of thumb. This numbers may vary from machine to machine.

The problem bounds are as important as your algorithm's time complexity in determining if your solution is appropriate. Suppose that you can only devise a relatively-simple-to-code algorithm that runs with a horrendous time complexity of $O(n^4)$. This may appear to be an infeasible solution, but if $n \leq 50$, then you have actually solved the problem. You can implement your $O(n^4)$ algorithm with impunity since 50^4 is just $6.25M$ and your algorithm should still run in around a second.

Note, however, that the order of complexity does not necessarily indicate the actual number of operations that your algorithm will require. If each iteration involves a large number of operations (many floating point calculations, or a significant number of constant sub-loops), or if your implementation has a high 'constant' in its execution (unnecessarily repeated loops or multiple passes, or even I/O or execution overhead), your code may take longer to execute than expected. However, this will usually not be the case as the problem authors should have designed the time limits so that a well-coded algorithm with a suitable time complexity will achieve an AC verdict.

By analyzing the complexity of your algorithm with the given input bound and the stated time/memory limit, you can better decide whether you should attempt to implement your algorithm (which will take up precious time in the ICPCs and IOIs), attempt to improve your algorithm first, or switch to other problems in the problem set.

As mentioned in the preface of this book, we will *not* discuss the concept of algorithmic analysis in details. We *assume* that you already have this basic skill. There are a multitude of other reference books (for example, the "Introduction to Algorithms" [7], "Algorithm Design" [38], "Algorithms" [8], etc) that will help you to understand the following prerequisite concepts/techniques in algorithmic analysis:

- Basic time and space complexity analysis for iterative and recursive algorithms:
  - An algorithm with k-nested loops of about n iterations each has $O(n^k)$ complexity.
  - If your algorithm is recursive with b recursive calls per level and has L levels, the algorithm has roughly $O(b^L)$ complexity, but this is a only a rough upper bound. The actual complexity depends on what actions are done per level and whether pruning is possible.
  - A Dynamic Programming algorithm or other iterative routine which processes a 2D $n \times n$ matrix in $O(k)$ per cell runs in $O(k \times n^2)$ time. This is explained in further detail in Section 3.5.
- More advanced analysis techniques:
  - Prove the correctness of an algorithm (especially for Greedy algorithms in Section 3.4), to minimize your chance of getting the 'Wrong Answer' verdict.
  - Perform the amortized analysis (e.g. see Chapter 17 of [7])—although rarely used in contests—to minimize your chance of getting the 'Time Limit Exceeded' verdict, or worse, considering your algorithm to be too slow and skips the problem when it is in fact fast enough in amortized sense.
  - Do output-sensitive analysis to analyze algorithm which (also) depends on output size and minimize your chance of getting the 'Time Limit Exceeded' verdict. For example, an algorithm to search for a string with length m in a long string with the help of a Suffix Tree (that is already built) runs in $O(m+occ)$ time. The time taken for this algorithm to run depends not only on the input size m but also the output size—the number of occurrences occ (see more details in Section 6.6).

- Familiarity with these bounds:
  - $2^{10} = 1,024 \approx 10^3$, $2^{20} = 1,048,576 \approx 10^6$.
  - 32-bit signed integers (`int`) and 64-bit signed integers (`long long`) have upper limits of $2^{31}-1 \approx 2\times10^9$ (safe for up to $\approx$ 9 decimal digits) and $2^{63}-1 \approx 9\times10^{18}$ (safe for up to $\approx$ 18 decimal digits) respectively.
  - Unsigned integers can be used if only non-negative numbers are required. 32-bit unsigned integers (`unsigned int`) and 64-bit unsigned integers (`unsigned long long`) have upper limits of $2^{32}-1 \approx 4\times10^9$ and $2^{64}-1 \approx 1.8\times10^{19}$ respectively.
  - If you need to store integers $\geq 2^{64}$, use the Big Integer technique (Section 5.3).
  - There are $n!$ permutations and 2^n subsets (or combinations) of n elements.
  - The best time complexity of a comparison-based sorting algorithm is $\Omega(n \log_2 n)$.
  - Usually, $O(n \log_2 n)$ algorithms are sufficient to solve most contest problems.
  - The largest input size for typical programming contest problems must be $< 1M$. Beyond that, the time needed to read the input (the Input/Output routine) will be the bottleneck.
  - A typical year 2013 CPU can process $100M = 10^8$ operations in a few seconds.

Many novice programmers would skip this phase and immediately begin implementing the first (naïve) algorithm that they can think of only to realize that the chosen data structure or algorithm is not efficient enough (or wrong). Our advice for ICPC contestants[6]: Refrain from coding until you are sure that your algorithm is both correct and fast enough.

n	Worst AC Algorithm	Comment
$\leq$ [10..11]	$O(n!), O(n^6)$	e.g. Enumerating permutations (Section 3.2)
$\leq$ [15..18]	$O(2^n \times n^2)$	e.g. DP TSP (Section 3.5.2)
$\leq$ [18..22]	$O(2^n \times n)$	e.g. DP with bitmask technique (Section 8.3.1)
≤ 100	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, ${}_nC_{k=4}$
≤ 400	$O(n^3)$	e.g. Floyd Warshall's (Section 4.5)
$\leq 2K$	$O(n^2 \log_2 n)$	e.g. 2-nested loops + a tree-related DS (Section 2.3)
$\leq 10K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort (Section 2.2)
$\leq 1M$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree (Section 2.3)
$\leq 100M$	$O(n), O(\log_2 n), O(1)$	Most contest problem has $n \leq 1M$ (I/O bottleneck)

Table 1.4: Rule of thumb time complexities for the 'Worst AC Algorithm' for various single-test-case input sizes n, assuming that your CPU can compute $100M$ items in 3s.

To help you understand the growth of several common time complexities, and thus help you judge how fast is 'enough', refer to Table 1.4. Variants of such tables are also found in many other books on data structures and algorithms. This table is written from a *programming contestant's perspective*. Usually, the input size constraints are given in a (good) problem description. With the assumption that a typical CPU can execute a hundred million operations in around 3 seconds (the typical time limit in most UVa [47] problems), we can predict the 'worst' algorithm that can still pass the time limit. Usually, the simplest algorithm has the poorest time complexity, but if it can pass the time limit, just use it!

[6]Unlike ICPC, the IOI tasks can usually be solved (partially or fully) using several possible solutions, each with different time complexities and subtask scores. To gain valuable points, it may be good to use a brute force solution to score a few points and to understand the problem better. There will be no significant time penalty as IOI is not a speed contest. Then, iteratively improve the solution to gain more points.

From Table 1.4, we see the importance of using good algorithms with small orders of growth as they allow us to solve problems with larger input sizes. But a faster algorithm is usually non-trivial and sometimes substantially harder to implement. In Section 3.2.3, we discuss a few tips that may allow the same class of algorithms to be used with larger input sizes. In subsequent chapters, we also explain efficient algorithms for various computing problems.

Exercise 1.2.2: Please answer the following questions below using your current knowledge about classic algorithms and their time complexities. After you have finished reading this book once, it may be beneficial to attempt this exercise again.

1. There are n webpages ($1 \le n \le 10M$). Each webpage i has a page rank r_i. You want to pick the top 10 pages with the highest page ranks. Which method is better?
   (a) Load all n webpages' page rank to memory, sort (Section 2.2) them in descending page rank order, obtaining the top 10.
   (b) Use a priority queue data structure (a heap) (Section 2.3).
2. Given an $M \times N$ integer matrix Q ($1 \le M, N \le 30$), determine if there exists a sub-matrix of Q of size $A \times B$ ($1 \le A \le M, 1 \le B \le N$) where `mean`$(Q) = 7$.
   (a) Try all possible sub-matrices and check if the mean of each sub-matrix is 7. This algorithm runs in $O(M^3 \times N^3)$.
   (b) Try all possible sub-matrices, but in $O(M^2 \times N^2)$ with this technique: ________ .
3. Given a list L with $10K$ integers, you need to *frequently* obtain `sum(i, j)`, i.e. the sum of `L[i] + L[i+1] + ...+ L[j]`. Which data structure should you use?
   (a) Simple Array (Section 2.2).
   (b) Simple Array pre-processed with Dynamic Programming (Section 2.2 & 3.5).
   (c) Balanced Binary Search Tree (Section 2.3).
   (d) Binary Heap (Section 2.3).
   (e) Segment Tree (Section 2.4.3).
   (f) Binary Indexed (Fenwick) Tree (Section 2.4.4).
   (g) Suffix Tree (Section 6.6.2) or its alternative, Suffix Array (Section 6.6.4).
4. Given a set S of N points *randomly* scattered on a 2D plane ($2 \le N \le 1000$), find two points $\in S$ that have the greatest separating Euclidean distance. Is an $O(N^2)$ complete search algorithm that tries all possible pairs feasible?
   (a) Yes, such complete search is possible.
   (b) No, we must find another way. We must use: ________ .
5. You have to compute the shortest path between two vertices on a weighted Directed Acyclic Graph (DAG) with $|V|, |E| \le 100K$. Which algorithm(s) can be used in a typical programming contest (that is, with a time limit of approximately 3 seconds)?
   (a) Dynamic Programming (Section 3.5, 4.2.5, & 4.7.1).
   (b) Breadth First Search (Section 4.2.2 & 4.4.2).
   (c) Dijkstra's (Section 4.4.3).

   (d) Bellman Ford's (Section 4.4.4).
   (e) Floyd Warshall's (Section 4.5).

6. Which algorithm produces a list of the first $10K$ prime numbers with the best time complexity? (Section 5.5.1)

   (a) Sieve of Eratosthenes (Section 5.5.1).
   (b) For each number $i \in$ `[1..10K]`, test if `isPrime(i)` is true (Section 5.5.1).

7. You want to test if the factorial of n, i.e. $n!$ is divisible by an integer m. $1 \leq n \leq 10000$. What should you do?

   (a) Test if $n!$ % $m == 0$.
   (b) The naïve approach above will not work, use: _________ (Section 5.5.1).

8. Question 4, but with a larger set of points: $N \leq 1M$ and one additional constraint: The points are *randomly scattered* on a 2D plane.

   (a) The complete search mentioned in question 3 can still be used.
   (b) The naïve approach above will not work, use: _________ (Section 7.3.7).

9. You want to enumerate all occurrences of a substring P (of length m) in a (long) string T (of length n), if any. Both n and m have a maximum of 1M characters.

   (a) Use the following C++ code snippet:

```
for (int i = 0; i < n; i++) {
  bool found = true;
  for (int j = 0; j < m && found; j++)
    if (i + j >= n || P[j] != T[i + j]) found = false;
  if (found) printf("P is found at index %d in T\n", i);
}
```

   (b) The naïve approach above will not work, use: _________ (Section 6.4 or 6.6).

1.2.4 Tip 4: Master Programming Languages

There are several programming languages supported in ICPC[7], including C/C++ and Java. Which programming languages should one aim to master?

Our experience gives us this answer: We prefer C++ with its built-in Standard Template Library (STL) but we still need to master Java. Even though it is slower, Java has powerful built-in libraries and APIs such as BigInteger/BigDecimal, GregorianCalendar, Regex, etc. Java programs are easier to debug with the virtual machine's ability to provide a stack trace

[7]Personal opinion: According to the latest IOI 2012 competition rules, Java is currently still not supported in IOI. The programming languages allowed in IOI are C, C++, and Pascal. On the other hand, the ICPC World Finals (and thus most Regionals) allows C, C++ and Java to be used in the contest. Therefore, it is seems that the 'best' language to master is C++ as it is supported in both competitions and it has strong STL support. If IOI contestants choose to master C++, they will have the benefit of being able to use the same language (with an increased level of mastery) for ACM ICPC in their University level pursuits.

when it crashes (as opposed to core dumps or segmentation faults in C/C++). On the other hand, C/C++ has its own merits as well. Depending on the problem at hand, either language may be the better choice for implementing a solution in the shortest time.

Suppose that a problem requires you to compute 25! (the factorial of 25). The answer is very large: 15,511,210,043,330,985,984,000,000. This far exceeds the largest built-in primitive integer data type (`unsigned long long`: $2^{64}-1$). As there is no built-in arbitrary-precision arithmetic library in C/C++ as of yet, we would have needed to implement one from scratch. The Java code, however, is exceedingly simple (more details in Section 5.3). In this case, using Java definitely makes for shorter coding time.

```
import java.util.Scanner;
import java.math.BigInteger;

class Main {                          // standard Java class name in UVa OJ
  public static void main(String[] args) {
    BigInteger fac = BigInteger.ONE;
    for (int i = 2; i <= 25; i++)
      fac = fac.multiply(BigInteger.valueOf(i));   // it is in the library!
    System.out.println(fac);
} }
```

Mastering and understanding the full capability of your favourite programming language is also important. Take this problem with a non-standard input format: the first line of input is an integer N. This is followed by N lines, each starting with the character '0', followed by a dot '.', then followed by an unknown number of digits (up to 100 digits), and finally terminated with three dots '...'.

```
3
0.1227...
0.517611738...
0.7341231223444344389923899277...
```

One possible solution is as follows:

```
#include <cstdio>
using namespace std;

int N;         // using global variables in contests can be a good strategy
char x[110];  // make it a habit to set array size a bit larger than needed

int main() {
  scanf("%d\n", &N);
  while (N--) {                  // we simply loop from N, N-1, N-2, ..., 0
    scanf("0.%[0-9]...\n", &x);   // '&' is optional when x is a char array
                         // note: if you are surprised with the trick above,
                      // please check scanf details in www.cppreference.com
    printf("the digits are 0.%s\n", x);
} } // return 0;
```

Source code: `ch1_01_factorial.java`; `ch1_02_scanf.cpp`

Not many C/C++ programmers are aware of partial regex capabilities built into the C standard I/O library. Although `scanf/printf` are C-style I/O routines, they can still be used in C++ code. Many C++ programmers 'force' themselves to use `cin/cout` all the time even though it is sometimes not as flexible as `scanf/printf` and is also far slower.

In programming contests, especially ICPCs, coding time should *not* be the primary bottleneck. Once you figure out the 'worst AC algorithm' that will pass the given time limit, you are expected to be able to translate it into a bug-free code and fast!

Now, try some of the exercises below! If you need more than 10 lines of code to solve any of them, you should revisit and update your knowledge of your programming language(s)! A mastery of the programming languages you use and their built-in routines is extremely important and will help you a lot in programming contests.

Exercise 1.2.3: Produce working code that is *as concise as possible* for the following tasks:

1. Using **Java**, read in a double
   (e.g. `1.4732`, `15.324547327`, etc.)
   echo it, but with a minimum field width of 7 and 3 digits after the decimal point
   (e.g. `ss1.473` (where '`s`' denotes a space), `s15.325`, etc.)
2. Given an integer n ($n \leq 15$), print π to n digits after the decimal point (rounded).
   (e.g. for $n = 2$, print `3.14`; for $n = 4$, print `3.1416`; for $n = 5$, prints `3.14159`.)
3. Given a date, determine the day of the week (Monday, . . . , Sunday) on that day.
   (e.g. 9 August 2010—the launch date of the first edition of this book—is a Monday.)
4. Given n random integers, print the distinct (unique) integers in sorted order.
5. Given the distinct and valid birthdates of n people as triples (DD, MM, YYYY), order them first by ascending birth months (MM), then by ascending birth dates (DD), and finally by ascending age.
6. Given a list of *sorted* integers L of size up to $1M$ items, determine whether a value v exists in L with no more than 20 comparisons (more details in Section 2.2).
7. Generate all possible permutations of {'A', 'B', 'C', . . . , 'J'}, the first $N = 10$ letters in the alphabet (see Section 3.2.1).
8. Generate all possible subsets of $\{0, 1, 2, \ldots, N\text{-}1\}$, for $N = 20$ (see Section 3.2.1).
9. Given a string that represents a base X number, convert it to an equivalent string in base Y, $2 \leq X, Y \leq 36$. For example: "FF" in base X = 16 (hexadecimal) is "255" in base $Y_1 = 10$ (decimal) and "11111111" in base $Y_2 = 2$ (binary). See Section 5.3.2.
10. Let's define a 'special word' as a lowercase alphabet followed by two consecutive digits.
    Given a string, replace all 'special words' of length 3 with 3 stars "***", e.g.
    S = "line: a70 and z72 will be replaced, aa24 and a872 will not"
    should be transformed into:
    S = "line: *** and *** will be replaced, aa24 and a872 will not".
11. Given a *valid* mathematical expression involving '+', '-', '*', '/', '(', and ')' in a single line, evaluate that expression. (e.g. a rather complicated but valid expression `3 + (8 - 7.5) * 10 / 5 - (2 + 5 * 7)` should produce `-33.0` when evaluated with standard operator precedence.)

1.2.5 Tip 5: Master the Art of Testing Code

You thought you nailed a particular problem. You identified its problem type, designed the algorithm for it, verified that the algorithm (with the data structures it uses) will run in time (and within memory limits) by considering the time (and space) complexity, and implemented the algorithm, but your solution is still not Accepted (AC).

Depending on the programming contest, you may or may not get credit for solving the problem partially. In ICPC, you will only get points for a particular problem if your team's code solves **all** the secret test cases for that problem. Other verdicts such as Presentation Error (PE), Wrong Answer (WA), Time Limit Exceeded (TLE), Memory Limit Exceeded (MLE), Run Time Error (RTE), etc. do not increase your team's points. In current IOI (2010-2012), the subtask scoring system is used. Test cases are grouped into subtasks, usually simpler variants of the original task with smaller input bounds. You will only be credited for solving a subtask if your code solves all test cases in it. You are given *tokens* that you can use (sparingly) throughout the contest to view the judge's evaluation of your code.

In either case, you will need to be able to design good, comprehensive and tricky test cases. The sample input-output given in the problem description is by nature trivial and therefore usually not a good means for determining the correctness of your code.

Rather than wasting submissions (and thus accumulating time or score penalties) in ICPC or tokens in IOI, you may want to design tricky test cases for testing your code on your own machine[8]. Ensure that your code is able to solve them correctly (otherwise, there is no point submitting your solution since it is likely to be incorrect—unless you want to test the test data bounds).

Some coaches encourage their students to compete with each other by designing test cases. If student A's test cases can break student B's code, then A will get bonus points. You may want to try this in your team training :).

Here are some guidelines for designing good test cases from our experience.
These are typically the steps that have been taken by problem authors.

1. Your test cases should include the sample test cases since the sample output is guaranteed to be correct. Use '`fc`' in Windows or '`diff`' in UNIX to check your code's output (when given the sample input) against the sample output. Avoid manual comparison as humans are prone to error and are not good at performing such tasks, especially for problems with strict output formats (e.g. blank line *between* test cases versus *after every* test cases). To do this, *copy and paste* the sample input and sample output from the problem description, then save them to files (named as '`input`' and '`output`' or anything else that is sensible). Then, after compiling your program (let's assume the executable's name is the '`g++`' default '`a.out`'), execute it with an I/O redirection: '`./a.out < input > myoutput`'. Finally, execute '`diff myoutput output`' to highlight any (potentially subtle) differences, if any exist.

2. For problems with multiple test cases in a single run (see Section 1.3.2), you should include two identical sample test cases consecutively in the same run. Both must output the same known correct answers. This helps to determine if you have forgotten to initialize any variables—if the first instance produces the correct answer but the second one does not, it is likely that you have not reset your variables.

3. Your test cases should include tricky corner cases. Think like the problem author and try to come up with the worst possible input for your algorithm by identifying cases

[8]Programming contest environments differ from one contest to another. This can disadvantage contestants who rely too much on fancy Integrated Development Environment (IDE) (e.g. Visual Studio, Eclipse, etc) for debugging. It may be a good idea to practice coding with just a **text editor** and a **compiler**!

that are 'hidden' or implied within the problem description. These cases are usually included in the judge's secret test cases but *not* in the sample input and output. Corner cases typically occur at extreme values such as $N = 0$, $N = 1$, negative values, large final (and/or intermediate) values that does not fit 32-bit signed integer, etc.

4. Your test cases should include *large* cases. Increase the input size incrementally up to the maximum input bounds stated in the problem description. Use large test cases with trivial structures that are easy to verify with manual computation and large *random* test cases to test if your code terminates in time and still produces reasonable output (since the correctness would be difficult to verify here). Sometimes your program may work for small test cases, but produces wrong answer, crashes, or exceeds the time limit when the input size increases. If that happens, check for overflows, out of bound errors, or improve your algorithm.
5. Though this is rare in modern programming contests, do not assume that the input will always be nicely formatted if the problem description does not explicitly state it (especially for a badly written problem). Try inserting additional whitespace (spaces, tabs) in the input and test if your code is still able to obtain the values correctly without crashing.

However, after carefully following all these steps, you may still get non-AC verdicts. In ICPC, you (and your team) can actually consider the judge's verdict and the leader board (usually available for the first four hours of the contest) in determining your next course of action. In IOI 2010-2012, contestants have a limited number of tokens to use for checking the correctness of their submitted code against the secret test cases. With more experience in such contests, you will be able to make better judgments and choices.

Exercise 1.2.4: Situational awareness
(mostly applicable in the ICPC setting—this is not as relevant in IOI).

1. You receive a WA verdict for a very easy problem. What should you do?
   (a) Abandon this problem for another.
   (b) Improve the performance of your solution (code optimizations/better algorithm).
   (c) Create tricky test cases to find the bug.
   (d) (In team contests): Ask your team mate to re-do the problem.
2. You receive a TLE verdict for your $O(N^3)$ solution.
   However, the maximum N is just 100. What should you do?
   (a) Abandon this problem for another.
   (b) Improve the performance of your solution (code optimizations/better algorithm).
   (c) Create tricky test cases to find the bug.
3. Follow up to Question 2: What if the maximum N is 100.000?
4. Another follow up to Question 2: What if the maximum N is 1.000, the output only depends on the size of input N, and you still have *four hours* of competition time left?
5. You receive an RTE verdict. Your code (seems to) execute perfectly on your machine. What should you do?

6. Thirty minutes into the contest, you take a glance at the leader board. There are *many* other teams that have solved a problem X that your team has not attempted. What should you do?

7. Midway through the contest, you take a glance at the leader board. The leading team (assume that it is not your team) has just solved problem Y. What should you do?

8. Your team has spent two hours on a nasty problem. You have submitted several implementations by different team members. All submissions have been judged incorrect. You have no idea what's wrong. What should you do?

9. There is one hour to go before the end of the contest. You have 1 WA code and 1 fresh idea for *another* problem. What should you (or your team) do?

   (a) Abandon the problem with the WA code, switch to the other problem in an attempt to solve one more problem.

   (b) Insist that you have to debug the WA code. There is not enough time to start working on a new problem.

   (c) (In ICPC): Print the WA code. Ask two other team members to scrutinize it while you switch to that other problem in an attempt to solve *two* more problems.

1.2.6 Tip 6: Practice and More Practice

Competitive programmers, like real athletes, must train regularly and keep 'programming-fit'. Thus in our second last tip, we provide a list of several websites with resources that can help improve your problem solving skill. We believe that success comes as a result of a continuous effort to better yourself.

The University of Valladolid (UVa, from Spain) Online Judge [47] contains past ACM contest problems (Locals, Regionals, and up to World Finals) plus problems from other sources, including various problems from contests hosted by UVa. You can solve these problems and submit your solutions to the Online Judge. The correctness of your program will be reported as soon as possible. Try solving the problems mentioned in this book and you might see your name on the top-500 authors rank list someday :-).

As of 24 May 2013, one needs to solve $\geq$ 542 problems to be in the top-500. Steven is ranked 27 (for solving 1674 problems) while Felix is ranked 37 (for solving 1487 problems) out of $\approx$ 149008 UVa users (and a total of $\approx$ 4097 problems).

UVa's 'sister' online judge is the ACM ICPC Live Archive [33] that contains *almost all* recent ACM ICPC Regionals and World Final problem sets since year 2000. Train here if you want to do well in future ICPCs. Note that in October 2011, about hundreds of Live Archive problems (including the ones listed in the second edition of this book) are mirrored in the UVa Online Judge.

Figure 1.2: Left: University of Valladolid Online Judge; Right: ACM ICPC Live Archive.