PERGAMON

Computers & Education 36 (2001) 299–315

COMPUTERS & EDUCATION

www.elsevier.com/locate/compedu

# Online Judge

Andy Kurnia [a,*], Andrew Lim [a], Brenda Cheang [b]

[a] *Department of Computer Science, National University of Singapore, 3 Science Drive 2, 117543 Singapore*
[b] *I-OPT, 5 Jalan Besar, Ong Ban Hong Leong Bldg, 03-01A, 208785 Singapore*

Received 7 August 2000; accepted 13 January 2001

**Abstract**

This report describes and evaluates the implementation and applicability of an automatic programming assignment grading system we named the online judge. We compared this with the manual grading system that is currently being used and showed that the automatic grading system, when implemented carefully, is more convenient, fairer, and more secure than the former. We have successfully tested the system on two courses. However, further studies need to be conducted to improve the effectiveness of learning through this system. 

*Keywords:* Automatic grading system; Online judge; Programming assignments

## 1. Introduction

Programming is one of the major skills taught in the Department of Computer Science. Virtually every Computer Science module involves some form of programming. Needless to say, some Computer Science modules require more programming than others.

Programming is a skill acquired through practice. To help students in this respect, programming assignments are given to students. The number of programming assignments reveals how much emphasis on programming a module has. Most of the higher-level modules would assume that students are already comfortable with programming in virtually any programming language, and thus has only one major programming project that emphasizes on the topic being covered, such as client-server networking and state space searching. However, modules that emphasize programming, such as Programming Methodology, Data Structures and Algorithms, and Competitive Programming, aim to nurture students in programming, and have many programming assignments with questions of varying difficulties.

* Corresponding author.
*E-mail addresses:* andykurn@comp.nus.edu.sg (A. Kurnia), alim@comp.nus.edu.sg (A. Lim).

0360-1315/01/$ - see front matter 
PII: S0360-1315(01)00018-5

Considering how an undergraduate student would apply the economic principle of maximizing gain with minimum effort to be effective, programming assignments have to be graded as otherwise they would not even be attempted. Consequently, teaching staff need to grade the students' attempts on the programming assignments.

Normally, grading would involve several paid graders to manually go through all the source files that students submit, either in printed form or electronically. The going gets tougher as more students take the module and as more programming assignments are thrown into the module, because the graders get more work to do, possibly without an increase in their pay rate.

Manual grading, as illustrated in Section 2, has its problems. As such, it would be a worthy and enriching experience to try another approach. Naturally, the other approach to this problem would be to have grading done automatically. A program would take in another program and report its judgement, which would then be translated into numerical grades. This program, like a human grader, would have to know the programming questions.

Interestingly enough, programming assignments for these modules are algorithmic in nature, and therefore do not require special user interfaces. They would not, for example, need to delve into the intricacies of machine-specific implementations such as graphical user interface, mouse input, printer output, audio output, and so on. Most programs that are algorithmic in nature would only need to take some input from the standard input in some prescribed format, just as if this input were the output of another program. They would then process them in memory, and output the results of the computation in some prescribed format, ready for yet another program to read in the results and make use of the computation. In short, they would act like a filter. By having a program that would pass in the input, and another program that would take in the output and verify them for correctness, it becomes possible to have grading of programming assignments done automatically.

While manual grading can only be done offline, automatic grading enables grading to be done either offline or online. Before the deadline, an offline automatic grader would take in submissions, but would otherwise be idle. When the deadline is reached, it would start checking all submissions that have been received and produce a report on its judgements on these submissions. On the contrary, before the deadline, an online automatic grader would take in submissions and produce its judgement immediately, allowing the students to decide if they would like to have another attempt on the question to obtain a better mark. An online automatic grader would only be idle after the deadline.

In this project, we are investigating how having an online judge would help grading. We used this method of grading in July–November 1999 and 2000 for CS3233 Competitive Programming, in October 1999 for ACM Programming Contest selection, and in January-April 2000 for CS1102 Data Structures and Algorithms. C++ was the programming language used for the first two, and Java was used for the third.

The use of computer technologies to help in grading assignments is not new. Some such works have been reported in Arnow (1995), Arnow and Bradshaw (1999), Kay (1998), Kay, Scott, Isaacson, and Reek (1994) and Reek (1996).

## 2. Flaws in manual grading

Traditionally, grading has always been done manually. This required the students to submit their source files, be it electronically, printed, or handwritten. After the deadline, grading would

take place. The graders would go through the scripts and assign a grade for each of them. It would take a lot of effort to have each script graded once, and it is rarely justified to have a second grading. As such, grading is necessarily subjective. Graders would award grades based on their different preferred styles. Furthermore, sometimes they are affected by their moods, which may change drastically at any time. In addition, graders have their own objectives. For example, they would want to finish marking as soon as possible. Some of the flaws are as outlined below.

### 2.1. Emphasis on standard solution

For the correctness aspect of grading, students are expected to solve questions given based on a standard approach, because the grader is only expected to be familiar with the model answer; and if the grader is not familiar with the algorithm the program uses to produce a correct answer, it would not be given full mark.

However, students are innovative. They design their own ways to solve the questions. It should be noted that often, there are indeed many ways to solve a particular question. At times, given a source file, it may be difficult to follow and to understand what algorithm it uses, and even when the algorithm is understood, it may be difficult to see whether that certain algorithm works, let alone to understand why.

We illustrate this point by giving some examples.

1. Given a positive integer $x$, there are several ways to compute the result of $\lfloor\sqrt{x}\rfloor$, the greatest integer less than or equal to the square root of $x$. A straightforward but slow method would be to convert $x$ to real number, use the library function sqrt($x$) or pow($x, \frac{1}{2}$), and convert the result back into an integer. A smart method would be to observe that $y^2 = y \cdot y = y \cdot \frac{2y}{2} = y \cdot \frac{2y+1-1}{2} = y \cdot \frac{1+(2y-1)}{2} = 1 + 3 + \ldots + (2y-1)$, simplifying the equation to a summation loop. For $x = y^2 \leqslant 2^{32}$, $\lfloor\sqrt{x}\rfloor \leqslant y \leqslant 2^{16}$, and the loop executes fairly faster because no floating-point calculation is involved. Yet another method, if the range of input were known to be small enough, would be to precompute all the values of $\lfloor\sqrt{x}\rfloor$ given any $x$ in range, and store them in a constant array such that the running time of the algorithm becomes constant.
2. Consider the shortest path problem. To solve this problem, one might use a short and simple but slow depth-first search algorithm. Dijkstra would, however, solve it with a very efficient algorithm, but the program becomes more difficult to understand.
3. Given many lines of input, the program is to reproduce them in reverse order, that is, the last line of input would become the first line of output, and vice versa. A straightforward approach would be to observe that the stack data structure, with its last-in first-out property, would help perform the reversal. The program would only need to push each line onto a stack, and after the whole input has been read, to pop them out and display them. In turn, there are many ways one may implement this stack, for example, by using a linked list, a vector, or an array with enough number of elements. An equally straightforward approach would be to observe that after the first line has been read, the problem has been reduced to a smaller instance of the same problem. It would only need to solve the reversal of the rest of the input, and once that is done, to display the first line. When the base case of empty input is reached, the program does nothing. Thus, recursion would be another way to solve this problem.

An intriguing approach would be to have a string that is initially empty, and as each line is read, this line and its accompanying newline would be prepended to the ever-growing string. At the end of the program, this long string would contain the required output, and is simply printed without an additional newline.

Consequently, if the approach that a student takes is unfamiliar to the grader, the grader may not accept the answer even though it is right. A flip side of this would be that students who seem to solve the questions in a way that vaguely resembles the standard approach, although with a subtle bug buried deep down inside the source files, would escape and be given the benefit of doubt. The grader would glance through the program and in the interest of being able to mark more scripts within the same amount of time would most likely award the program full marks.

### 2.2. Emphasis on aesthetical aspects

Many a time, awarding of full marks depend highly on the layout and presentation of the source files itself. Some correct attempts may be graded as incorrect, simply because they were not presented in the way the grader expects them to have been. The source files are required only to be neat and understandable. The emphasis is often incorrectly placed upon the aesthetical aspects of the presentation of the source files, and not necessarily upon the correctness of the source files.

#### 2.2.1. Indentation

Programs have to be visually appealing. Unfortunately, one cannot enforce a particular indentation style on others. An analogy to that would be dictating how one should indent a handwritten essay by specifying exact measurements for the page margins, the paragraph margin, the quotation margin, the point sizes, how many lines there should be between the title and the first paragraph, etc. While to a certain extent it is possible to enforce virtually any habit on students, it is often not a good idea to do so.

The problem with grading the indentation is that different graders have different preferences. For example, a grader who prefers this:

```
if (expression)  {
   something();
}
```

may penalize students who do this:

```
if (expression)
{
   something();
}
```

while another grader may do the exact opposite.

In other words, a style that would earn full marks from one grader might be one that would earn no marks from another grader. Most students would not know who would be grading their scripts, and therefore cannot tailor their indentation style to suit that particular grader.

### 2.2.2. Identifier

The programs submitted must have identifiers in readable English to get full marks. For example, all of the following declarations may earn different marks:

```
int  i,  j, k;
int d, m, y;
int d, m, y; /* date, month, year */
int date, month, year;
struct tian { int nian, yue, ri; } jintian;
struct day { int date, month, year; } today;
```

To the compiler, of course, they are all equally acceptable. It would not reject foreigners who program with identifiers in a non-English language. Also, it would not reject someone who programs with identifiers in a language that has never existed either. As long as the syntax of the programming language used identifies the token as an identifier, it has no reasons to complain.

### 2.2.3. Modularity

The more modular a program gets, the more named building blocks there are in the program and the shorter each building block gets, the more marks the student earns. On the other hand, function calls do impose a certain time penalty when the program is executed. The problem is that there is no standard way to determine how modular is modular. What is modular enough to one grader might not be modular enough to another. A standard guideline is that a subprogram should not be any longer than a screen page. However, that guideline is not very useful, because a long subprogram can be broken into several illogical parts to claim the modularity marks. Modularity is subjective.

### 2.2.4. Style

Some things can be expressed in more than one way. For example, in English, most sentences can be phrased in active and passive forms. In Java, one who calls the .toString() method explicitly will suffer mark deduction, simply because leaving it implicitly called by Java will result in a less noisy source file. In C, one who uses for would get more marks than one who imitates it using while, who in turn still gets slightly more marks than one who uses a pair of if and goto. In Perl, there is more than one way to do almost everything. One who uses for to mean a foreach may suffer a mark deduction, although they are semantically the same.

### 2.2.5. Comments

Adding comments is deemed to be necessary. However, this is only true in large-scale projects. In small-scale projects such as programming assignments, the assignments are very simple and comments are hardly useful.

Since comments are graded, students tend to throw in useless comments. An example would be:

```
while (i > 0 && stillneedtodecrement()) {
    i--; // decrement i
} // end while
```

There is also no standard way to comment. The kind of comments a grader likes may not be the kind of comments another grader likes.

### 2.3. Problems measuring the running time

In general, it is impossible to find out whether a program that has executed for a long time will or will not terminate. To be consistent, one would need to set a time limit, and declare that all programs that have not terminated when the time is up would never terminate and is hence, wrong. For example, if the time limit is 30 s, all correct programs that take 29.9 s to execute must be graded as correct, and all otherwise correct programs that would take 30.1 s to execute must be graded as inefficient.

There is no easy way to ensure that all graders will be consistent in this. Even when they are manually executing the program, it would be very difficult to break the program exactly after 30 s, unless the program is terminated by another program.

### 2.4. No defense against malicious programs

The only sure way to tackle the problem with consistent running time limit, which would also eliminate the problem of understanding alternative algorithms to find out whether they work, is to have the programs executed. Executing a program is the only way one can demonstrate that a program works or not.

However, once students learn that their programs are going to be executed by the staff member who happens to be grading their programs, all sorts of malicious programs may be submitted. A perfect example would be one that simulates the recursive removal of all files and directories starting from the staff member's home directory.

It is clearly not feasible to have staff members prescrutinize the source files to determine what they are going to do, in order to completely prevent the execution of malicious programs. There has to be an automatic mechanism that would safeguard the system's integrity against malicious programs.

### 2.5. Inconsistency

The same program graded by the same grader may get different marks depending on when it is graded. The graders' moods change from time to time, and their moods indirectly affect the grades that they award.

Let us consider this scenario. At the beginning of a grading session, a grader gets to grade a program with long comments. This grader thinks: "The comments are long, but I will go through them anyway, since I am paid to do it and the student has worked hard to compose those paragraphs. Who knows it might help me understand the program better; and I may be able to give a more suitable grade." The comments turn out fine, and the grader deducts no marks for the long comments.

This grader goes on grading other programs. Near the end of the grading session, the same grader grades another program with equally long comments. However, at this point in time, the grader is already very tired, sleepy and has red eyes. The grader thinks: "The comments are too long, my eyes are already tired, and I will not bother to read them. Instead, I will punish the student for attempting to torture me further, by deducting three percent of the final mark."

Besides the abovementioned inconsistency, there is also the inconsistency caused by the fact that different students doing the same assignment may be graded by different graders. With the different standards, such as indentation preferences, it would be difficult, if not impossible, to have all scripts graded in a fair and consistent manner.

### 2.6. Difficult to regrade

It is not easy to grade a script, and it is even more difficult to have a graded script regraded; for example, to counter-check whether the grade assigned is fair.

### 2.7. Less programming assignments

Only few programming assignments can be given in a semester, since graders can be tired, as previously illustrated. The number of programming assignments that can be given in a semester is directly proportional to the number of graders, and is inversely proportional to the number of students taking the module.

Modules with emphasis on programming would suffer, because with the same number of graders and a big increase in the number of students, there will be fewer programming assignments; and the modules will not be able to achieve their goal.

### 2.8. Grading is slow

Grading takes too much time. Most graders would need to print the scripts, grade them, and key in the grades manually.

### 2.9. More manpower

A grader cannot grade too many programs and still meet a given deadline. Therefore, the increase in the number of students implies that there will be more programs to be graded, which in turn implies that more graders should be hired and problems such as lack of manpower or lack of financial resources to hire more manpower would arise. There is no need for many graders. One who has access to the system will do, thus, eliminating financial and manpower problems.

### 2.10. Possible special treatments

If a grader personally knows the student being graded, the grades may be affected. For example, it may be the case that a female student has a boyfriend who happens to be teaching the programming module she is taking.

Even if the student is identified by a number, for example the matriculation number, the grader knowing the student would take efforts to find out and memorize the student's identification number; and would be able to give special treatment to this student.

Special treatments are not good. Consider a grader who has an unsettled personal matter with a student. The grader would be able to penalize the student in terms of marks, because there is often no counter-checking by other graders.

### *2.11. Less participants*

Manual grading only the submissions from students taking a module is difficult let alone having students not taking the module to also participate in attempting the programming assignments for their own interest.

## 3. Automatic grading

### *3.1. The model*

To attempt to alleviate the problems outlined in the previous section, we intend to have an online judge system mounted on a dedicated machine. It knows about all the programming assignment questions. Students submit their programs electronically to this machine. The system will check the correctness of the submitted programs. A dedicated machine would be reasonably secure, since it would have its permissions set correctly. With it, we prevent some problems that might occur when an unsuspecting grader executes a student's malicious program.

This dedicated machine would ideally have all the permission settings set correctly, such that it is impossible for a submitted program to do the malicious actions as described previously. This machine would also handle the killing of the slow programs.

The model we are proposing is as follows. Students submit their programs. The machine receives, logs, and grades them. With all the permission settings set correctly, any attempt to do malicious actions such as removing all files would not succeed. Note that even if it succeeded, the files owned by staff members would not be affected since this would be a dedicated machine.

As we have noted earlier, the programs we are interested in grading are algorithmic in nature, and can thus be used as if they were filters. They take in some input, do some algorithmic processes, and write out the results. Our system will provide the input through the standard input handle and take in the output from the standard output handle.

The programs themselves will not be looked at. Instead, their output will. A program will be accepted only if it produces the correct output within the specified time limit. It will be rejected otherwise. A program's output will not be examined at all if the program does not terminate within the time limit.

A question will be phrased in terms of how the input looks, and what each of the components mean; how the program processes the input, and how the output should look. Parsing of input is usually not the main focus of the problem, the input format is usually very simple and the input data is guaranteed to match the specifications. In other words, there is no need to check, for example, whether a number in the input is indeed within the prescribed range.

In order to help students understand the problem and to test their solution, a sample test set, consisting of the input and the corresponding output, is provided. This would instantly give an idea on the format of the input and output.

In order to grade their solutions, however, there has to be other pairs of secret test sets. The secret test sets would weed out programs that would always display the sample output. The secret test sets have to be secret. Otherwise students would submit a program that would check whether the input is that of the first test set. If so, the output of the first test set would be displayed as a result. The same for the rest of the test sets would be done.

There are two claims on using multiple test sets. One is that if a program fails on any one test set, it is not a working program. The other is that each test set is there for a specific purpose — perhaps to test different boundary conditions. Imperfect programs that passed some of the test sets are to be given partial credits.

We go by the second claim, with the idea that when the first is desired, we can always use just one test set with multiple test cases, and amend the input/output specifications in the question that all of the test cases must be processed.

### *3.2. Advantages*

The main advantage the automatic grading method has over the manual grading method is that automatic grading addresses the flaws in manual grading.

#### *3.2.1. No standard solution*

There is no need for a model answer on which to base the grading. Any answer would be graded based on its actual correctness. A non-standard algorithm that is nevertheless correct would be properly graded as correct.

#### *3.2.2. No aesthetical aspects*

The proposed grading system would not grade on aesthetical aspects such as indentation, identifier naming, modularity, style, or comments. Students should not be required to spend time beautifying their source files. On the other hand, most students would agree that having some consistent personal standard on indentation, identifier naming, modularity, style, and comments would help them come up with a working program faster and easier. It is merely a convenience to them.

#### *3.2.3. The running time limit*

After a program has consumed enough time, it will be killed automatically without requiring human intervention. The limit is precise. If it is set at 30 s, it will never be 29.9 or 30.1 s instead.

#### *3.2.4. Defense against malicious programs*

This dedicated system is supposed to be secure enough to deter the submission of malicious programs.

There are essentially two possible approaches to this:

1. Source files are scanned for unwanted tokens, and such source files are rejected. The advantage of this approach is that if we also log rejection, we would be able to find out

which students were trying to hack. The disadvantage would be to introduce a new language to the system, it has to be learnt thoroughly to find out what functions can be misused. It would be very difficult to introduce some languages such as Perl, which can execute any statement constructed in a string before evaluation.

2. The environment has to be very secure. Calls to unwanted functions will fail, but most programs just silently ignore a function's return value. This is the approach we are using.

#### *3.2.5. Consistency*

The grading is done automatically, using the same program and under the same constraints. There is neither a notion of tired eyes, nor different moods.

#### *3.2.6. Rejudgeability*

It is easy to rejudge a question, for example, when an error is discovered in the secret test sets.

#### *3.2.7. More programming assignments*

Since grading is no longer a chore, more programming assignments can be given. This means more practice for students. Modules that emphasize programming would be able to maintain this status.

#### *3.2.8. Faster grading*

Automatic grading will generally be much faster than manual grading. In fact, if manual grading still required the execution of a program, automatic grading would always be faster.

#### *3.2.9. Less manpower*

There is no need for many graders. One who has access to the system will do. Thus, there is no problem with lack of financial resources to hire additional manpower.

#### *3.2.10. No special treatments*

There may be no special treatments to any particular student because such special treatments would easily be spotted by the grader who has access to the system.

#### *3.2.11. More participants*

Students who are not taking the module can also join in the fun and attempt the programming assignments. Their submissions will be judged, just as if they were taking the module, because the increase in the number of students attempting the programming assignments would not become an issue.

This way, students who have passed the module can still get even more practice, and students who have not passed the module will have an incentive to answer the question because it is proven to be feasible.

## 4. Implementation details

In this section, we discuss how we implemented the initial prototype of the system.

### 4.1. Machine

We used andy.comp.nus.edu.sg, a 450MHz processor.

### 4.2. Operating system

Of the many operating systems, we picked Linux, mainly because it is available for free; and it can provide the basic capabilities we need to implement this system.

### 4.3. Programming language

Certain parts of the project were done with C and shell script, but Perl was used predominantly throughout the system. Perl is very suitable to quickly construct a workable system like our online judge.

### 4.4. Technical issues

#### 4.4.1. Mail

Rather than implementing our own submission mechanism, we used the tool that is already available to us, which is e-mail.

Initially, users of our system would submit their programs in an e-mail message, with certain headers and a password to identify themselves.

To make it simpler, we later decided to build a wrapper that would still essentially e-mail the program with the correct headers.

#### 4.4.2. Web

The online judge has its own webpage. Through this webpage, one can check, among others, the grading progress and the statistics of a user. In addition, normally, the judgement is e-mailed to the user. The non-Java judge's webpage is available at: http://andy.comp.nus.edu.sg/judge.cgi/home. The Java judge's webpage is avalaible at: http://andy.comp.nus.edu.sg/cs1102.cgi/home.

#### 4.4.3. Queues

There are two main queues in the program. When a mail is received, it will be saved and queued into the "new mail queue", and the judging process will be triggered. The judging process will move the mails in the "new mail" queue to the "awaiting judgement" queue so that the insertion of new mails into the "awaiting judgement" queue does not interfere with the judging process.

#### 4.4.4. Judgements

A program may be judged as Compile Error if it does not compile properly. The user is notified of such cases.

For Java, we had an extra error message, Class named main not found, since we have to fix the class name because there is no easy way to determine what the class name is.

If a program compiles, the compiled program is executed once for each test set to generate each of the following individual judgements:

1. Accepted: the program terminated within the time limit, and produced correct output in the correct format.
2. Format Error: the program terminated within the time limit, and produced correct output, but in the wrong format, for example, with an extra blank space, or with lowercase letters instead of uppercase letters.
3. Wrong Answer: the program terminated within the time limit, but produced incorrect output.
4. Signal...: the program caught a signal.
5. Time Limit Exceeded: the program did not terminate within the time limit, and it was killed with signal 9 (SIGKILL).

The earlier version of the judge could not identify Format Errors. If there were at least two different test sets with different judgements, the user will get "Partial" instead. Such an outcome would summarize concisely what happened for each test set. For example, [.fxt] (1 correct) would mean that there are four test sets, and the program's results were Accepted, Format Error, Wrong Answer, and Time Limit Exceeded, in that order. If all test sets have the same judgement, the judgement name is displayed. For example, Accepted instead of Partial [...] (4 correct).

#### 4.4.5. *Multi-linguism*

Our prototype supports, among others, C, C++, Perl, Pascal, and Java. More languages can also be added at a later time.

### 4.5. *Security issues*

We have noted previously that we can receive a malicious program at any time. We have also stated that scanning source files for unwanted tokens such as unlink may not work because:

1. It is far too easy to forget about a keyword. A hacker knowing the loophole might just make use of it.
2. It is very difficult to introduce a new language to the server. More often than not, the language has to be thoroughly learnt in order to extract a list of keywords that are unsafe. Again, it is far too easy to forget about a keyword.
3. It is virtually impossible to introduce an interpretive language that has the ability to execute a string it creates during runtime. In Perl, one would construct a string that has malicious commands, and then evaluate this string. Keyword banning will not identify it because the string can be constructed in a very obscure manner, perhaps by first constructing the commands that construct this string into another string.

We adopted a different approach. We wanted to be able to introduce new programming languages. Sometimes it may be difficult, but it should be possible. The above approach would not work on languages like Perl.

We made our environment very secure, and only execute the programs under that secure environment. We do not care about what the program does. But when it attempts to do such unexpected actions, it should fail.

Ideally, this involves a major rewriting of the kernel, to weed out the unwanted functions (such as networking) in order to prevent a program from doing something nasty with these functions.

#### 4.5.1. Using chroot

There is a very useful system function called chroot, that will set a certain directory to be the root directory. From that point on, the process and all its children can never find out that there are directories outside the chosen subtree. They will not be able to find the parent directory of the designated new root directory.

Many security-aware programs make use of it. An example is the ftp daemon ftpd. To someone who does an anonymous login, the directories in the file system is overly simplified. However, it is actually fake. What outsiders see as /etc/passwd is actually /home/ftp/etc/passwd.

There are catches, however. Surprisingly, these are not documented. The most serious one is about the libraries. Many compiled programs silently make use of libraries. They expect these to be located at a certain place. For example, if a program needs /lib/libc.so.6, it has to be accessible as /lib/libc.so.6, or else it will fail to execute at all. The file is usually there, but once the root directory is changed, what /lib/libc.so.6 refers to will be different. It will refer to the file libc.so.6 under the directory lib under the designated new root directory, not the standard /lib which may be way up the tree. We managed to do this by finding *.so and *.so.* in all directories and copying them with the path retained.

The second catch is that chroot can only be called by the super user. After the chroot, one may simply forget to drop the super user privileges. We handled this as well.

#### 4.5.2. File access

A submitted program will run as a non-existent unprivileged user. As such, it can only access world-readable files such as the system libraries and the compiled version of itself. It will not be able to write into any directories in the chrooted subtree. There is no directory in which it can produce temporary files. All computations must be done in the memory.

It will not be able to look outside the chrooted subtree. The sample input and expected output files, the judge's log files that indicate how many times the user has submitted a Wrong Answer for that problem, and all other sensitive information is kept outside this subtree. There is no sendmail program that it can use to e-mail the input data to the student.

The program can attempt to remove, rename, or overwrite files, or do whatever it likes. It will never have the required permission to do so. On the other hand, interpreters such as java will exist and be accessible.

#### 4.5.3. Environment variables

Environment variables, such as PATH, are conveniently used to pass data from a program to its children. However, it may also expose some sensitive information. We decided to clear all the environment variables before executing the submitted program.

#### 4.5.4. Resource limits

We can limit the usage of resources using the ulimit command. We do this to prevent the misuse of the server. We set the hard limits so that the program cannot go beyond the prescribed limits. For example, we limit the number of user processes to one, so that the program cannot launch another process.

#### 4.5.5. Time limit

We limit the execution time using ulimit as above. The program is allowed to execute until its user time and system time total more than one more second than what we set.

This is fair because it does not count the time when other processes, not necessarily related to judging, are running. One such process is the web daemon. When the program overtimes, it will be terminated using the SIGKILL signal, which guarantees the termination of the process.

However, this has its problems. Consider a user who submitted a program that does sleep(86400);. The program would have slept for a day without consuming any user time, and consequently, the judge would also have spent a day waiting for the program to terminate.

To prevent this, we decided to also limit the real time to a certain multiplicative factor of the user time. This limit is enforced manually. We will have two processes: one is the program being judged while the other is the judge itself counting how long the process has been executed. The judge will stop its counting when the submitted program terminates. If it has not stopped counting after the real time limit has been exhausted, it will send a SIGKILL signal to the judged process, guaranteeing its termination. Since the user is limited to executing only one process, we only need to send the signal to one process.

#### 4.5.6. Signals

Since the submitted program is run as a non-existent unprivileged user, it will be the only process the user is running. Consequently, it will be the only process which it can send signals to. In particular, it cannot terminate the judge.

#### 4.5.7. Program size

There is an upper limit for this. Programs can never exceed a certain size. This is done to protect the system from running out of disk space. As a side effect, it becomes harder to submit a precomputed data set.

#### 4.5.8. System load

Since the judge is running on is a dedicated machine, it is very idle. The judge's existence will hardly overload the system. We set the process priorities such that the judge system does not interfere with other processes running on the server. We found that the system works well by setting the judge's real-time-counter program's priority very low, and the program being judged to be slightly higher.

## 5. Applicability

### 5.1. CS3233 Competitive Programming

We tried the automatic grading system on CS3233 in July–November in year 1999 and 2000.

#### 5.1.1. Module description

This module aims to prepare students in competitive problem solving in computing — covering techniques for attacking, solving, and writing computer programs for challenging computational problems. It also covers algorithmic and programming language toolkits used in problem solving supported by the solution of representative or well-known problems in the various algorithmic paradigms. The course is a "hands-on" course with weekly problem sets to be solved.

#### 5.1.2. Applicability

The module had many programming assignments. Twenty of the questions used the online judge. These are questions numbered 105–124 in http://andy.comp.nus.edu.sg/judge.cgi/problems. In short, the online judging system is applicable in CS3233.

### 5.2. ACM Programming Contest selection

We tried the automatic grading system for ACM Programming Contest selection in October 1999.

#### 5.2.1. Event description

The contest is a two-tiered competition among teams of students representing institutions of higher education. Teams first compete in regional contests held around the world from September to November each year. The winning team from each regional contest advances to the ACM International Collegiate Programming Contest World Finals, typically held the following March to mid-April. Additional high-ranking teams may be invited to the World Finals as wild card teams. More info at http://acm.baylor.edu/acmicpc/Info/.

#### 5.2.2. Applicability

We needed to have very similar contests to select the students who eventually would represent our university in the regional contests and possibly the world finals. This selection was conducted with the help of our online judge.

Incidentally, this selection is part of CS3233, and used the same online judge. Questions 118 and 119 were meant for trial, and 120–124 for the selection. In summary, the online judging system is applicable in ACM Programming Contest selection.

### 5.3. CS1102 Data Structures and Algorithms

We tried the automatic grading system on CS1102 in January–April 2000.

*5.3.1. Module description*

The aim of this module is to give a systematic introduction to data structures and algorithms for constructing efficient computer programs. Emphasis is on data abstraction issues (through ADTs) in the program development process, and on efficient implementations of chosen data structures and algorithms. Commonly used data structures covered include stacks, queues, trees (including binary search tree, heap and AVL trees), hashing tables, and graphs; together with their corresponding algorithms (tree and graph traversals, minimum spanning trees). Simple algorithmic paradigms, such as generate-and-test (search) algorithms, greedy algorithms and divide-and-conquer algorithms will be introduced. Elementary analyses of algorithmic complexities will also be taught, and laboratory work is essential in this course.

*5.3.2. Applicability*

The module had many programming assignments in Java. These questions are supposed to emphasize the use of data structures in solving algorithmic questions. There were sixteen questions, and all of them used the online judge. These are questions numbered 100–115 in http://andy.comp.nus.edu.sg/cs1102.cgi/problems. In short, the online judging system is applicable in CS1102.

## 6. Summary

We have shown that in manual grading, correctness is often sacrificed in favor of other aspects. For example, programs that look correct, but are actually flawed, and one that has a missing semicolon that prevents it from being compiled, may get more marks than programs that do work as expected but do not visually look good.

Students should not be led into thinking that programming is all about making programs that are visually attractive but not necessarily correct. Therefore, we cannot forever remain using the manual grading method.

We have also shown that automatic grading would overcome some of the deficiencies of manual grading. However, this requires careful implementation of the system; and although we have given it our best try, there are still many security holes that will need to be patched before we actually start using it.

Based on our testing, we conclude that automatic grading is indeed applicable to modules that emphasize programming.

## 7. Implications

We have successfully tested the online judge on two courses. For the Competitive Programming course, the online judge is indispensable. For the Data Structures and Algorithms course, the system resulted in substantial savings in manpower, consistency in grading, and increase in the number of programming assignments for the students.

However, the usage of the online judge is not suited for every course. It is most suited for courses with programming assignments that are set in a way where the results are exact and precise. In addition, the test data of varying difficulties must be set to make evaluation effective.

While instantaneous responses from the judge promotes effective learning, some students may get frustrated if they are unable to get all the test cases right. Correctness of a program which is based solely on execution of various data sets may also intimidate some students. These frustrations and fears can be partially alleviated by providing a library of old sample problems for students to get used to the system; by modification of the grading criteria to include the human element in grading; and by awarding partial credits by setting a number of simple test cases.

The judge also allows us to archive past problems, solutions and student submissions easily. On top of the effective reuse of past problems and solutions, analysis of students solutions can help in making the course materials more effective.

## References

Arnow, D. (1995). When you grade that: using email and the network in programming courses. In *Symposium on applied computing* (pp. 10–13).

Arnow, D., & Barshay, O. (1999). On-line programming examinations using Web to teach. In *Annual joint conference integrating technology into computer science education*, (pp. 21–24).

Kay, D. (1998). Large introductory computer science classes: strategies for effective course management. In *Technical symposium on computer science education*, (pp. 21–24).

Kay, D., Scott, T., Isaacson, P., & Reek, K. (1994). Automated grading assistance for student programs. In *Technical symposium on computer science education*, (pp. 381–382).

Reek, K. (1996). A software infrastructure to support introductory computer science courses. In *Technical symposium on computer science education*, (pp. 125–129).