An Investigation into the ACM Programming Competition (The SA contest of the ICPC)

Subtitle:

A Taxonomy and Evaluation of Algorithms and Solutions of Selected Problems

Submitted in partial fulfilment of the requirements of the degree of Bachelor of Science (Honours) in Computer Science

Rhodes University

Supervisor: A.J. Ebden

Researcher: Douglas Hobson

Abstract

The principal aims of this project are to develop a classification of the algorithms found in the South African ACM ICPC and to gauge their difficulty for humans to understand. The classification was developed and it was found that a very large number of problems require some form of mathematics to solve them. It was discovered that the type of algorithm needed to solve a problem does not determine the difficulty of the problem. This was unexpected and led to the development of a method of grading problem difficulty that is based on the time taken to generate a working solution to a problem. Three of the most difficult problems were solved in order to test the grading system.

November 2007

Acknowledgements

I acknowledge the financial and technical support of this project of Telkom SA, Business Connexion, Comverse SA, Verso Technologies, Stortech, Tellabs, Amatole, Mars Technologies, Bright Ideas Projects 39 and THRIP through the Telkom Centre of Excellence at Rhodes University.

Special thanks to my supervisor, John Ebden, for his help throughout the year, his encouragements, his corrections and his patience.

I would like to acknowledge Nick Pilkington for providing some very insightful tips which led to the solutions to some of the harder problems.

Thank you to Ryan Rainer for his many exhortations and encouragements to produce work that is excellent and pleasing to the Lord. Without your encouragement and support I doubt that I would have made it through the year.

"God delights in concealing things;
scientists delight in discovering things."
- Proverbs 25:2 (MSG)

Contents

1. Introduction page 4
2. Literature Review page 5
2.1 Introduction page 5
2.2 Other People's Classifications page 5
2.2.1 Howard Cheng's Classification page 5
2.2.2 Hal Burch's Classification page 7
2.2.3 Top Coder's Classification page 8
2.2.4 Comparison of Classifications page 9
2.3 Algorithm Resources page 10
2.4 Conclusion page 10
3. My Classification ` page 11
3.1 Introduction page 11
3.2 Classification of Algorithms and Knowledge page 11
3.3 Classification in Detail page 12
3.4 Design of Classification page 16
3.5 Conclusion page 17
4. Evaluation of Algorithms and Problems page 18
4.1 Introduction page 18
4.2 Method of Evaluation page 18
4.4 Conclusion page 21
5. Solutions to Selected Problems page 22
5.1 Introduction page 22
5.2 Solutions page 23
5.2.1 Problem 2006 – 2 page 23
5.2.2 Problem 2006 – 4 page 26
5.2.3 Problem 2006 – 5 page 32
5.3 Conclusion page 36
6. Conclusion page 37
7. Future Work page 39
8. References page 40
Appendix A
Appendix B

1. Introduction

This project is an analysis of the South African regional ACM ICPC (International Collegiate Programming Contest) from 2001 to 2006 inclusive.

Every year our university takes part in the ACM Programming Competition. The competition is becoming more and more popular with over 1700 universities and more than 6000 teams from all over the world participating in last year's event. [ACM International Website, 2007] Rhodes' teams do quite well on a regional level, and they do it on their own. There has been no work done at our university to formalise anything from the ACM competition. There is no training provided and no material available to help teams perform well. There is a need for proper treatment of the ACM Programming Competition at Rhodes University. This project will provide a start in that direction by providing a much needed investigation into the ACM competition.

The findings of this project promise to be of great help to future contestants and organisers

Something which must be noted is that this project is not about the computational complexity of the algorithms in the ACM ICPC, Big-O notation or anything of that sort. It is about the complexity of the algorithms for humans, not computers. Big-O notation is used to describe how difficult it is for a computer to solve the problem, it has very little to do with how difficult it is for humans to understand and solve the problem. There are times when this contributes significantly to the difficulty of a problem, for instance when the programmer has to find a way of adapting an algorithm that is usually very time inefficient to cause it to execute within the two minute time limit of the ACM ICPC, so you will find O(n) and the like mentioned from time to time, but that is not what this project is about.

2. Literature Review

2.1. Introduction

I have found several good sources in the area of algorithm classification. Most of the work is informal in nature, but very similar to the work that I am doing and so is particularly useful. In this chapter will have a look at several classifications that I have found, some specifically dealing with the ACM, another derived by Hal Burch from the USA Computer Olympiad [Burch, 1999] and the classification given by the Top Coder website [Top Coder, 2007].

The other thing that will be looked at in this chapter is the different types of algorithms already defined and sources that list and explain these. One such source is the Stony Brook Algorithm Repository maintained by Steven Skiena and another is the Dictionary of Algorithms and Data Structures maintained by NIST, the National Institute of Standards and Technology [Black, 2007].

These two broad categories of sources overlap a little. The classifications often include well defined algorithms and explanations of how those algorithms work similar to that found in the algorithm resources.

2.2 Other People's Classifications

2.2.1 Howard Cheng's Classification

This classification is derived from the ACM Programming Competition. It is very good was particularly useful. It was developed by Howard Cheng from the University of Lethbridge [Cheng, 2006]. Cheng has been heavily involved with the ACM Programming Competition as a contestant, a judge, a problem designer, a system administrator, and a local organizer. Cheng's classification does not give a clear description of any of the algorithms, but instead has provided a multitude of examples of each problem type. Fortunately several of the names that Cheng has given to categories in his classification are commonly used and are well defined elsewhere.

Classification from Howard Cheng:

Arithmetic
Backtracking
Big Number
Combinatorics
Data Structure
Dynamic Programming
Geometry
Graph
Greedy
Miscellaneous
Number Theory
Parsing
Permutations
Probability
Recursion
Search
Simulation
Sorting
Straightforward
String

Figure 2.1 Cheng's Classification

The categories that are in bold are the categories that I have adopted for use in my own classification. Some of them use a different name or fall into an overarching category. An example of this is Cheng's Arithmetic which I have called Mathematics. Those that are not bolded did not appear in my problem set or were considered unsuitable for inclusion. In particular, Cheng's Recursion is unsuitable as recursion is a fundamental part of programming and recursive solutions could be found for a very large number of the problems in my problem set. Including a category called Recursion would cause the results to give disproportional prominence to it as so many problems could be solved that way.

2.2.2 Hal Burch's Classification

The classification developed by Hal Burch from USACO [Burch, 1999] looks to be the most useful of all the classifications that I have come across. It has a fairly good list of algorithms as well as good descriptions of how each works. It also provides examples of problems that fall into each category. Another good thing about this classification is that the problem set that it was derived from is similar to the sort of problems given in the ACM Programming Competition.

Hal Burch's classification:

Ad Hoc Problems
Approximate Search
BigNums
Complete Search
Computational Geometry
Dynamic Programming
*Eulerian Path**
Flood Fill
Greedy
Heuristic Search
Knapsack
*Minimum Spanning Tree**
*Network Flow**
Recursive Search Techniques
*Shortest Path**
Two-Dimensional Convex Hull

Figure 2.2 Burch's Classification

Again, the bolded categories are ones that I have included in my classification. The four categories that have been italicised and had a '*' added are subsets of Graph Theory and fall into my Graph Theory category.

2.2.3 Top Coder's Classification

Most classifications were developed by one person or a small group working together. An exception to this is the Top Coder classification. Top Coder is a commercial site so it is likely that their classification is a little more reliable, but it is derived from problems found on the Top Coder site. This means that while it is useful for defining algorithm types, it does not cover the same set of questions as the ACM Programming Competition.

Classification from Top Coder:

Advanced Math
Brute Force
Dynamic Programming
Encryption/Compression
Geometry
Graph Theory
Greedy
Math
Recursion
Search
Simple Math
Simple Search/Iteration
Simulation
Sorting
String Manipulation
String Parsing

Figure 2.3 Top Coder's Classification

The main reason for the inclusion of the Top Coder classification is that it includes several mathematical categories, namely Advanced Math, Math and Simple Math which I have bolded in the table. Seeing these different categories that all deal with mathematics was what prompted me to divide my large Mathematics category into subcategories. This is discussed further in chapter 3.

2.2.4 Comparison of Classifications

Both Cheng and Burch's classifications include a catchall category. Cheng's has two: "Straightforward" and "Miscellaneous". Burch's has "Ad hoc". Both seem to think that there are usually a few problems in each competition that do not conform to any specific algorithm or pattern and each problem that falls into these categories needs to be solved on its own. The impression that I got from both Cheng and Burch is that these problems are fairly easy but occasionally appear challenging until properly understood.

There are several categories in Cheng and Burch's classifications that describe the same thing or are related in some way. The following table shows these relationships with comments to clarify where appropriate.

Cheng	Burch
Big Number	BigNums
Combinatorics	Knapsack (specific class of Combinatorics)
Dynamic Programming	Dynamic Programming
Geometry	Computational Geometry
Graph (Graph Theory)	Shortest Path Network Flow (all are subsets Eulerian Path of Graph Theory) Minimum Spanning Tree
Greedy	Greedy
Miscellaneous Straightforward	Ad Hoc Problems
Search	Heurist Search (not identical, but related) Note that Burch's "Complete Search" is not related to Cheng's "Search" at all but would be better known as Brute Force.

Figure 2.4 A comparison of Cheng and Burch's classifications

The Top Coder classification was not included in this table, despite it also having several categories that are related to categories in Cheng and Burch's classifications, because its problem set is quite different from ACM problems.

2.3 Algorithm Resources

The Stony Brook Algorithm Repository maintained by Steven Skiena of Stony Brook University [Skiena, 2001] looks as though it will be useful. Steven Skiena is involved with the Stony Brook ACM ICPC and the Stony Brook team has done well under his guidance [Skiena, 2001]. His repository contains several algorithms commonly found in the ACM ICMP with good descriptions of each algorithm type as well as code implementations in several common programming languages.

Another good resource is the Dictionary of Algorithms and Data Structures maintained by NIST [Black, 2007]. It is very extensive and has excellent descriptions of nearly every algorithm I have ever come across. It also often provides links to other sites to provide further clarification or examples of the algorithm being implemented.

Most of the classifications can also be regarded as algorithm resources, if to a slightly lesser degree than Skiena and Black's.

2.4 Conclusion

There are several classifications available, though they are informal, they are very good. They provide excellent reference material and give me something fairly reliable against which to compare my classification. The Top Coder classification [Top Coder, 2007], while unrelated to the ACM, is still valuable for the purposes of understanding and defining the algorithms involved. Cheng [Cheng, 2006] and Burch's [Burch, 1999] classifications are excellent resources and have been very useful to me in the design of my own classification by providing names for algorithms that are widely used and exposing me to the many formally defined algorithms that are needed for such a classification.

The two algorithm resources, the Stony Brook Algorithm Repository [Skiena, 2001] and the Dictionary of Algorithms and Data Structures [Black, 2007], provide suitable reinforcement of anything that is not clear in the classifications. They are also very nice to have for their formal treatment of algorithms that I can refer to in this sometimes informal area.

3. My Classification

3.1 Introduction

This chapter will cover my classification. It is derived from all the ACM ICPC problems from the South African regional competition over the last six years, from 2001 to 2006 inclusive. This chapter will include the classification itself along with more detailed descriptions of each category. After that there will be a section on the design of the classification describing how I went about actually making the classification.

3.2 Classification of Algorithms and Knowledge

This is my classification. It lists each of the categories in the classification along with how many problems from the problem set fall into each category. Some problems are in more than one category.

Name of Category	Number of problems
Backtracking	2
Big Numbers	4
Complete Search	8
Dynamic Programming	2
Encryption	3
Game/Puzzle	2
Graph Theory	7
Greedy	1
Mathematical ➢ Advanced ➢ Bases ➢ General ➢ Geometry ➢ Physics	18 1 4 7 4 2
Parsing	2
Straightforward	1
Tree	3

Figure 3.1 My classification along with the number of problems in each category

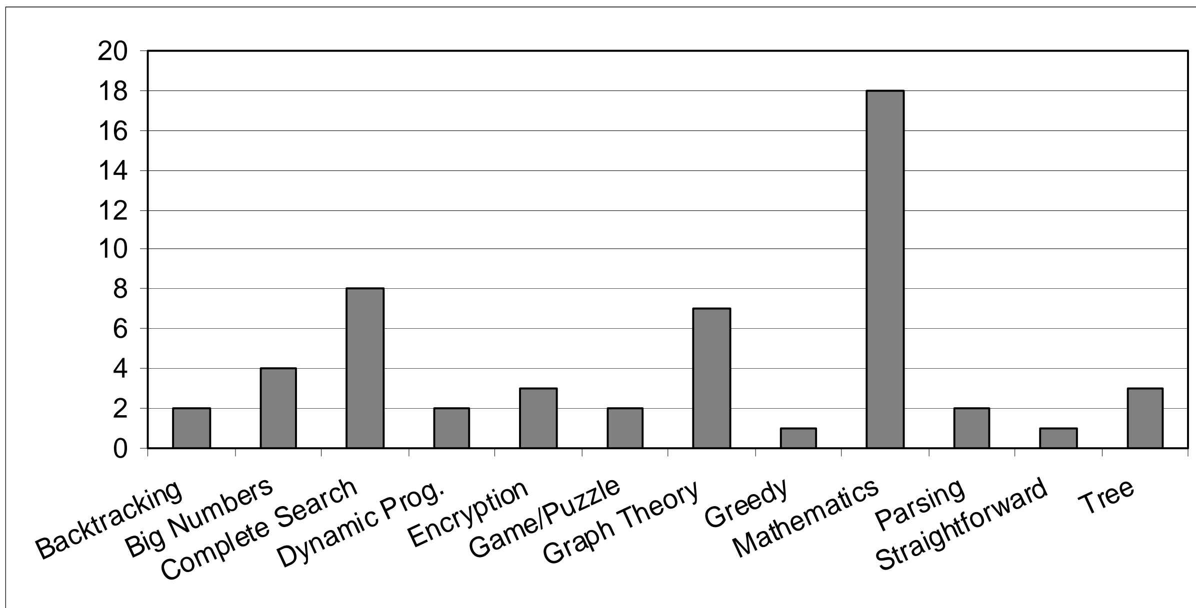

Figure 3.2 Number of problems in each category

The graph clearly shows that the number of problems requiring knowledge of mathematics is very large, fully half of the problems in the problem set fall into this category. This indicates that Mathematics is essential to anyone wishing to participate in the ACM ICPC.

Other categories that stand out are Graph Theory, Big Numbers and Complete Search. While dwarfed by Mathematics, they are significant when compared to the rest of the categories. Another category that deserves an honourable mention is Dynamic Programming. While I have only classified two of the problems as needing it, several of them could be done using Dynamic Programming techniques.

3.3 Classification in Detail

This section will briefly describe each of the categories in my classification. It will become apparent that not all the categories refer to algorithms. Instead they are more of a description of what is needed to solve a problem. In many cases an algorithm is needed, but sometimes it is knowledge that is needed. For example, Mathematics cannot be described as an algorithm, but is necessary for solving half of the problems in my problem set.

3.3.1 Backtracking

Backtracking algorithms are algorithms that make some choice which may be incorrect and if so go back to that point in the calculation and make the opposite choice. The diagram along side is of a decision tree generated by backtracking algorithms.

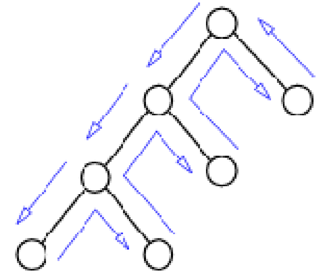

Figure 3.3 Decision tree used by backtracking algorithms

3.3.2 Big Numbers

Some problems require that numbers that are larger than all the standard data types be used. This is a programming technique that allows programmers to get around the problem.

3.3.3 Complete Search

The complete search method of solving a problem uses simple brute force to find the solution. It tries all possible answers, checking each one to determine if it is correct. The solution is often quite simple and relies on the computers power and speed to arrive at a solution rather than the programmer's brain and time.

If the number of possible solutions to be checked is less than something in the region of 10^8, then it is possible that a solution of this form will work. If the computer can solve the problem within the time constraints of the competition then the solution is good enough and the contestant can focus on other problems that perhaps need more time and thought put into them.

Often problems will not initially appear to be solvable with this method but the number of possible solutions can usually be brought down to a manageable number by recognising various patterns. If the order in which things are done is unimportant or if repeating the same operation several times produces the original input then it drastically cuts down on the number of solutions to be tested.

3.3.4 Dynamic Programming

This is a technique which looks for problems that are made up of sub problems or problems that need to make the same calculations repeatedly. It solves the sub problems or performs the calculations and then sores the results and uses them to build up the main solution. I have found that it usually has the very useful effect of moving a problems complexity or cost from time to space, making programs that would normally execute very slowly execute extremely fast. This is best described through an example.

A common way of producing the Fibonacci sequence is to use recursion:

Fib(n) = Fib(n-1) + Fib(n-2) with Fib(0) = 1 and Fib(1) = 1.

This is acceptable for small values of n, but as n increases, the time it takes to calculate Fib(n) increases exponentially. A much faster solution is to use dynamic programming and store the result from each of the sub problems. Here is some pseudo code that demonstrates this:

```
int[n] fib
fib[0] = 1, fib[1] = 1
for i = 2 to n – 1
        fib[n] = fib[n-1] + fib[n-2]
end for
return fib[n-1]
```

This will compute the value of the nth Fibonacci number in linear time but can start to use very large amounts of space for the array when n is large.

3.3.5 Encryption

Any problem that involves some form of encryption or decryption will fall into this category.

3.3.6 Game/Puzzle

This category is quite fun and interesting. I included this to indicate that basic (or sometimes in-depth) knowledge of a given game or puzzle would be very useful in solving the problem. It also describes problems where the mechanics of some real or fictional game are described in the problem statement and it is the task of the programmer to implement the game, a portion of the game or a program that can play the game.

3.3.7 Graph Theory

This is a broad category that covers all parts of graph theory except for trees.

3.3.8 Greedy

> "Greedy algorithms are **fast**, generally linear to quadratic and require little extra memory. Unfortunately, they usually aren't correct. But when they do work, they are often easy to implement and fast enough to execute." [USACO, 2007]

Greedy algorithms look for what seems to be the best local solution in the hopes that it will lead to the best global solution, and they do it very quickly. A fairly good description of a greedy algorithm would be Backtracking, without the backtracking, just make a choice and keep going while hoping it is the right choice. As the quote from USACO suggests, they are wonderful when they work, but they can be tricky to get right.

Only one problem from the 36 in the problem set looks as though it can be solved this way, but I suspect that there are others that I am not able to detect. I think this is because the flexible nature of the greedy algorithm and the many ways in which it could be applied coupled with its tendency to give the wrong answer if used incorrectly.

3.3.9 Mathematical

I have broken this category down into the subcategories of: Advanced, Bases, General, Geometry and Physics. These are all fairly self-explanatory except for Physics which could be argued should be in its own category. I have included it within Mathematics because at its core physics is just maths with a bit of meaning attached to the numbers.

Physics problems often have large equations in them. These equations are often differential equations or contain exponents. Sometimes this will lead to the equations being unsolvable analytically. Then numerical methods are called for to solve the problem.

3.3.10 Parsing

Problems that require that data be read in from the input file in a way that is not completely simple will be in this category. This does not refer to the IO operation itself, but to the processing of the input. For example, reading in the number "3512" from an input file is trivial, but reading in "three thousand five hundred and twelve" and still ending up with the integer value of 3512 is not.

3.3.11 Straightforward

I have also included a category like Cheng's called Straightforward to cater for problems that may be too simple to assign to any algorithm class or as a comment to add onto the classification of a

problem to indicate that understanding of the specified algorithm, while potentially useful, is not necessary to solve the problem.

3.3.12 Tree

This category refers specifically to those algorithms that require the construction and traversal of a tree data structure. This includes all the possible methods of traversing a tree, i.e. breadth and depth first searches. While they are separate algorithms, they are closely related. Therefore I have put them into one category.

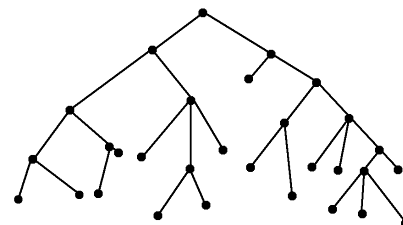

Figure 3.4 Tree Data Structure

3.4 Design of Classification

Here I will discuss how I went about doing my project and why I did it the way I did.

To make my classification I had three resources that I relied heavily upon. The first two are the classifications by Cheng and Burch which have been discussed in chapter 3, and the third is the set of ACM problems that I have been working with. I used other resources, but only sparingly; these three form the bulk of where my project came from.

First I developed a very basic classification of the algorithms found in the problem set. Then I began to work through the USACO website [USACO, 2007], solving problems and finding out about the algorithms in Burch's classification. While doing this I made changes and adjustments to the initial classification to include the new knowledge coming from the USACO website. Information from Cheng's classification was also assimilated.

After solving several problems from the USACO website I began to work on problems from the ACM problem set. I continued to change and update the classification as more problems were solved and the need for changes became apparent. I when through a cycle of solving problems and then making changes to the classification several times.

I chose to include various categories in the classification that are not algorithms at all but rather descriptions of what is needed to solve the problem. These categories would be Big Numbers, Game/Puzzle, Mathematical and Straightforward. Algorithms can also be described as knowledge

that is needed to solve a problem. For this reason I hypothesise that my classification is a classification of what knowledge is needed to solve the problems rather than a list of algorithms.

This is clear when looking at a few sample problems. Problem 2001 – 5 is a good example, if the programmer does not know how to handle the level of accuracy required (Big Numbers category) the problem is virtually unsolvable.

Another example is Problem 2003 – 4. The problem deals with data that has been encoded into frequencies or tones. The problem statement describes how to retrieve the data, it is rather complicated mathematics. If the programmer does not or cannot understand how to extract the data then this problem will become extremely hard.

So in many cases the category I assigned a problem to a category to describe the knowledge needed, and not just the algorithm needed, to solve the problem.

Both of the problem statements for these examples can be found in Appendix B.

3.5 Conclusion

Looking at the distribution of problems over the various categories, it is clear that mathematics plays a big part in the ACM ICPC. This could form the basis of future work; this is discussed further in chapter 7. The other categories that featured prominently are a good indication of the sort of preparation that would be necessary for programmers that wish to participate in the ACM ICPC. That more than just algorithmic and programming knowledge is needed to solve the problems in the ACM ICPC is an important finding of this project.

4. Evaluation of the Algorithms and Problems

4.1 Introduction

This chapter will look at how I evaluated the algorithms and problems. I will discuss several very interesting findings: that algorithm type does not determine problem difficulty, that problem difficulty is closely related to the time taken to arrive at a working solution and that problem difficulty is not an average of its parts but rather the weighted average of its parts.

4.2 Method of Evaluation

When I began this project I fully expected to find that some algorithms were simply harder than others and that problems involving them would be more difficult. I have found that this is not the case. There are some algorithms that are less well known than others, and some are a little more complicated. However, once a previously unknown algorithm has been studied and understood then it ceases to have and impact on problem difficulty. A problem can be made very easy or incredibly hard and still use the same algorithm.

I have found that these are significant factors relating to algorithms that make problems difficult:

- Is there a standard algorithm that can be used or does a new one need to be designed?
- How obscure is it as to which algorithm should be used?
- How well does the algorithm suit the problem? Does it need lots of customisation to make it work?
- How many algorithms need to be used?
- If more than one, how are they related? Is one used inside another or are they sequential?

It is clear that none of these are affected by which specific algorithm is being used, although determining the algorithm type is mentioned. It is therefore not very useful to talk about the difficulty of a specific algorithm, but rather only of specific problems.

I have developed a method of grading problem difficulty. It incorporates the discussion of the previous few paragraphs but is based largely on time. Because the ACM competition has fairly tight

time constraints with a little under an hour available to solve each problem, problem difficulty is closely related to the time taken to arrive at a working solution. The longer it takes to solve a problem the greater the cost to the competition as a whole.

This leads to the following times being good indicators of a problem's difficulty:

- How long did it take to read and understand the problem?
- How long to identify/develop the algorithm?
- How long to code the algorithm correctly?

If any one of these three times is large then it makes the problem difficult.

This forms the basis for my method of grading problem difficulty. More complicated problem statements will cause the time taken to understand the problem to increase. If the required algorithm is well hidden, needs lots of customisation or is a combination of algorithms then it will increase the time taken to identify/develop the algorithm(s). If the algorithm is difficult to code or the input is given in a difficult to process format then the coding time will be increased.

This leads me to the three main sections that problem difficult can be broken down into.
The difficulty of

- understanding the problem statement (S)
- the algorithm identification/design (A)
- coding the chosen algorithm (C)

Each section is closely related to the time taken to accomplish the task described in that section. Sometimes these three sections overlap a little. In particular the sections dealing with algorithmic and coding difficulty interfere with each other. The algorithm can almost always does affect the coding and the coding may cause the algorithm to be modified, but they are still separate processes.

I have given each of these three parts a difficulty rating out of ten. Large numbers indicate greater difficulty. These numbers are based on the time taken to complete that part of the problem, with subtle difficulties within that section causing the value to be raised slightly. An example would be the problem statement of the problem discussed in 5.3.3. It would normally have received a very low difficulty rating but it was increased to 4/10 for the reasons described in 5.3.3.

Total problem difficulty is a combination of these three ratings. However it is not just an average of the three or even a simple weighted average.

It is rather a dynamically weighted average of the three. If part of the problem is very difficult then that part's contribution to the total difficulty should be larger than that of the other parts.

At first a simple average was tried as a way of combining the difficulty ratings of the three different parts of each problem. This did not work very well when the three ratings were very different form each other.

The problem discussed in section 5.4.2 (Problem 2006 – 4) illustrates this nicely. The problem discussed there actually has a fairly easy solution that is not hard to code, but only one team even attempted to solve it. This is because the problem statement is five pages long and is very difficult to understand quickly. While the algorithmic and coding parts of the problem were easy, understanding the problem statement was not. This caused the problem as a whole to become very difficult, far more difficult than the average of the three parts of the problem.

Following that, a maximum was tried. Taking the total difficulty of a problem to be the maximum of the three ratings is almost satisfactory. It solves the problems that taking a simple average caused, but still was deficient in one way which is best described through an example. Two problems, one clearly more difficult than the other, will end up with the same difficulty rating:

Problem statement:	9/10	Problem statement:	9/10
Algorithm design:	2/10	Algorithm design:	9/10
Coding:	3/10	Coding:	9/10
Total difficulty:	9/10	Total difficulty:	9/10

To get around this problem I decided to use the rating itself as the weighting factor. This caused the contribution from the less important smaller ratings to be diminished. This is done by multiplying both the numerator and denominator by the square of the numerator. At first I tried just the numerator as the weight but the smaller ratings still had too much of an effect on the difficulty. These values are then used to calculate the total difficulty of the problem. An example of how this is applied is on the next page.

Problem statement:	3/10	x 3 x 3 =	27/90
Algorithm design:	4/10	x 4 x 4 =	64/160
Coding:	7/10	x 7 x 7 =	343/490
Total difficulty:			434/740 → 5.9/10

4.3 Conclusion

The finding that the type of algorithm needed to solve a problem does not determine problem difficulty was surprising and interesting. It was the motivation behind the development of the method of grading problem difficulty. The other interesting finding was that problem difficulty is closely related to the time taken to complete the problem.

5. Solutions to Selected Problems

5.1 Introduction

In this chapter I will discuss several problems in detail. For each problem I will explain exactly what is needed to solve the problem, which parts of it are difficult and which are easy. I will actually solve the problems in this discussion and will include pieces of code that will help explain some of the more difficult or interesting parts. The complete problem statement and code for each problem discussed here can be found in Appendix A.
The problems are discussed as for the ACM ICPC. Their solution must have a runtime of less than two minutes and use less than 64mb of memory.

Please note this is not the only way to solve these problems, this is how I would solve them.

The problems that will be discussed are:

- 2006 – 2: Peasant (Green)
- 2006 – 4: Not quite ALE (Blue)
- 2006 – 5: Delay (Orange)

I have selected some of the most difficult problems, those solved by the least number of teams in the competition. This is based on this table from the South African 2006 ACM ICPC website [ACM, 2006] showing the number of correct solutions for each problem in the 2006 competition. I have bolded the columns that are related to these three problems.

	2006 – 1 Perfect Squares	**2006 – 2 Peasant**	2006 – 3 Sudoku	**2006 – 4 Not quite ALE**	**2006 – 5 Delay**	2006 – 6 Numbers Game
# of teams attempted problem	49	**11**	16	**1**	**20**	18
# of correct solutions	32	**0**	13	**1**	**4**	11

Figure 5.1 The number of correct solutions to the 2006 problems

Several problems from other years of the competition are also quite challenging and would make good examples.

5.2 Solutions

5.2.1 Problem 2006 – 2: Peasant

This problem was not solved by any of the teams in the 2006 competition and was particularly challenging.

Problem Statement:

> "A well-known problem is to determine the maximum number of queens that can be placed on a chess-board without any of them attacking any other. In this problem you have to answer this question for a new type of chess piece, the peasant.
>
> Unlike the other pieces, a peasant's valid moves change from time to time. You will thus be given the valid moves as part of the input. Each valid move consists of a pair (*R*, *C*), indicating that the peasant may move *R* rows forward and *C* columns right, provided that this does not take the piece off the board (*The values of R and C are further constrained later on*). Negative values indicate motion in the opposite direction. For example, a knight in chess would be described by the valid moves (+/-1, +/-2) and (+/-2, +/-1). Like a knight, a peasant moves directly to the target square, even if there are other pieces in between.
>
> The input will consist of multiple cases. Each cases starts with a line containing *N*, the width and height of the board. The second line contains *M*, the number of valid moves that a peasant has. The next *M* lines describe the valid moves for this test case. Each line consists of two space-separated integers, *R* and *C*, which have the meanings described above. The last test case is followed by a line containing only a 0.
>
> (*The paragraph is important.*)
> The following constraints are in place: $1 <= N <= 12$; $0 <= R <= 1$ and $-N < C < N$. For each case, each (*R*, *C*) pair is unique and not equal to (0, 0).
>
> For each case, determine the maximum number of peasants that can be placed on a board without any peasant attacking (i.e. being able to move directly to the square occupied by) any other; write this number to the output." (italics mine)

This problem statement is not very hard to understand. The only thing that needs clarifying is that a single peasant move will be of the form (x, y) and not (±x, ±y) as the example of the knight's moves could lead one to believe.

To solve this problem it appears as though one needs to test all possible arrangements of the peasants on the board to cater for any really strange combinations of moves in the test data. (It goes without saying that if there is test data that can break your algorithm, then that is the test data that you will be given.) On a 12x12 board this means that there are 144 positions that need to be tested for the first peasant and 143 that need to be checked for danger from the first peasant and then probably tested for the second peasant and so on. This quickly escalates and causes the runtime of the solution to grow larger than the two minute time limit.

The solution to this problem is found in the restrictions on the values of R and C. I have made italicised comments in the problem statement to draw the reader's attention to them. C can range over the whole length of the board, but R is restricted to either 0 or 1. This means that each peasant can only affect its own row and the row immediately after its own. This completely changes the problem and allows the board to be filled up sequentially row by row which drastically reduces the computational complexity of the problem.

This problem is a Complete Search type problem and is an excellent example thereof. It is clear that the whole board must be checked, and it is clear that this cannot be done within the two minute time limit. Then additional information reduces the size of the search to something that can be completed within the time limit.

The computational complexity can be further reduced because within a few rows the placement of the peasants begins to repeat (I have only ever seen it start to repeat after one row, but there may be input that could cause it to repeat after more rows). It is then possible to simply replicate the preceding rows over and over until the board is full. This step is not necessary, but could be done by more zealous programmers.

The full solution is in Appendix A. Including code here will not greatly aid in the understanding of the problem.

Analysis of problem difficulty:

The problem statement is not difficult to understand but is not wholly trivial. That it is possible to be confused about the possible values of R and C causes it to receive a difficulty rating of four.

The algorithm design/detection part of the problem is very hard. If one does not pay careful attention to the bounds of R then one cannot solve this problem. Even when one is aware of the bounds of R, there is lots of mental legwork that needs to be done to realize how it affects the problem.

The coding of this problem was fairly difficult. Performing all the checks for peasants endangering each other and placing the peasant was not trivial. It took quite a long time to determine how to perform the search through the board while making all the necessary checks. This contributed slightly to the algorithmic difficulty as well as the algorithm was adapted to make the coding a little easier.

Difficulty rating for each section:

Problem statement (S):	4/10	x 16	= 64/160
Algorithm design (A):	10/10	x 100	= 1000/1000
Coding (C):	8/10	x 64	= 512/640
Total difficulty:			1576/1656 → 9.5/10

5.2.2 Problem 2006 – 4: Not quite ALE

This problem is interesting in that it is actually not very hard. However, only one team solved it and that team was the only team to attempt it. This is because all the difficulty is in the problem statement.

The problem statement is far too long (five pages) for inclusion here so I will present a summarised version to facilitate easy reading. To get an idea of how daunting the unabbreviated problem statement really is I refer you to the unabbreviated problem statement in the appendix.

Problem statement:
You must design a program that will decrypt encrypted radio messages. The purpose of the messages is link establishment and as such they only contain information necessary to set up a communications channel between two radios. All the radios have a call sign of between one and three digits in length that is made up of the digits 0 to 9. These messages are presented to the programmer as strings of encrypted hex values and must be decrypted to ASCII plaintext.

Each message takes the form of a series of words. An example of a message in plaintext form is:

TO 1
DATA 3
TO 1
DATA 3
TIS 4
DATA 5
REP 9

What this means is that radio "13" is being called by radio "459". The address of the receiver is repeated before the sender identifies itself.

Each line in the above example is taken to be one word. Each word is represented by one byte of hexadecimal, each byte being divided into two parts, the upper four bits and the lower four bits. The upper four bits carry the command part of each word and the lower four carry the data associated with that command.

"TO 1" is one word. The command part is "TO" and the data part is "1".

TO is the command which designates the destination of the message.

TIS is an abbreviation of THIS IS and designates the source of the message.
DATA carries additional data for the preceding command. A message to “1” would not need a DATA command after the TO, but a message to “13” would.
REP is an abbreviation of REPEAT and this command takes on the meaning of whatever command precedes it. A message to ‘123’ would need a TO, a DATA and a REP.

Before the plaintext is encrypted it is converted to hexadecimal with each word becoming one byte as described above. The commands correspond to the following hexadecimal codes:

TO = 0x00
TIS = 0x10
DATA = 0x20
REP = 0x30

In the programming languages used for the ICPC numbers preceded by ‘0x’ are taken to be hexadecimal values. All the numbers here that are preceded by ‘0x’ are hexadecimal.
The data part of each word is simply the hex value of the data:

0 = 0x00
….
9 = 0x09

When these parts are logically ORed together using a bitwise-OR then we get the full hexadecimal word:

TO 1 = 0x00 OR 0x01 = 0x01
DATA 3 = 0x20 OR 0x03 = 0x23
And so on.

The hexadecimal values are encrypted by using the easily reversible XOR operation (bitwise), with one of three keys. The keys are ten words long. The nth word of each message is XORed with the $(n \bmod 10)^{th}$ word of the key with n of the first word being equal to 0. Which word is being used to encrypt each part of the message is called the phase which ranges from 0 to 9.

Key #	Key phase									
	0	1	2	3	4	5	6	7	8	9
1	0x2a	0x15	0x2a	0x22	0x15	0x11	0x1a	0x2d	0x25	0x1e
2	**0x17**	**0x2e**	0x15	0x17	0x0f	0x28	0x2d	0x1e	0x3b	0x2f
3	0x11	0x2c	0x13	0x0c	0x23	0x28	0x3f	0x16	0x07	0x32

Figure 5.2 The three possible keys (one in each row) used for encrypting the messages

So if using key number 2, then the first word of the message would be XORed with 0x17 and the second with 0x2e and so on. (bolded in the table above)

To help the receiving radio pick up the whole signal (it scans through several channels and could miss the first few words), the first word may be repeated a number of times. The phase of the key used to encrypt these preceding words is selected so that the first word of the real message uses phase 0 and all the preceding words use an alternating phase of 0 or 1. Here is an example that illustrates this where I have bolded the part that should be noted.

		Phase	
TO	**1**	**1**	**|**
TO	**1**	**0**	**| → preceding repetition of first word**
TO	**1**	**1**	**|**
TO	1	0	
DATA	3	1	
TO	1	2	
DATA	3	3	
TIS	4	4	

And so on.

You are presented with several lines of encrypted hexadecimal numbers with each line terminated by ‘0xff’ and must produce output similar to the original plaintext described above.

For example:

0x10 0x2d 0x04 0x2b 0xff

should be decrypted to

TO 1
TO 1
TIS 7
DATA 7

That concludes the problem statement.

To further help in the understanding of the problem I have made the table on the next page which depicts the progression from plaintext to encrypted message when using the second key to perform the encryption. I have bolded the preceding repetition of the first word to make it clearer and separated it and the other two parts to aid in the understanding of the problem.

Plaintext Message	OR	Hexadecimal Encoding	Phase	Word from key		Encrypted Message
TO 1	**0x00 OR 0x01**	**0x01**	**1**	**0x2e**		**0x2f**
TO 1	**0x00 OR 0x01**	**0x01**	**0**	**0x17**		**0x16**
TO 1	**0x00 OR 0x01**	**0x01**	**1**	**0x2e**		**0x2f**
TO 1	0x00 OR 0x01	0x01	0	0x17	XOR	0x16
DATA 3	0x20 OR 0x03	0x23	1	0x2e		0x0d
TO 1	0x00 OR 0x01	0x01	2	0x15		0x14
DATA 3	0x20 OR 0x03	0x23	3	0x17		0x34
TIS 4	0x10 OR 0x04	0x14	4	0x0f		0x1b
DATA 5	0x20 OR 0x05	0x25	5	0x28		0x0d
REP 9	0x30 OR 0x09	0x39	6	0x2d		0x14
→	→	→	→	→	→	

Figure 5.3 The process of encrypting a plaintext message

If all that can be understood then solving the actual problem is quite easy. What needs to be done is the exact reverse of what has already been done to the message. The only difficult part is determining which key is being used for the encryption and whether the phase starts with 0 or 1 (because of the preceding repetition of the first word).

This is easily solved when it becomes apparent that there are only six ways of encrypting the first word, three keys with either of the first two phases being used. Just test all of them and see which give valid output. It is possible that the wrong phase or key could yield valid output for only one word so expand the test to include the second word of the message as well.

On the next page there is C++ code which determines the key used for the encryption and returns the phase of the first word. If it fails to find a key and phase it returns that phase as 3, which is clearly invalid.

```
int findKeyPhase()
{
	//for all 3 keys
	for (int i = 0; i < 3; i++)
	//and the first 2 phases of each key
	for (int j = 0; j < 2; j++)
	{
		//XOR the first word of the message with a word from the key
		int temp = message[0] ^ lookup[i][j];
		//separate the command and data parts of the word
		int temp1 = temp & 0xf0;
		int temp2 = temp & 0x0f;

		//if the output is valid
		if(temp1 == 0x00 && temp2 < 0x0a)
		{
			//then check the second word of the message for valid output
			//when using the same key but the opposite phase
			temp = message[1] ^ lookup[i][!j];
			temp1 = temp & 0xf0;
			int temp3 = temp & 0x0f;

			if ((temp1 == 0x00 && temp3 == temp2) //check for repetition
			|| ((temp1 == 0x10 || temp1 == 0x20) //check for start of rest
			of message
			&& temp2 < 0x0a))
			{
				//set the phase and return the key
				phase = j;
				return i;
			}
		}
	}
	//return something that is clearly wrong if no key is found
	return 3;
}
```

Figure 5.4 C++ code that determines the key and phase used to perform the encryption.

Then once the key and phase have been determined, all that remains is to decrypt the message and convert it to plaintext. This is quite simple.

The only difficulty with the decryption is determining when the preceding repeated words stop. This can be checked for by looking for valid output when on phase 1 that is either DATA or TIS as these are the only commands that ever follow TO.

The decrypted hexadecimal needs to be ANDed with 0xf0 and 0x0f to get the command and data parts of each word respectively. The command part then needs to be converted to its plaintext. This is done by comparing it with a table of the hexadecimal codes and then outputting the correct value. The data part must just be printed as a single decimal integer which requires no further processing.

Analysis of problem difficulty:
The algorithm (A) and coding parts (C) are very easy. They require no special tricks or techniques. The code could be slightly difficult when designing the conditions for all the checks, but not overly so. The problem statement (S) however, is very hard. Even this shorter version that I have presented is quite hard to get through quickly. In competition conditions the time needed to read and understand this problem statement is very long and very costly. All but one team never even got around to the coding stage and I submit that this is because of the length and complexity of the problem statement.

Difficulty rating for each section:

Problem statement (S):	10/10	x 100	= 1000/1000
Algorithm design (A):	2/10	x 4	= 8/40
Coding (C):	3/10	x 9	= 27/90
Total difficulty:			1035/1130 → 9.1/10

5.2.3 Problem 2006 – 5: Delay

Only four teams solved this problem in the 2006 competition. None of the Rhodes teams solved it.

Problem Statement:

> "You are at a LAN party where nobody has brought a switch big enough for everybody to plug into. Instead, you have built a complicated network by connecting multiple switches together. What is worse, the network cables are of dubious quality, and introduce delays into the network when packets must be retransmitted.
>
> You are trying to determine whether the network problems will be serious enough to affect gameplay. Given a description of the network, determine the maximum delay between any two devices on the network.
>
> The network consists of N devices (computers or switches), connected by N-1 cables. **There is exactly one route from any device to any other device**. The delay between two devices is the sum of the delays of the cables (assume that switching does not introduce any delay)." (bold mine)

One clarification: for the purposes of solving the problem, whether a device is a switch or computer is irrelevant, they are only interested in the delay between any two devices.

This problem statement is easy to understand. I understood exactly what was wanted after a single reading. "Classic graph theory problem!" is what I thought after reading this. A weighted network, either find the minimum spanning tree and then compute the longest path through the tree or just find the shortest path from each node to every other node.

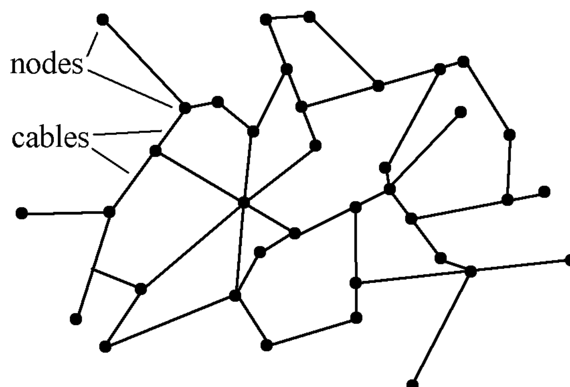

Figure 5.5 The network after one reading of the problem statement

However, we are also told that:

"2 <= N <= 100 000"

and that there are **multiple test cases**.

Here is when it becomes apparent that all the usual Graph Theory algorithms (Djikstra and others) will not work because the size of the network is too large to complete the calculation in less than two minutes. All the standard algorithms will compute the solution to this problem at best $O(n^2)$ which is much too slow when N can be as large as 100 000 and there can be several test cases.

Now is when you go back and read through the problem statement very slowly and think about each sentence. The important thing that contestants don't see at first see is this line which I bolded in the problem statement:

"There is exactly one route from any device to any other device."

This means that the network has no cycles and is already a minimum spanning tree, which makes this problem solvable.

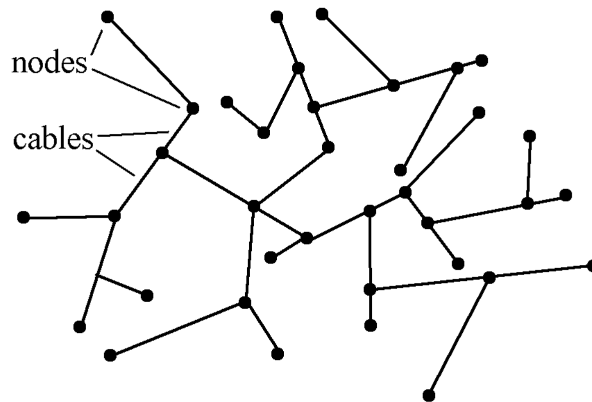

Figure 5.6 The network without any cycles (acyclic)

This diagram, Figure 5.6, can be viewed as a tree by selecting any node as the root and letting the rest of them hang off it as in Figure 5.7 (next page).

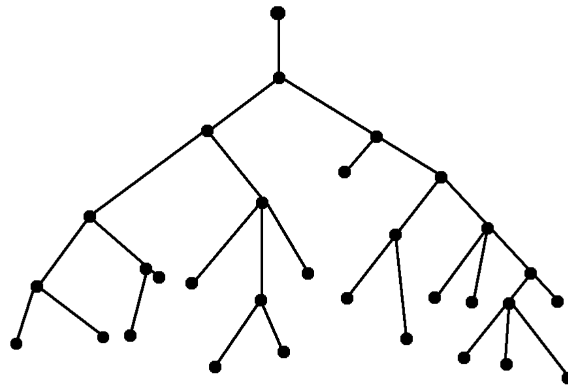

Figure 5.7 The network as a tree

An easy way to find the longest delay is to search the tree once for each node, with each node getting its turn as root and finding the longest branch. This is $O(n^2)$ and while it would be a little faster than algorithms used for graphs with cycles, it is not fast enough. Using a complicated depth first traversal of the tree a solution can be found that is O(n) and that will compute the solution within the time constraints.

I have programmed this traversal as a recursive function that calculates two things:

- The longest delay between any two nodes in each sub tree. (called *round* in my code)
- The maximum delay between the root of the sub tree and any of the other nodes in the sub tree. (called *tip* in my code)

It then passes these two values up to the previous level in the recursive traversal which in turn uses them to repeat this process.

The original call on the function then returns two values: the longest delay from the root to any other node and the longest delay between any other nodes in the tree. The solution to the problem is then the larger of these two values.

In my program, the tree was stored as an array of nodes, each with a linked list attached to it to store the locations in the array of that nodes children. This form of storage made the reading in of data from the input file fairly simple while giving the rest of the program easy access to the data.

The code that follows is the C++ code for the recursive function. The struct *info* is simply used to allow two values to be returned from the function. The nodes in the linked list that relates parent nodes to their children are structs called *cable.* Each *cable* contains the location in the array of the

other node that it is connected to and the delay associated with that *cable*. The function is called with the index of the root of the tree as its target.

```
info depthFirst(int target)
{
	int tip = 0, round = 0;

	// set curr equal to the first cable in the
	// linked list belonging to the node specified by target
	cable * curr = network[target].head;

	// while the linked list has not been exhausted
	while (curr != NULL)
	{
		// ask the current sub tree root for its info
		info temp = depthFirst(curr->dest);

		// calculate the longest node to node delay
		round = max(tip + temp.tip + curr->delay, temp.round);

		// calculate the longest delay from
		// root to any node in sub tree
		tip = max(tip, temp.tip + curr->delay);
		curr = curr->next;
	}
	info result;
	result.tip = tip;
	result.round = round;
	return result;
}
```

Figure 5.8 The code for the traversal over the tree

This solution is O(n) and should compute the answer within the two minute time limit. The 64mb space limit should not be a problem. Even with the maximum of 100 000 instances of the function running at once, it allows for each function call plus one element from the array to use up ~640 bytes of memory, which it does not.

Analysis of problem difficulty:

When the difficulty of this problem is analysed, the problem statement (S) is easy to understand (except for the easy to miss part that leads to the network being a tree), but the algorithm (A) needed to solve the problem is quite complicated. The algorithm is not just a standard algorithm that needed a little tuning to make it work; it is a whole new algorithm that is piggy backing on a depth first traversal. The coding (C) for this is also quite difficult in that it needs to be efficient and a data structure needs to be designed to suit the input data and the algorithm.

Difficulty rating for each section:

Problem statement:	4/10 x 16 = 64/160
Algorithm design:	9/10 x 81 = 729/810
Coding:	8/10 x 64 = 512/640
Total difficulty:	1305/1610 → 8.1/10

5.5 Conclusion

The method that I used to rate problem difficulty appears to work well. The difficulty of the three problems corresponds well with the number of correct solutions.

Something worth pointing out is that the problem statements are sometimes quite obscure. Often they could be improved by the inclusion of diagrams or charts to make them less obscure. There is the distinct possibility that the obscurity is deliberate and is being used as a way of increasing problem difficulty. If this is the case then it is to be decried. It would be better to design problems so that their difficulty is located in the algorithmic and coding parts of the solution rather than in the obscurity of the problem statement so as to better test the skills of the contestant.

6. Conclusion

This project has been an investigation into the South African regional ACM ICPC. The ACM ICPC is a very popular and challenging competition that has participants from thousands of universities all over the world.

Rhodes University has participated in the regional event for since its inception and students have performed admirably. However, there has been no work done at Rhodes in the area of analysing the ACM ICPC and there is no material available for training participants. This project has provided a start in this direction.

Similar work has been done by Hal Burch of the USA Computing Olympiad and Howard Cheng, a man who has been heavily involved with the ACM ICPC. Both produced classifications or lists of algorithms appearing in their respective competitions that have been very useful to me in the creation of my own classification of the ACM problems.

My classification is a classification of the knowledge needed to solve the problems in the problem set. The very large number of problems that need some form of mathematics to solve them indicates that mathematics plays a big part in the ACM ICPC. The other categories that featured prominently are a good indication of the sort of preparation that would be necessary for programmers that wish to participate in the ACM ICPC. That more than just algorithmic and programming knowledge is needed to solve the problems in the ACM ICPC is an important finding of this project.

One of the main objectives of the project has been to analyse the algorithms used in the ACM ICPC for their difficulty for humans to understand them. Therefore there has been little mention of Big-O notation or of computational complexity in this project.

While pursuing this objective of analysing difficulty for humans I found that the type of algorithm needed to solve a problem does not determine problem difficulty. This was unexpected and interesting. It motivated me to developed a method of grading problem difficulty that is based on the time that it takes a programmer (as opposed to the runtime of the solution) to solve a problem. I determined that each problem can usually be broken down into three separate stages, each with its own associated difficulty rating. These stages are understanding the problem statement, designing or detecting the algorithm needed to solve the problem and coding the solution. When these three stage-difficulties are combined they are combined as a weighted average where the weighting factor

is the square of the difficulty rating itself. This allows more difficult parts of problems to affect the difficulty more than the easier parts.

I have solved several problems and have discussed three of them in this project. I have used them to illustrate the sort of techniques needed to solve problems and to prove the usefulness of my method of grading problem difficulty.

7. Future Work

(a) Develop material for training ACM participants.

(b) There are thousands of other problems available for analysis.

(c) Investigate the mathematical categories in more detail. This is in view of their frequent recurrence and that the countries that do well internationally in the ACM ICPC have a strong focus on mathematics.

8. References

ACM ICPC International Factsheet, Published: unknown, Accessed: 31-10-2007, <http://icpc.baylor.edu/icpc/About/Factsheet.pdf >

ACM ICPC South Africa, Published: unknown, Accessed: 25-10-2007, <http://acm.cs.up.ac.za/2006/index.html>

Black, Paul E, ed., U.S. National Institute of Standards and Technology, *Dictionary of Algorithms and Data Structures*, Published: 2007-02-12, Accessed: 2007-06-24, <http://www.nist.gov/dads>

Burch, Hal, Published: 1999, Accessed: 2007-06-24,
<http://ace.delos.com/usacotext2?a=L3LjaPPIHL3&S=probs>
< http://ace.delos.com/usacotext2?a=pRwk7Xv3avS&S=greedy>
Login Required. Local copies at: <http://www.cs.ru.ac.za/research/g04h2708/USACO.html>

Cheng, Howard, *Problem Classification on Spanish Archive*, Published: 2006-12-17, Accessed: 2007-06-24, <http://www.cs.uleth.ca/~cheng/contest/hints.html>

Skiena, Steven, *Stony Brook Algorithm Repository*, Published: 2001-03-07, Accessed: 2007-06-24, <http://www.cs.sunysb.edu/~algorith/>

Top Coder, Published: unknown, Accessed: 2007-06-24, <http://www.topcoder.com/tc?module=ProblemArchive>

Appendix A

This appendix includes the following (in this order):

- Problem Statement of Problem 2006 – 2
- Solution to Problem 2006 – 2
- Problem Statement of Problem 2006 – 4
- Solution to Problem 2006 – 4
- Problem Statement of Problem 2006 – 5
- Solution to Problem 2006 – 5

Appendix B

This appendix contains the following problem statements:

- Problem 2001 – 5
- Problem 2003 – 4