Guide for Advanced Algorithms for Australia and New Zealand Algorithmics & Computing League Competition.

17-Aug–2012

Prepared By: Darran Kartaschew

Document Version: 1.0

Last Updated on: 17-Aug–2012

Contents

List of Figures

List of Algorithms

1 Introduction

1.1 About the Competition

The programming contests held in Universities across Australia and New Zealand, are part of the Australia and New Zealand Algorithmics & Computing League Competition and is used in conjunction with the ACM-ICPC competition. These competitions are aimed at challenging students in completing a set number of problems within the allocated time slot (typically 5 hours), with the winners in each location given some prestige.

In recent years teams from not only Universities taken part, but teams from TAFE and other educational institutions have taken part in the competition. Additionally teams outside of Australia and New Zealand such as those from the Phillipines have also taken part.

1.1.1 ANZAC 2012

The ANZAC 2012 competition takes place in 5 to 6 rounds each year and are sponsored by a local University and associated Faculty member. Typically, a single round will run for 5 hours (starting at midday for East Coast Australia), and at least 6 problems will be presented for completion by students.

All challenges require some form of problem solving skills or techniques and do require at least a basic understanding of different algorithms in order to complete the challenges, let alone to be competitive in the competition.

In order to compete within the competition it is recommended that 3 students form a team to work together on solving the challenges. Each team is only given 1 computer to work on, and all reference material brought into the competion must be in printed form only[1].

Scores are awarded for completed challenges (typically 1 point), and the time elasped from the start of the competition to accepted submission of the challenge is also noted. If a submitted challenge fails, then a 20 minute time penalty to added to the teams total time value.

As a minimum each contest will allow either C/C++ and Java, however additional programming languages may also be included. Typically C# has been allowed in recent years, due to the popularity of the language, especially as it is taught farily early in a students undergraduate degree.

Overall, the competition is designed to be challenging, fun and also students to advance within their field of study. It is also a great way to network amongst other equally capable students within the programming field.

1.1.2 ACM-ICPC

The ACM-ICPC competition is an International level competition sponsored by IBM, ACM and Upsilon Pi Epsilon, and contestants who make the world finals are often sort after by industry for later employment, as well as bringing notoriety and prestige to the University or College to which the constestants originate from. The regional component of the competition is typically held as the last ANZAC competition, as both competitions share the same tools, resources and rules.

The top two teams from each region (and in the case of Australia and New Zealand, the top team from Australia and top team from New Zealand), attend the International competition held annually in late March/early April in an overseas location. The 2012 ACM-ICPC Finals consisting of teams from all over the world was held in Warsow, Poland.

1.2 About this Guide

This guide is designed to give students some background knowledge of the environments utilised within the competition, as well as information on various algorithms needed to solve problems. The included

[1] The printed material requirement is to ensure that no copying of existing source code is allowed, only transcription of source code from written form

algorithms are by no means exhaustive, however represent the bulk of the algorithms that will be useful in completion of challenges.

This guide book is split into multiple parts:

1. Basic Source Templates that cover the basic frameworks needed for challenge submissions.
2. Basic Algorithms and techniques.
3. Advanced Algorithms.

All algorithms described will include:

1. A short statement on the algorithm and the intended uses, as well as other possible uses.
2. The pseudocode for the algorithm.
3. An actual implementation in at least 1 programming language. This will typically be in the form of a function or method call.
4. An example challenge that requires the use of the algorithm.
5. An example solution to the challenge.

Throughout the guide there will be notes on performance aspects of each algorithm, as well as helpful utility functions to make better use of the algorithm implementations. One example will be a function to convert an Adjancy List into an Adjancy Matrix used for different graph based algorithms.

1.2.1 Development System

The applications and source code snippets developed for this guide were performed on the following hardware and software combinations as noted below. Any performance measurements, in particular times required for certain functions reflect times as acquired with listed hardware and software combinations. Performance measurements will vary accordingly with different hardware and software combinations when performing your own performance measurements.

1.2.1.1 Hardware

HP xw4600 Workstation, with:

- Intel Core 2 Quad, Q9400 @ 2.66GHz (Quad core, 2.66GHz, 6MB L2 cache, 64bit enabled).
- 4GB RAM (4 × 1GB Reg ECC DDR2-800Mhz)
- 250GB 7200rpm HDD
- nVidia Quadro FX580 graphics card.
- Dual 20” LCD Monitors (1680x1050 resolution).

1.2.1.2 Software

Oracle Solaris 11 11/11, with:

- Solaris Studio 12.3 (C++)
- Netbeans 7.2 (Java)
- Java 6 JDK 1.6u26 or Java 7 JDK 1.7u5
- gcc 4.5.2 (C++)

Microsoft Windows XP x64, with:

- Mircosoft Visual Studio 2010 (C# and C++)
- .NET Frameowrk 2.0

1.3 License

2 Supported Competition Environments

The guide will focus on Java being developed in Eclipse, and C# being developed in Visual Studio. However there will be examples in C++ when appropriate. Most other IDEs have similar options, when used for development, debugging and/or profiling.

2.1 PC^2

The primary tool that allow students to submit their challenge entries to be judged in the PC^2 Software Suite. The application itself is developed by California State University, Sacramento for the purposes of programming competitions and has been adopted by both the Australia and New Zealand Algorithmics & Computing League (ANZACL) and ACM for their respective competitions.

An example of the Login Interface is shown in Figure 2.1.

Figure 2.1: PC^2 Login Screen

Once logged into the system, the following options are typically available:

Submit Run Allows you to submit a challenge entry to be judged, or alternatively to test your entry against some supplied sample data.

View Runs Allows you to view a history of submissions made to the judges.

Request Clarification Allows you to request a clarification from the judges about one of the challenges.

View Clarifications Allows you to see the responses to your requests for clarifications.

Options Allows you to access various options that control the clients operation. However this tab, only allows you to view the operational log of the client.

Most of the operations on the various areas are self explanatory, so won't be covered in detail. The main screen that competitors will utilise is the **Submit Run** tab as shown in Figure 2.2.

This screen has two main modes of operation, allow a competitor to test their submission against some sample input, or submit their source code to be judged. Both have similar operations, except the test has one additional step.

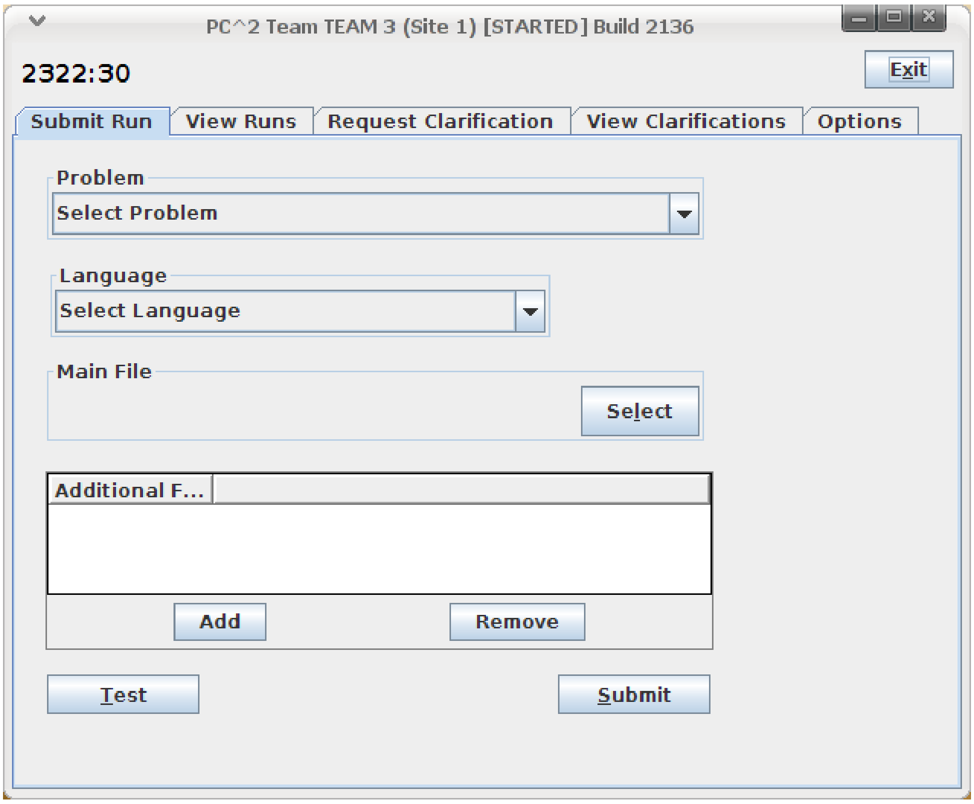

Figure 2.2: PC^2 Client Submit Run Tab

2.1.1 Submit Run

To submit a run for judging, perform the following steps:

1. From the Problem dropdown list select the challenge that you are attempting.
2. From the Language dropdown list select the programming language in the submission is written in.
3. Use the Select button to select the source code file for the submission. (Note: A single Source Code file is required, do not attempt to submit data files or executable files).
4. Use the Add Button to select any additional files needed to complete your submission. (Note: This is rarely needed).
5. Click on Submit, and Yes to confirm to have your submission judged.
6. You will receive a confirmation dialog confirming that your entry has been submitted.

Once you entry has been judged you will receive one of the following confirmations:

- Yes - Your submission was successful in passing all tests. Congratulations, you have been awarded one point.
- No - Your submission failed one or more tests.
- Time Overrun - Your submission took more time that allowed for the challenge.

An example Judge's Response Dialog is shown in Figure 2.3.

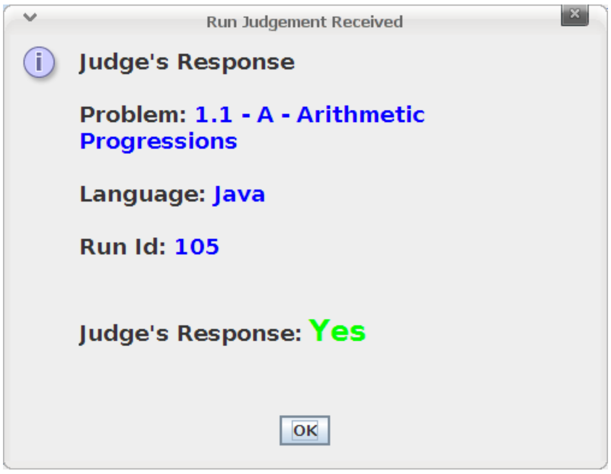

Figure 2.3: Sample Judge's Response

2.1.2 Submit Test

Before your submit your solution to be judged it is **highly recommended** that you perform a test run on your submission first, due to possible differences between the environment you utilised for development and the environment in which your submission will be run on the judges machine.

To test your submission first, perform the following steps:

1. Ensure that your source code file and the sample data files are in the same folder/directory on your system.
2. From the Problem dropdown list select the challenge that you are attempting.
3. From the Language dropdown list select the programming language in the submission is written in.
4. Use the Select button to select the source code file for the submission. (Note: A single Source Code file is required, do not attempt to submit data files or executable files).
5. Use the Add Button to select any additional files needed to complete your submission. (Note: This is rarely needed).
6. Click on "Test".
7. Select the appropriate sample input file in the open dialog box. (Typically the sample input fille will be `<challengename>_sample_in.txt`).
8. Wait for the output dialog and compare to the expected output.
9. If you are happy with your submission, then submit your solution for judging, by clicking on "Submit".

2.2 Software Languages

Currently the competition support the following software development languages with some variations between regional areas: Java, C++ and C#.

2.2.1 Java

Java is compiled utilising the Oracle Java 6SE JRE implementation, however future competitions may migrate to Java 7SE as Java 7 becomes more popular. (This guide will target the Oracle Java 6 SE JRE).

By default the competition will utilise the 32bit JRE, however this may vary as needed between each region. Additionally the Java compiler and JVM are run using default settings only.

2.2.2 C++

C++ (and by extension C) is compiled with an POSIX compatible compiler, typically being mingw on Windows. mingw utilises the GNU GCC compiler suite, and offers a near complete POSIX environment including the C++ STL.

It should be noted, that in some instances the Microsoft Visual Studio C++ compiler has been used within the competition, so it is best to check with the local staff supporting the competition which compiler will be utilised.

Irrespective of the C++ compiler and/or environment, it should be noted that the default compiler settings are utilised through the competition, so features including optimisation flags or 64bit operation are not enabled.

This guide will target a 100% pure POSIX environment.

2.2.3 C#

C# will typically be compiled by Microsoft Visual Studio 2010 with the .NET 2.0 Framework. However there may be variations to this, so it is best to check with the local staff supporting the competition which compiler and/or .NET framework will be utilised.

2.3 IDEs

At the moment there are no official supported IDEs utilised by the competition, however the majority of contestants utilise either Eclipse and/or Visual Studio.

Other IDEs or Editors commonly utilised by competitors include NetBeans (Java, C++), Code::Blocks (C++) and Notepad++ (Java, C#, C++).

2.3.1 Eclipse

Eclipse may be utilised to develop either Java applications or C++ applications (on provision the appropriate eclipse plugins for C++ are installed, and a compatible C++ compiler such as mingw is also installed).

There is no special configuration for Eclipse to be utilised within the competition. As all competition entries operate within a command line only interface there is no requirement for any GUI builder plugins to be present.

To utilise Eclipse for Java development, perform the following steps:

1. Start Eclipse, and switch to a Workspace that is empty, or has been designated for use for competition. (Use `File -> Switch Workspace` to move).
2. In the `File` menu, select `New`.
3. In the `New` dialog box, select `Java Project`. Click on `Next`.
4. Enter any name for the project name. Leave all other settings as default.
5. Click on `Finish`. This will create a basic project that can be used for developing submissions.
6. In the Package Explorer pane, right click on the `'src'` package, and select `New -> Java Class`. This will be the first submission that you will work on. When developing further submissions, simple start at this point and following the remaining steps.
7. The in `New Java Class` dialog, enter in the challenge name in the `Name:` field, and click on "`public static void main(String[] args)`" to select this option. Leave all other options as default, and click on `Finish`. (See Figure 2.5).

8. The new submission Java file will open in the file pane. (See Figure 2.5)
9. You are now reading to develop you submission.

All debugging facilities may be utilised within Eclipse with no restrictions.

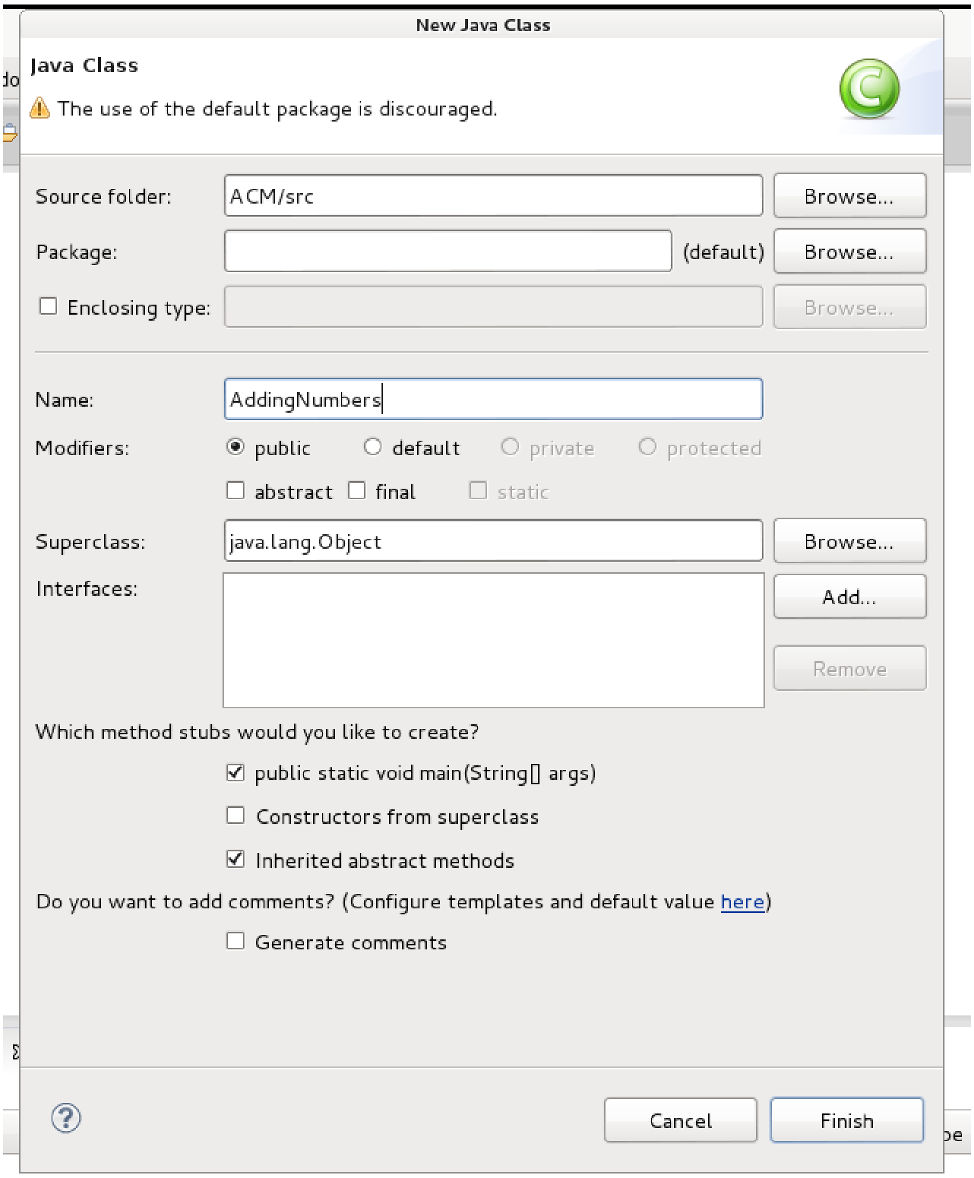

Figure 2.4: Eclipse - New Java Class

One item to note with the file structure of Eclipse and essentially all Java application development, you will need to note the exact location of the source files so are able to find them later in order to submit your solutions.

Using the example in Figure 2.4, note the "`Source Folder`" location, this is the location that your submissions will be located in, in this case `ACM/src/AddingNumbers.java`.

When testing your submission with PC^2 you will be required to either:

1. Copy your source code file to the same location as sample input files, or
2. Copy the sample input files into the same location as the source code file.

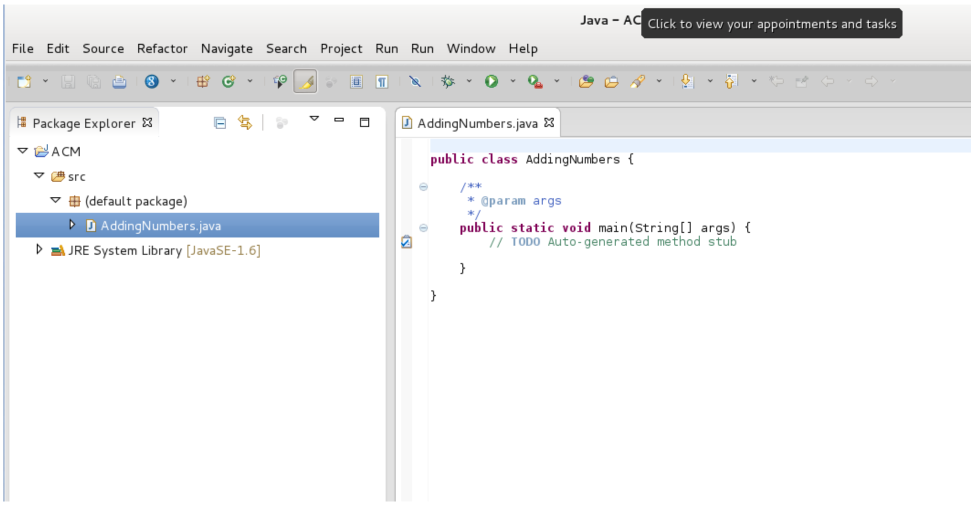

Figure 2.5: Eclipse - New Java Application

2.3.2 Visual Studio

Visual Studio is capable of working with a number of programming languages, including C# and C++. This guide will cover working with both langauges, noting the differences between the two.

2.3.2.1 C# There is no special configuration for Visual Studio to be utilised within the competition. As all competition entries operate within a command line only interface there is no requirement for any GUI builder plugins to be present.

To utilise Visual Studio for C# development, perform the following steps:

1. Start Visual Studio.
2. If you receive a "`Select Development Language`" dialog, select `Microsoft C#`.
3. Select "`New Project`" from the Start Page, or alternatively from the `File` menu.
4. In the "`New Project`" dialog, ensure that:
   (a) `Visual C#` > `Windows` is selected in the Installed Templates pane.
   (b) `Console Application - Visual C#` is selected in the Application Type pane.
   (c) `.NET Framework 2.0` is selected in the .NET Framework dropdown.
   (d) Enter the name of the challenge in the `Name:` field.
   (e) Note the location in which the project is being created.
   (f) Click on `OK` to build the project. (See Figure 2.6)
5. The new `Program.cs` file will be opened and displayed in the File Pane. (See Figure 2.7)

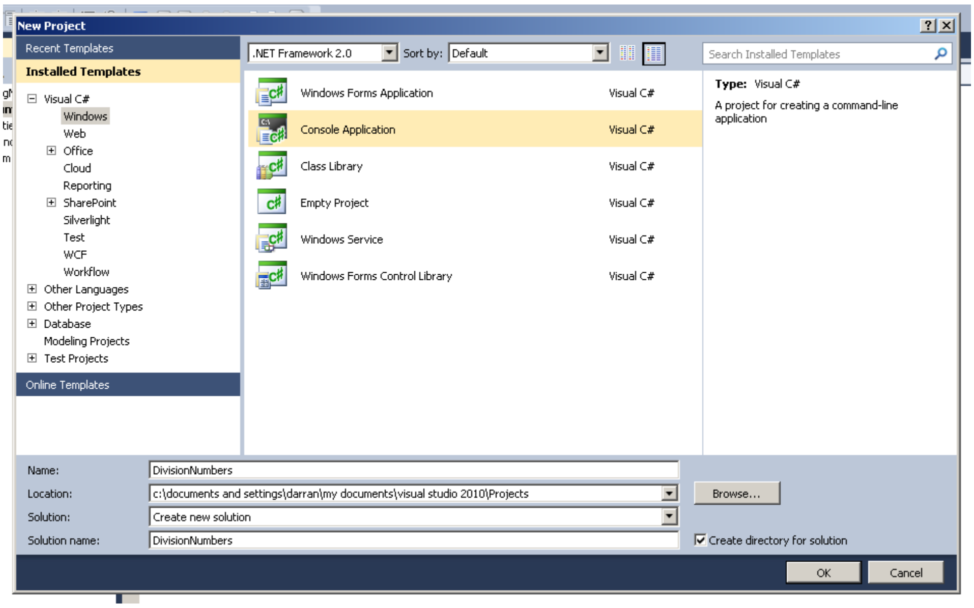

Figure 2.6: Visual Studio C# New Project

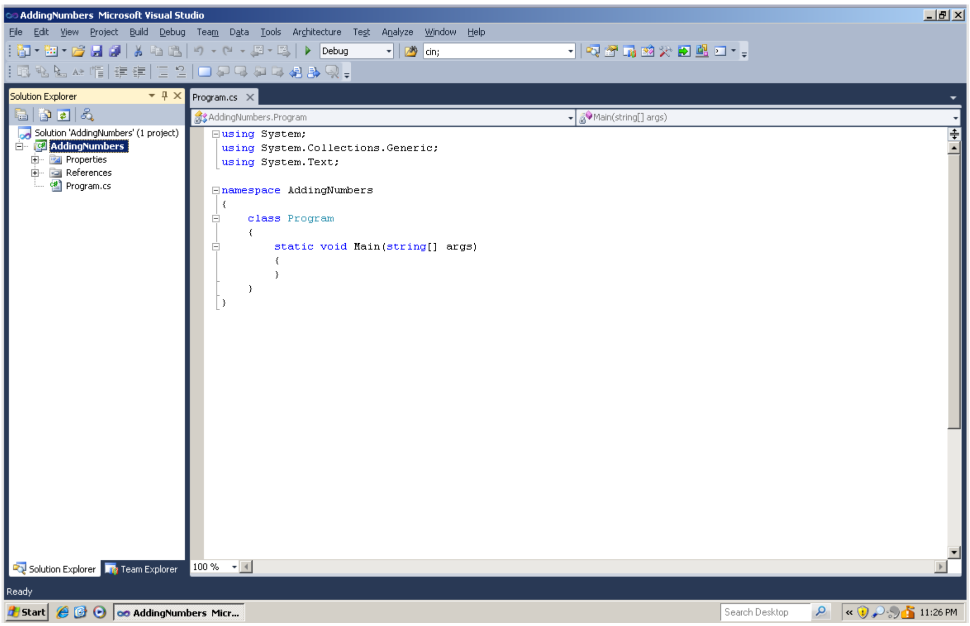

Figure 2.7: Visual Studio C# Program.cs

When testing your submission with PC^2 you will be required to either:

1. Copy your source code file to the same location as sample input files, or
2. Copy the sample input files into the same location as the source code file.

2.3.2.2 C++ To utilise Visual Studio for C++ development, perform the following steps:

1. Start Visual Studio.
2. If you receive a "`Select Development Language`" dialog, select `Microsoft C++`.
3. Select "`New Project`" from the Start Page, or alternatively from the `File` menu.

4. In the “New Project” dialog, ensure that:
   (a) Visual C++ > Win32 is selected in the Installed Templates pane.
   (b) Win32 Console Application - Visual C++ is selected in the Application Type pane.
   (c) .NET Framework 2.0 is selected in the .NET Framework dropdown.
   (d) Enter the name of the challenge in the Name: field.
   (e) Note the location in which the project is being created.
   (f) Click on OK to build the project. (See Figure 2.8)
5. The Win32 Application Wizard will run. Select Next on the Wizard Welcome screen.
6. On the Applications Settings dialog, enusre that “Console Application” is checked, and “Precompiled Headers” is unchecked. (See Figure 2.9). Click on Finish to build the project.
7. The new <Application>.cpp file will be opened and displayed in the File Pane. (See Figure 2.10)
8. The following changes are recommended to the main source file to ensure maximum platform compatibility:
   (a) Add “using namespace std;” before any included headers.
   (b) Remove or comment out the “#include "stdafx.h"” line
   (c) Add “#include <stdio.h>” and “#include <iostream>” to ensure POSIX compatibility.
   (d) Change int _tmain(int argc, _TCHAR* argv[])“” to “int main()”
9. You are now ready to develop your submissions.

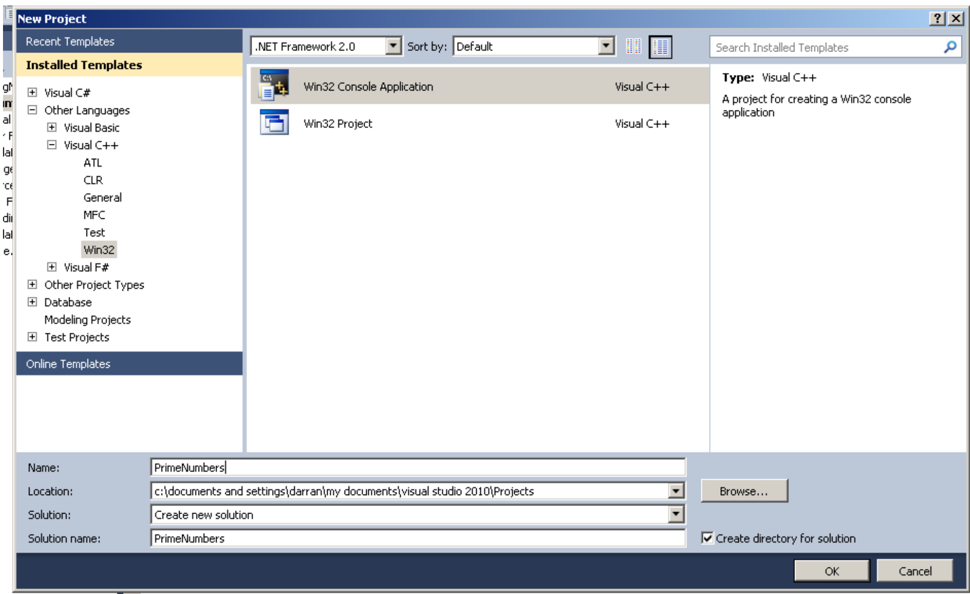

Figure 2.8: Visual Studio C++ New Project

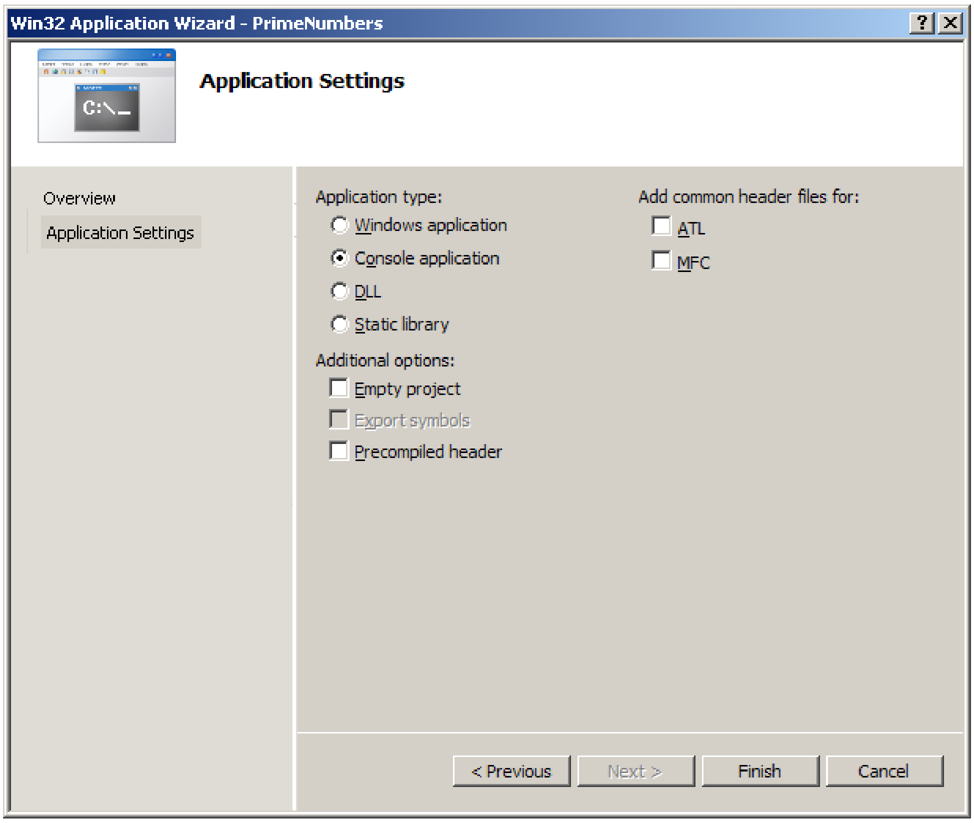

Figure 2.9: Visual Studio C++ Project Options

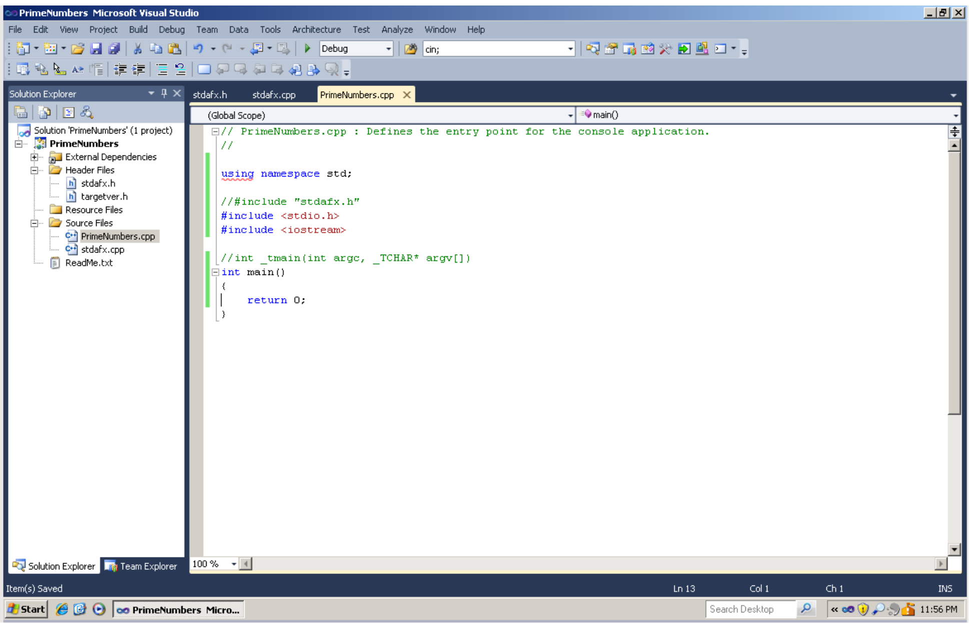

Figure 2.10: Visual Studio C++ Program.cpp

When testing your submission with PC^2 you will be required to either:

1. Copy your source code file to the same location as sample input files, or
2. Copy the sample input files into the same location as the source code file.

2.4 Submission Guidelines

The primary requirement of any submission, is that all source code required for the submission is located in a single text file.

To test your submission, the judges machine will compile your source code to an exectuable or class file in the case of Java, then execute it. All input for the application is feed in via `stdin` (or using standard console input), and all application responses and feedback should be returned via `stdout` (or using standard console output). In effect the judges machine will run:

```
$          submission.exe < challenge1_input.txt > challenge1_output.txt
```

Any output the application produces is saved to a file, and this file is then compared to a known correct answer file. Any variations from between the applications output and the answer file will result in a **`No`** response. If both the output of the application and answer file match, then the judges machine will return a **`Yes`** response.

3 Performance

3.1 BigO Notation

Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the **worst-case** scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.

Typically, algorithms will be assigned the following functions:

O(1) This algorithm operates in the same time irrespective of the number of elements to be processed. (The ideal algorithm).

O($\log_2 n$) This algorithm will have a worst-case runtime of $\log_2 n$ with n elements to be processed.

O(n) This algorithm will have a worst case runtime in the order of the number of elements to be processed.

O($n\log_2 n$) This algorithm will have a worst case runtime in the order of $n\log_2 n$ with n elements to be processed.

O(n^2) This algoritm will have a worst case runtime in the order of n^2 with n elements to be processed.

O(n^3) This algorithm will have a worst case runtime in the order of n^3 with n elements to be processed.

O(2^n) This algorithm will have a worst case runtime in the order of 2^n with n elements to be processed.

As an example, suppose the each operation can be done in 1 microsecond, and we have 256 elements to be processed. The respective worst case runtimes for each function is shown in Table 1.

Function	Time
$\log_2 n$	8 microseconds
n	256 microseconds
$n\log_2 n$	2 milliseconds
n^2	65 milliseconds
n^3	17 seconds
2^n	3.7×10^{64} centuries

Table 1: Big O runtimes

The information within this section is a very brief introduction to BigO notation, is included to help you choose between two different algorithms to complete a task based on the Order function given in the algorithm description. Much research has been completed on algorithm analysis, with many text books and reference books being authored on this one subject of Computer Science.

3.2 Measuring Performance

While information described with algorithms can be useful to gain an understanding of the algorithm complexity, it can be helpful to be able to measure the actual exectution time needed to complete sections of code. All modern languages or software libraries contatins functions to determine exectution times[2].

3.2.1 C#

The .NET Framework provides a Stopwatch class that is capable of being used to measure execution times. Figure 3.1 shows an example of the syntax.

[2] Modern IDEs such as Oracle Solaris Studio 12.3 include profiling tools to determine 'hot-spots' within applications and also be able to automatically record execution times of individual functions/methods to later analysis.

```
Stopwatch st = new Stopwatch();
st.Start();
// code to be timed goes here
st.Stop();
// time in milliseconds
long elapsed = (long) st.ElapsedTicks * 1000000 / Stopwatch.Frequency;
Console.WriteLine("time␣=␣{0}", elapsed);
```

Figure 3.1: Timing - C#

3.2.2 C++

The C standard library includes time specific functions in `<time.h>` on most systems. The primary function is the clock() function that returns the number of 'clicks' since the application started execution. A macro `CLOCKS_PER_SEC` is used to determine the ratio between clicks and seconds. Figure 3.2 shows an example of the syntax.

```
#include <time.h>
#include <stdio.h>
clock_t start = clock();
// code to be timed goes here
clock_t end = clock();
// time in clicks
long elapsed = (long)(end - start)/CLOCKS_PER_SEC;
printf("time␣=␣%d\n", elapsed);
```

Figure 3.2: Timing - C++

3.2.3 Java

The Java System library provides multiple timers with varying accuracy. Since Java 5, nanosecond timers has been available via the `System.nanoTime()` method[3]. Figure 3.3 shows an example of the syntax.

```
long start = System.nanoTime();
// code to be timed goes here
long end = System.nanoTime();
// time in nanoseconds
long elapsed = (end - start)/1000000;
System.out.printf("time␣=␣%d%n", elapsed);
```

Figure 3.3: Timing - Java

3.2.4 Integers vs Floating Point

Modern CPUs have integrated high-performance floating-point execution units, however it should be noted that the choice of Integer, Long's and Floating Point number will have an impact of performance of your solution. The code snippet in Figure 3.4 demonstrates the performance differences of using 'int', 'long' and 'double' for a simple add, multiple and divide sequence.

[3] Accuracy of the System.nanoTime() method is reliant on the JVM version and underlying Operating System. However most modern operating systems do provide some form on nanosecond timer.

,

```
public class NumberTypeTesting {
  static final long ITERATIONS = 1000000000;
  public static void main(String[] args) {
    final double DOUBLE_PRIME = 73;
    final int INT_PRIME = 73;
    final long LONG_PRIME = 73;
    long start;
    long end;
    long count;
    double valueDoubleA = 1.00;
    double valueDoubleB = Math.PI; // pi = 3.142
    double valueDoubleC = Math.E; // e = 2.718
    int valueIntA = 1;
    int valueIntB = 10;
    int valueIntC = 31;
    long valueLongA = 1;
    long valueLongB = 31;
    long valueLongC = 33;

    // Integers
    count = ITERATIONS;
    start = System.nanoTime();
    while (count-- != 0) {
      valueIntA += valueIntC * valueIntB / INT_PRIME;
    }
    end = System.nanoTime();
    System.out.printf("Integer␣time␣=␣%d␣msec%n", (end - start) / 1000000);

    // Long
    count = ITERATIONS;
    start = System.nanoTime();
    while (count-- != 0) {
      valueLongA += valueLongC * valueLongB / LONG_PRIME;
    }
    end = System.nanoTime();
    System.out.printf("Long␣time␣=␣%d␣msec%n", (end - start) / 1000000);

    // Double
    count = ITERATIONS;
    start = System.nanoTime();
    while (count-- != 0) {
      valueDoubleA += valueDoubleC * valueDoubleB / DOUBLE_PRIME;
    }
    end = System.nanoTime();
    System.out.printf("Double␣time␣=␣%d␣msec%n", (end - start) / 1000000);
  }
}
```

Figure 3.4: Source code for timing test

The results[4] for the above test in Figure 3.4 are:

```
Integer time = 1131 msec
Long time = 2673 msec
Double time = 1298 msec
```

While there is a minor performance drop for using floating point numbers, using 64bit longs yields over double the execution time. This may easily be fixed by utilising an environment that runs as 64bit code, but this is not guaranteed to be available during the competition.

[4]Java 6SE 32bit was used to generate the following results.

3.3 Implementation and Modern Software Engineering Practices

One of the aims of the competition is to develop efficient solutions to the challenges being presented. However often this also means not following modern software engineering practices and taking as many shortcuts as possible.

Some items that are typically seen (and encouraged) are:

1. Liberal use of global variables utlised by direct access.
2. Libreal use of function pointers and jump tables in C++.
3. Dispite strong OOP principles with each programming language, these are often ignored for more simple data structures and items like inheritence and encapsulation are ignored.
4. Nested classes liberally use public variables allowing for direct access.
5. The "`goto`" statement being used in C++ and C# submissions[5].
6. Ignoring typical design patterns, unless they provide direct and significant benefit in utlising an algorithm to complete a challenge.
7. Error checking is kept to a minimum, mainly designed around corner cases for algorithms to handle rather than handling bad and malformed input.

As mentioned in Section 2.4, that all source code required for the submission is located in a single text file, also requires some creative uses of both local and anonymous classes.

[5] While most professional programmers avoid "goto" as it's considered inherently evil, there are some instances where its use can save execution time and/or reduce code complexity.

4 Basic Source Templates

All source code submissions are to consist of a single source code file, as previously mentioned. This section aims to provide simple templates that can be utilised to create your submissions. It will also cover some of the basic console functions available with each langauge.

4.1 Input / Output

4.1.1 C#

The .NET Framework unfortunately has rather cumbersome support for handling console input and output. The `System.Console`[6] class provides methods for dealing with the console. The three main methods that are typically used are:

1. `Console.ReadLine();`
2. `Console.Write();`
3. `Console.WriteLine();`

4.1.1.1 Input The primary function for input from the Console is the `Console.ReadLine()` method which as the name indicates, reads a single line from the console and returns a `string`.

In order to extract information from the string, it is needed to split the string based on a delimiter (typically a space), then attempt to convert each part into the desired type. Figure 4.1 shows how to read a group of 3 integers (per line) from the console, until a three 0's (zeroes) are entered. The numbers for each line is added, and the sum is written back to the console.

4.1.1.2 Output The primary functions for output to the Console are the `Console.Write()` and `Console.WriteLine()` methods. These two differ only by the latter terminating the line with a `carriage-return`, while the former does not.

One item to note, that a single `Console.Write()` method may only take up to 5 parameters, the first being a string, and the other 4 being items to be inserted into the string. Item placement within the string is denoted by a number with `{}` brackets. (See the last line in Figure 4.1 for an example). The item placement parameter, may also take a second argument, being the type to have the item converted to, or displayed as. Common types include:

```
(C) Currency:  . . . . . . . . .  {0:C}
(D) Decimal:.  . . . . . . . . .  {0:D}
(E) Scientific:.  . . . . . . . . {0:E}
(F) Fixed point:  . . . . . . . . {0:F}
(G) General:.  . . . . . . . . .  {0:G}
(P) Percent:.  . . . . . . . . .  {0:P}
(X) Hexadecimal:  . . . . . . . . {0:X}
```

By default, console output is buffered, and only written periodically as determined by underlying system settings. To flush the output to console immediately, the `Console.Out.Flush()` method can be utilised.

[6]http://msdn.microsoft.com/en-us/library/system.console.aspx

,

```
using System;
namespace AddNumbers {
  class Program {
    static void Main(string[] args) {

      // Define our numbers to read.
      int[] numbers;
      int index;
      int sum;
      string line;
      string[] linesplit;

      // Keep reading the input from the console until we have nothing left.
      while ((line = Console.ReadLine()) != null) {

        // Test for exit condition.
        if (line.CompareTo("0␣0␣0") == 0) {
          break;
        }

        // Reset array indices and sum of numbers
        index = 0;
        sum = 0;

        // Split the line read, and create a new int array to hold our value.
        linesplit = line.Split(' ');
        numbers = new int[linesplit.Length];

        // Attempt to convert the individual parts to int's
        foreach (string element in linesplit) {
          try {
            numbers[index] = Convert.ToInt32(element);
          }
          catch {
            numbers[index] = 0;
          }
          index++;
        }

        // Sum our number and output the sum to Console.
        foreach (int number in numbers) {
          sum += number;
        }
        Console.WriteLine("{0}", sum);
      }
    }
  }
}
```

Figure 4.1: C# Input Template

4.1.2 C++

Due to the environment in which C++ was originally developed, C++ has very strong capabilities for handling both console input and output. C++ offers two methods when working with the console:

1. iostreams
2. C standard library functions.

While the two methods can be intermixed, it is recommended that programmers utilise a single method for their application[7]. For the purposes of this guide, I'll only explain the `iostreams` method as it is often seen as easy to use of the two methods.

4.1.2.1 Input Input from the console is handled by the `std::cin` stream, and has the ability to take multiple types of inputs in a single line or function call. (This is possible due to operator overloading in C++). Figure 4.2 shows the same application written in C++, as shown in Figure 4.1.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main() {
    int a, b, c;

    do {
        // Get input of 3 integers and store in a, b and c.
        cin >> a >> b >> c;

        // Test exit condition, and exit if true.
        if(a == 0 && b == 0 && c == 0)
            break;

        // Output the sum.
        cout << (a + b + c) << endl;
    } while(true);
   return 0;
}
```

Figure 4.2: C++ input and output example

Since the `cin` and `cout` streams operate on single variables, an alternate method is required to read a complete line in one function call. This method is `getline ( istream& is, string& str );`, where `is` is the character stream, and `str` is the string to place the input into. As example of `getline()` is in Figure 4.3.

```
#include <iostream>
#include <string>
using namespace std;

int main () {
  string str;
  cout << "Please␣enter␣full␣name:␣";
  getline (cin,str);
  cout << "Thank␣you,␣" << str << ".\n";
  return 0;
}
```

Figure 4.3: C++ getline() example

Once the line has been fetched with the `getline()` method, you are free to use any of the other string functions to extract information from the string.

[7]Mixing the two methods is possible, on provision that all input and output streams are emtpy when switching between either method. This is due to the buffering that each method utilises during Console IO operations.

4.1.2.2 Output The primary method of output is via the `cout` iostream, as shown in Figure 4.2. The format is simply:

```
cout << {object} << {object} << {object} << " string " << ..  << endl;
```

where each {object} represents any C++ primitive type or any C++ object.

There are two methods to generating a newline character, utilising C++. You may either:

1. Output string `"\n"`, or
2. Output `std::endl` .

The difference between the two methods, is that "`std::endl`" will flush the output buffer to console, where "`\n`" will not.

To control the precision of floating point numbers, you can use the `setprecision(`x`)` method as part of the output sequence. eg: `cout << setprecision(4) << (double)1.23456788 << endl;` will output `1.235` to the console.

4.1.2.3 iostreams vs printf The other method to perform console output is the C function `printf()`. `printf()` offers the same features as the `cout` iostream, and may be used when very fine control over output is required especially with floating point numbers.

The general format of the printf() function is:

```
printf(const char *str, ...);
```

`str` is a formatted string, that may contain 0 or more place holders for additional arguments. Placeholders in the formatted string are simply filled in order of additional arguments as specified in the function call, and the additional arguments must be of the same type as specified by the placeholder.

Formats for placeholders include:

`%d` - decimal
`%du` - decimal unsigned
`%f` - floating point
`%s` - string (char*)
`%c` - character
`%x` - hexidecimal number
`%l` - long
`%lu` - long unsigned

Additional fields may be added to the place holders to specify field width and/or precisions. For example:

`%.5f` - will display a floating point number to 5 decimal places.
`%5s` - will consume at exactly 5 character spaces for a string.

There are a few special reserved characters for the printf(), some of these include:

`\n` - carriage return.
`\t` - tab character.

There are has been some debate over the performance aspects of the two methods, often siting that there is no performance difference. Utilising the source code in Figure 4.4, I've found that in some cases there can be significant differences in performance, with `printf()` being up to 250 times faster than the equivalent `cout` function.

Based on average times for 5 runs of the test application in Figure 4.4, yeilds the following results:

1. cout time = 67240 msec
2. printf() time = 280 msec

```
#include <cstdlib>
#include <iostream>

using namespace std;

#define LOOPCOUNT 1000000
#define NUMBER1 31.0
#define NUMBER2 21

int main() {
    long count = LOOPCOUNT;
    double dbl = NUMBER1;
    int Int = NUMBER2;
    char* str = (char*) &"hello";

    clock_t start = clock(); // Start timing
    while (count--) {
        cout << dbl << "␣" << Int++ << "␣" << str << "\n";
        dbl = dbl * (double) Int / ((double) Int * 2.0);
    }
    clock_t end = clock(); // End timing
    cerr << "cout␣time␣=␣" << double(end - start)*1000.0/CLOCKS_PER_SEC
         << "msec␣" << endl;

    // reset start values
    count = LOOPCOUNT;
    dbl = NUMBER1;
    Int = NUMBER2;
    start = clock(); // Start timing
    while (count--) {
        printf("%f␣%d␣%s\n", dbl, Int++, str);
        dbl = dbl * (double) Int / ((double) Int * 2.0);
    }
    end = clock(); // End timing
    cerr << "printf␣time␣=␣" << double(end - start)*1000.0/CLOCKS_PER_SEC
         << "msec" << endl;

    return 0;
}
```

Figure 4.4: C++, `cout` vs `printf()`

4.1.3 Java

Similar to C++, Java has a very capable set of support functions for handling console input and output. These are mainly archived through the `System.in` and `System.out` classes used in conjuction with the `Java.util.Scanner` class provided with the default Java libraries.

4.1.3.1 Input Java historically has had a large number of different methods for handling console input, with each new version of Java providing a more streamlined method of handling these functions.

The current prefered method for console input in Java is to use the `Java.util.Scanner` class tied with the `System.in` object to extract the required information from the console. An example of the `Scanner` class can be found in Figure 4.5 which solves the same problem as shown in Figure 4.1.

4.1.3.2 Output Output is easiest handled via the `System.out.printf()` method, as it offers a good match between flexiblity and performance. The format for the method call is the same as the C++ `printf()` function as described in Section 4.1.2.3. An example of the method call is also in Figure 4.5.

```
import java.util.Scanner;
public class AddNumbers {
   public static void main(String[] args) {

      Scanner in = new Scanner(System.in);

      // get first line and check for end of test cases
      String line = in.nextLine();

      // Continue until exit condition
      while (!line.equals("0␣0␣0")) {

         // extract three ints
         Scanner sc = new Scanner(line);
         int a = sc.nextInt();
         int b = sc.nextInt();
         int c = sc.nextInt();

         System.out.printf("%d%n", (a + b + c));

         // get next line
         line = in.nextLine();
      }
   }
}
```

Figure 4.5: Java Input and Output

5 Basic Algorithms

5.1 Sorting

The majority of data structures that will be utilised within the competition all provide some form of inbuilt sorting algorithm, or through their design are naturally sorted as in the case of a Binary Search Tree.

It is highly recommended as far as competition submissions are concerned, that you utilise the built-in sort methods rather that attempting to implement your own sort method.

Typically, most data structures will utilise either quicksort or merge sort (depending on the underlying structure) as they both offer **O(n log n)** performance in the average case.

5.2 Searching

When given a linear array of data items, search algorithms find information about a particular data item in the list or find the location of the data item in the list. Two primary search algorithms are:

1. Linear Search
2. Binary Search

5.2.1 Linear Search

Linear Search algorithms transverse through a list of data items in sequential order attempting to find the location of the data item. The list itself may or may not be sorted, and the underlying data structure may be a linear array or a linked list.

Algorithm 1 Linear Search

Input Vector S, with n elements, with search key k
Output if $k \in S$ return index of k, else return -1

```
procedure LinearSearch(S, k)
    for i = 0 to n − 1 do
        if S[i] = k then
            return i
        end if
    end for
    return -1
end procedure
```

5.2.1.1 Description of working The linear search algorithm takes a vector (aka array) of elements, and simply searches all elements in order as stored. This can yeild slow performance with large vectors, as the worst case for linear search is **O(n)**.

5.2.1.2 Implementation Figure 5.1 and Figure 5.2 show the linear search algorithm as implemented in Java and C++ respectively.

```
public int LinearSearch(E[] vector, E key){
  for(int index = 0; index < vector.length; index++){
    if(vector[index] == key){
      return index;
    }
  }
  return -1;
}
```

Figure 5.1: Linear Search Implementation (Java)

```
public int LinearSearch(E[] vector, int vsize, E key){
  for(int index = 0; index < vsize; index++){
    if(vector[index] == key){
      return index;
    }
  }
  return -1;
}
```

Figure 5.2: Linear Search Implementation (C++)

5.2.1.3 Sample Problem - Linear Search Given a list of numbers (integers) in a line, determine if the first value on the line is present within the subsequent list of numbers.

INPUT

Input consists of one or more lines, with the first line being the number of cases to test.

Each line consists of one or more integers in the range of 0 to 32767. The first integer is the key value, followed by a list of up to 32 integers forming a vector of numbers.

SAMPLE INPUT

```
3
10 12 327 0 10
1 2 3 4 5 6 7 8
10 20 30 40 50 60 70 90 10
```

OUTPUT

The output of each line should consist of a single integer being either the index of the key within the vector, being zero (0) offset, or the value -1 if the key is not present in the vector.

SAMPLE OUTPUT

```
3
-1
7
```

5.2.1.4 Problem Solution A solution to the above problem utilising a linear search can be seen in Figure 5.3. The solution included simply reads in a line of numbers, and attempts to find the first value in the list of other values in the line. It continues to do this, until the exit condition is reached.

Some of the test cases to handle include:

1. The case count being less or equal to 0.
2. The line itself contains a single integer, being the key, but provided with an empty vector to search.

,
```java
import java.util.Scanner;

public class LinearSearch {

  /**
   * Perform linear search of array (vector) for item (key).
   *
   * @param vector array of numbers
   * @param key item to look for in array
   * @return index of key in vector, or -1 is not present
   */
  public static int LinearSearch(int[] vector, int key) {
    for (int index = 0; index < vector.length; index++) {
      if (vector[index] == key) {
        return index;
      }
    }
    return -1;
  }

  /**
   * Main
   */
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);

    // get first line and get the number of cases to test.
    int caseCount = Integer.parseInt(in.nextLine());

    // Keep reading each line while caseCount > 0
    while (caseCount-- > 0) {

      // split by white space. so we have an array of numbers
      String[] numStrs = in.nextLine().split("\\s+");

      // create an array to hold our numbers, and convet the array of strings to
      // numbers. Note: numStrs[0] is the key value
      int[] nums = new int[numStrs.length - 1];
      for (int i = 1; i < nums.length; i++) {
        nums[i - 1] = Integer.parseInt(numStrs[i]);
      }

      // Output the index of the key in the vector
      System.out.printf("%d%n", LinearSearch(nums, Integer.parseInt(numStrs[0])));
    }
  }
}
```

Figure 5.3: Solution to Linear Search Problem (Java)

5.2.2 Binary Search

The binary search algorithm is a more efficient method of searching a vector, on provision that the vector is sorted and any element can be accessed in **O(1)** time. Becuase of these two conditions, it can't with some storage data structures like linked lists (as elements can't be accessed in **O(1)** time), nor is suitable for vectors that are unsorted.

Algorithm 2 Binary Search

Input An ordered vector S, with n elements, with search key k. Items low and $high$ indicate current search space of vector S
Output if $k \in S$ return index of k, else return -1

```
procedure BinarySearch(S, k, low, high)
    if low > high then
        return -1
    else
        mid ← ⌊(low + high)/2⌋
        if k = S[mid] then
            return mid
        else if k < S[mid] then
            return BinarySearch(S, k, low, mid − 1)
        else
            return BinarySearch(S, k, mid + 1, high)
        end if
    end if
end procedure
```

5.2.2.1 Description of working The binary search algorithm is a naturally recursive algorithm, in that it calls itself to continue searching the vector.

The algorithm starts with the entire space of the vector, and looks at the mid point between the *low* and *high* values. If this value is not the required key (k), it will then determine if the key is less than or greater than the current value at *mid*. If the key is lower than mid, then it will redefine the search space to be that between *low* and *mid-1*, otherwise redefine the search space between *mid*+1 and *high*. It then calls itself to perform another search. In the event that *low* is greater than *high*, it determines that the key is not in *S*, and will return -1.

What the algorithm effectively does is split the entire search space of vector *S* into 2 parts, if the key is not at *mid*. By virtue, if the value at *mid* is less than the key, it understands that there is no justification to look at values located to the left of the current *mid* point in the vector. With each iteration of the search it effectively reduces the search space by half.

By reduction of the search by half, the worst case performance of a binary search is **O(log *n*)**. As this is a vast improvement on a linear search, a binary search should be utilised when ever possible. However, this requires that the vector be sorted before a binary search can be performed.

Utilising a quicksort or merge sort, will add overhead (both of these typically yield **O(*n* log *n*)** performance), so for very large vectors, the overhead of a sort prior to search is not that great, but for small size vectors, the overhead of a sort may not yield greater performance over the simple (and slow) linear search.

5.2.2.2 Implementation Figure 5.4 shows an implementation of the binary search in Java.

5.2.2.3 Sample Problem For demonstration of the Binary Search algorithm, I will use the same problem as shown in Section 5.2.1.3, Linear Search Problem. However, it is expected that the input of numbers (except for the key) will be in order from lowest to highest.

5.2.2.4 Problem Solution A solution to the sample problem utilising a binary search can be seen in Figure 5.5. The solution included simply reads in a line of numbers, and attempts to find the first value in the list of other values in the line. It continues to do this, until the exit condition is reached.

Some of the test cases to handle include:

1. The case count being less or equal to 0.
2. The line itself contains a single integer, being the key, but provided with an empty vector to search.

,

```
public int BinarySearch(E[] vector, E key, int low, int high){
  if(low > high){
    return -1;
  }
  int mid = (low+high)/2;
  if(vector[mid] == key){
    return mid;
  } else {
    if(key < vector[mid]){
      return BinarySearch(vector, key, low, (mid-1));
    } else {
      return BinarySearch(vector, key, (mid+1), high);
    }
  }
}
```

Figure 5.4: Binary Search (Java)

5.3 Array Handling

When talking about arrays, we typical define one as a single string[8] or allocation of elements in a linear continous region. Ee can define an array of arrays to form a two dimensional array, or an array of arrays of arrays to form a three dimensional array, and continue to do so, allowing for infinite dimensional array. This section will typically discuss array operations on two dimensional arrays, such as rotation and mirroring that may support application of algorithms or may simply speed up implementions due to underlying hardware constraints.

5.3.1 Array Performance

When most people are taught programming in either High School or early University level, performance constraints in regards to arrays is either neglected or very limited discussion is made without concrete examples. This section aims to give some insight to performance issues when dealing with arrays, primarily around performance bottlenecks.

The primary reason for poor performance when using arrays, is not based on a programming language or library issue, but is based on lack of understanding the underlying hardware and how memory access works.

From a hardware achitectural viewpoint there are different classes of hardware memory:

1. Primary - The RAM that the CPU sees as the address space given to it.
2. Secondary - The Harddrive installed within the system, providing non-voliatile memory.
3. Teritiary - Removable non-voliatile memory such as DVD's, CD's, USB Flash Keys, etc.

What is typically not taught are the different levels of primary storage. Utilising a modern Intel x86 processor[9] as an example, the primary levels include:

1. CPU registers, internal to the CPU and these are where typically most operations are performed. These typically have a zero latency access.
2. Level 1 Data and Code caches, these hold the most recent code and data being accessed from the Level 2 cache. There are typically two Level 1 caches, each desginated for holding either code or

[8]A string in many languages (notably low level langauges like assembler) does not define a String of letters, numbers and punctuation, but rather a linear memory region of bytes.

[9]The size and access latencies described are taken from the Intel Core 2 Quad Family Datasheet and Vol 3 of the Intel Architecture Manuals.

,

```
import java.util.Scanner;

public class LinearSearch {

  /**
   * Perform binary search of array (vector) for item (key).
   *
   * @param vector array of numbers
   * @param key item to look for in array
   * @param low start position of array to search
   * @param high end position of array to search
   * @return index of key in vector, or -1 is not present
   */
  public static int BinarySearch(int[] vector, int key, int low, int high){
    if(low > high){
      return -1;
    }
    int mid = (low+high)/2;
    if(vector[mid] == key){
      return mid;
    } else {
      if(key < vector[mid]){
        return BinarySearch(vector, key, low, (mid-1));
      } else {
        return BinarySearch(vector, key, (mid+1), high);
      }
    }
  }

  /**
   * Main
   */
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);

    // get first line and get the number of cases to test.
    int caseCount = Integer.parseInt(in.nextLine());

    // Keep reading each line while caseCount > 0
    while (caseCount-- > 0) {

      // split by white space. so we have an array of numbers
      String[] numStrs = in.nextLine().split("\\s+");

      // create an array to hold our numbers, and convet the array of strings to
      // numbers. Note: numStrs[0] is the key value
      int[] nums = new int[numStrs.length - 1];
      for (int i = 1; i < nums.length; i++) {
        nums[i - 1] = Integer.parseInt(numStrs[i]);
      }

      // Output the index of the key in the vector
      System.out.printf("%d%n",
        BinarySearch(nums, Integer.parseInt(numStrs[0]), 0, nums.length-1));
    }
  }
}
```

Figure 5.5: Solution to Binary Search Problem (Java)

data, and are typically 16kilobytes size in size[10]. The Level 1 cache will typically have a 1-2 CPU cycle access latency, and besides the CPU registers is the fastest memory available to the CPU.

3. Level 2 unified Code and Data cache will typically be in the size of anywhere from 256kilobytes up to 16megabytes in size depending on the CPU make and model. This acts a large cache between the main memory of the system, and the CPU and Level 1 caches. The Level 2 cache will typically have a 5-10 CPU cycle access latency, primarily due to restrictions of the size of the cache. (The larger the cache, the slower the access due to it's size[11]).

4. Level 3 unified Code and Data caches are present in some CPUs and act as a third level between the CPU and main memory of the system. Level 3 caches are becoming more common with multi-core CPUs, as Level 2 and Level 1 caches are being tied to a particular CPU core, where the Level 3 cache can act as a unified cache for all CPU cores. While Level 3 caches can be quite big, in some cases now approaching 32MB in size[12], they are even slower that the Level 2 cache with access latency between 15-35 CPU cycles.

5. Main Memory, is typically in the form of the DIMMs that get installed in the mainboard of the computer. While systems are approaching very large capacities (32GB can be found in home desktop systems, and up to 194GB in workstations), they are very slow compared to the CPU registers and even the Level 1,2 and 3 caches. Typical access latencies can be measured anywhere from hundreds to thousands of CPU cycles. That is, if the CPU needs some information that is not present in one of the caches, it can potentially stall for 100's, if not 1000's of CPU cycles doing nothing while it waits for the information[13].

Why is knowing all about the different level of cache's important in regards to arrays? Simply, if you try to access an array element that is not in one of the caches, your application will suffer a performance hit whilst waiting for the information from main memory. By ensuring that your next memory access will be in one of the CPUs cache, you can ensure the best possible performance for your applicaton when dealing with any sized array.

To quote Terje Mathisen (a well known programming optimization guru[14]): "All programming is an exercise in caching."

The problem with current programming languages, notably Java and C# is that they run on top of virtual machines or utilise some form of JIT compilation, negating any direct control of the CPU and cache management functions. Even C++ applications lack cache management functions (unless you utilise inline assembler in your application). The way to work with these languages is to exploit the nature of the CPU's cache management engine to your advantage.

The CPUs cache management engine works by loading the contents of most recently accessed memory address into the cache in either 32byte chunks for the level 1 cache, or 4kilobyte chunks for the level 2 and 3 caches. To ensure that the next memory address is located in the cache, ensure that the next array element to access is located very close of the last one accessed. When the CPU cache management engine sees your last access was on a border of a chunk it will load in the next chunk in a linear fashion based on the last accessed memory address.

Therefore the to gain the best possible performance when dealing with arrays, either utilise very small arrays that will fit into the level 1 cache, or only access arrays in a linear fashion row by row.

To illustrate these cache performance aspects, the application in Figure 5.6 yeilds the results show in Table 2. As can be seen, when dealing with a 16384×16384 sized array (consuming 256MB), accessing the array row by row takes 1.35 seconds, however accessing it column by column takes just over 16 seconds. This clearly demonstrates the caches hits/misses taking place and confirms the latencies expected by the cache misses.

[10] Each CPU make/model can have different L1 cache sizes, for example the Intel E7 Xeon CPUs have 32KB for code and 32KB for data in it's L1 cache, and the AMD Opteron utilises 64KB L1 caches.

[11] This is a very crude approximation, as there are many factors that determine the performance of the L2 cache.

[12] The Sun UltraSPARC IV+ utilises a 32MB L3 cache and the Intel Itanium 9300 utilises a 24MB L3 cache.

[13] CPU vendors do a lot to avoid this, and even resort to techniques such as SMT (aka HyperThreading), or even offer CPU instructions that allow applications to preload the caches with information to avoid these stalls.

[14] Terje, at one time worked for iD Software on the original Doom and Quake games and was able to get Quake running a full 3D environment utilising a software based graphics renderer on hardware such as the Intel Pentium 60. (The Intel Pentium 60, ran at 60MHz, roughly 60-90 times slower than current CPUs).

,

```
using namespace std;
#include <time.h>
#include <stdio.h>
#include <iostream>
char array1[1024][1204];
char array2[2048][2048];
char array3[4096][4096];
char array4[8192][8192];
char array5[16384][16384];

int main() {

  // Row by row.
  clock_t start = clock();
  for (int y = 0; y < 1024; y++) {
    for (int x = 0; x < 1023; x++) {
      array1[y][x] = array1[y][x + 1];
    }
  }
  clock_t end = clock();
  long elapsed = (long) (end - start) / (CLOCKS_PER_SEC / 1000);
  printf("1MB␣row␣time␣=␣%dmsec\n", elapsed);

  // Column by Column
  start = clock();
  for (int x = 0; x < 1024; x++) {
    for (int y = 0; y < 1023; y++) {
      array1[y][x] = array1[y + 1][x];
    }
  }
  end = clock();
  elapsed = (long) (end - start) / (CLOCKS_PER_SEC / 1000);
  printf("1MB␣col␣time␣=␣%dmsec\n", elapsed);

  // .... <snip 4MB, 16MB and 64MB loops>

  // 256MB Row by row.
  start = clock();
  for (int y = 0; y < 16384; y++) {
    for (int x = 0; x < 16383; x++) {
      array5[y][x] = array5[y][x + 1];
    }
  }
  end = clock();
  elapsed = (long) (end - start) / (CLOCKS_PER_SEC / 1000);
  printf("256MB␣row␣time␣=␣%dmsec\n", elapsed);

  // Column by Column
  start = clock();
  for (int x = 0; x < 16384; x++) {
    for (int y = 0; y < 16383; y++) {
      array5[y][x] = array5[y + 1][x];
    }
  }
  end = clock();
  elapsed = (long) (end - start) / (CLOCKS_PER_SEC / 1000);
  printf("256MB␣col␣time␣=␣%dmsec\n", elapsed);
  return 0;
}
```

Figure 5.6: Array Access Performance (C++)

	Row x Row	Column x Column
1024 × 1024 (1MB)	~0 msec	10 msec
2048 × 2048 (4MB)	20 msec	220 msec
4096 × 4096 (16MB)	90 msec	830 msec
8192 × 8192 (64MB)	350 msec	4150 msec
16384 × 16384 (256MB)	1350 msec	16080 msec

Table 2: Array Access Performance

5.3.2 Array Traversal Methods

Any 3-dimension array, can be accessing in a variety of ways, including column-wise, row-wise and starting from the top to bottom, bottom to top, left to right and right to left.

As seen in Figure 5.6, row-wise and column-wise access methods where undertaken in a top to bottom, left to right fashion. Formalised algorithms for row-wise and column-wise access are shown in Algorithms 3 and 4 respectively.

To change from a left to right, to a right to left access pattern, simply count down from the width value to 0 for columns. To change from top to bottom, to a bottom to top access pattern, simply count down from the height value to 0 for rows.

The same applies to any 2+ dimensional array, to change the direction of travsel, either change from counting from 0 to width/height to counting down from the width/height to 0, and vice versa.

Algorithm 3 Row-wise Traversal of an Array

Input A source matrix S, with n by m elements.
Output Prints value in element of matrix

```
procedure MatrixRowWiseTraversal(S, n, m)
    for row = 0 to m − 1 do
        for column = 0 to n − 1 do
            print S[row][column]
        end for
    end for
end procedure
```

Algorithm 4 Column-wise Traversal of an Array

Input A source matrix S, with n by m elements.
Output Prints value in element of matrix

```
procedure MatrixColumnWiseTraversal(S, n, m)
    for column = 0 to n − 1 do
        for row = 0 to m − 1 do
            print S[row][column]
        end for
    end for
end procedure
```

5.3.3 Diagonal Traversal of an Array

Diagonal traversal of an array is used for many areas including image analysis, map scanning, simple path finding methods[15].

The one issue with this form of travesal through an array, are the underlying performance penalities that will occur due to cache misses, as each subsequent access to the array is non-linear. Like other array travesal techniques, diagonal traverse is an $\mathbf{O(n^2)}$ operation.

Algorithm 5 Diagonal Traversal of an Array

Input A source matrix S, with n by m elements.
Output Prints value in element of matrix

```
procedure MatrixDiagonalTraversal(S, n, m)
    x ← 0
    y ← 0
    while True do
        v ← x
        w ← y
        while v ≥ 0 ∧ w < m do
            print S[w][v]                                ▷ S[row][column]
            v ← v − 1
            w ← w + 1
        end while
        if x < n − 1 then
            x ← x + 1
        else if y < m − 1 then
            y ← y + 1
        else
            return
        end if
    end while
end procedure
```

5.3.3.1 Description of working As described in Algorithm 5, the algorithm utilises two loops, with the first loop (`while(true)`) determines the start position of the traverse, and the second while loop traverses the actual slice, starting from the top and moving downwards and to the left, (as denoted by the $v \leftarrow v - 1$ and $w \leftarrow w + 1$ operations).

The `if-else-if-else` statements recalculate the new start position for the next travesal, until the new start position exceeds the bounds of the array, in which case the algorithm exits.

The algorithm as described, starts in the top left corner, and moves towards the bottom right of the array. To modify the algorithm to scan from other origins to opposite corner, the line to be altered is the access function that prints the current element in the array. The start at the various origins and traverse to the opposite corner, the following forms are needed for the print function:

- Top-left origin to bottom-right traverse: print $S[w][v]$
- Bottom-left origin to top-right traverse: print $S[m - w - 1][v]$
- Top-right origin to bottom-left traverse: print $S[w][n - v - 1]$
- Bottom-right origin to top-left traverse: print $S[m - w - 1][n - v - 1]$

Figure 5.7 demonstrates the top-left origin to bottom-right traversel on a 3×4 array, with each slice (or starting location) shown.

[15]There are more comprehensive path finding techniques that utilise graphs/networks, so these methods are not commonly used.

The order of traversel for the other origins would result in the following sequences based on the forms listed above:

- Top-left origin to bottom-right traverse: `a, b, d, c, e, g, f, h, j, i, k, l`
- Bottom-left origin to top-right traverse: `j, k, g, l, h, d, i, e, a, f, b, c`
- Top-right origin to bottom-left traverse: `c, b, f, a, e, i, d, h, l, g, k, j`
- Bottom-right origin to top-left traverse: `l, k, i, j, h, f, g, e, c, d, b, a`

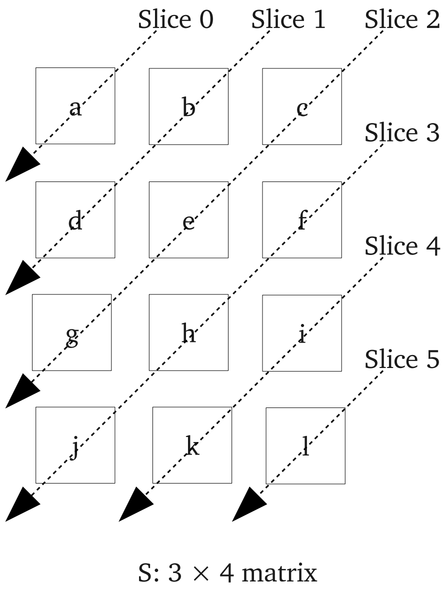

Figure 5.7: Diagonal Traversal of an Array with a Top-Left Origin

5.3.3.2 Implementation Figure 5.8 demonstrates the implementation of the algorithm as described.

,

```
public static void ArrayDiagonalTraverse(int[][] matrix, int width, int height){
  int start_x = 0;
  int start_y = 0;
  int column;
  int row;

  // Keep looping until exit condition.
  while (true) {

   // Initialise the starting location in the matrix for the current slice.
   column = start_x;
   row = start_y;

   // Traverse the current slice.
   while (column >= 0 && row < height) {
     System.out.printf("%d␣", matrix[row][column]);
     column--;
     row++;
   }

   // Update the start location for the next slice.
   if (start_x < width - 1) {
     start_x++;
   } else if (start_y < height - 1) {
     start_y++;
   } else {
     // Exit the method, as start locations are now out of matrix bounds.
     break;
   }
 }
}
```

Figure 5.8: Diagonal Traversal of an Array (Java)

5.3.4 Array Rotation

As shown in Section 5.3.1, the way that an array is accessed can effect the performance of your application. Methods to ensure that you can access an array row by row may require rotation of an array, or alternatively rotation of an array may be needed to utilise an algorithm or math function.

Any array rotation or mirroring function requires two copies of the array to be present at one time, the source array and target array. This must be considered in relation to the amount of memory required, as you effectively need double the memory requirement for either operation.

Both rotation and mirroring functions are **O(n²)** operations, due to all array members must be accessed to complete the opeation successfully.

Algorithm 6 Array Rotation Clockwise

Input A source matrix S, with n by m elements, with target matrix T, with m by n elements.
Output Matrix T represents matrix S rotated 90°clockwise.

procedure ROTATEMATRIXCLOCKWISE(S, T, n, m)
  $f \leftarrow m - 1$
  for $y = 0$ to $m - 1$ **do**
    for $x = 0$ to $n - 1$ **do**
      $T[x][(f - y)] \leftarrow S[y][x]$ ▷ S[row][column]
    end for
  end for
end procedure

Algorithm 7 Array Rotation AntiClockwise

Input A source matrix S, with n by m elements, with target matrix T, with m by n elements.
Output Matrix T represents matrix S rotated 90°anti-clockwise.

procedure RotateMatrixAntiClockwise(S, T, n, m)
  $f \leftarrow n - 1$
  for x = 0 to $n - 1$ **do**
    for y = 0 to $m - 1$ **do**
      $T[(f - x)][y] \leftarrow S[y][x]$ ▷ S[row][column]
    end for
  end for
end procedure

5.3.4.1 Description of working Algoritms 6 and 7 depict both clockwise and anti-clock array rotation respectively. Simply, both perform substitution of x and y values for the target array. An additionl variable is needed to hold an offset, so that the new array offsets may be calculated correctly[16]. Figure 5.9 shows the clockwise rotation operation in effect.

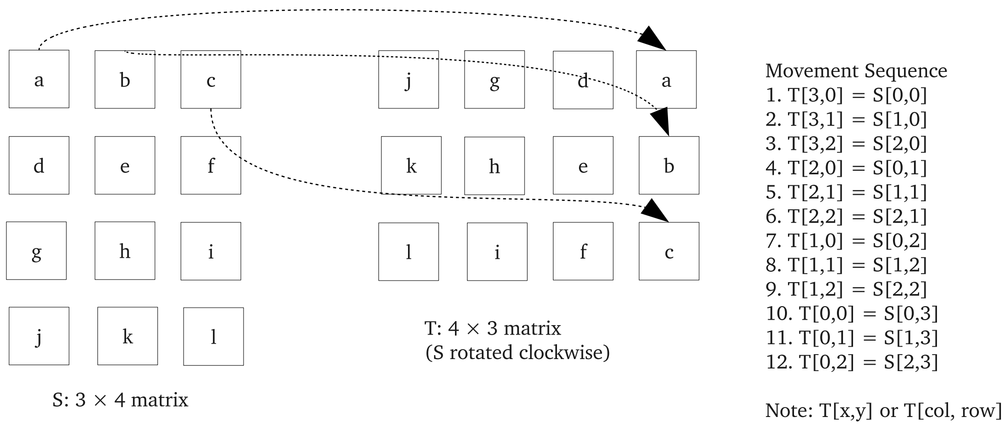

Figure 5.9: Array Rotation

5.3.4.2 Implementation Figure 5.10 and 5.11 show implementations of clockwise and anticlockwise rotation respectively.

```
public void ArrayRotateClockWise(E[][] source, E[][] target, int width, int height){
  int factor = height-1;
  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      target[x][(factor-y)] = source[y][x];
    }
  }
}
```

Figure 5.10: Array Clockwise Rotation (Java)

Performance of both of these implementations will suffer due to access not being performed in a row by row fashion, which limits the viability of the above algorithms. However they should still be considered and used, when the many operations are performed on the resulting array, rather than single use post rotation.

[16]While this is not necessary, it make for slightly cleaner code.

```
public void ArrayAntiRotateClockWise(E[][] source, E[][] T, int width, int height){
  int factor = width-1;
  for (int x = 0; x < width; x++) {
    for (int y = 0; y < height; y++) {
      target[(factor-x)][y] = source[y][x];
    }
  }
}
```

Figure 5.11: Array Anti-Clockwise Rotation (Java)

5.3.4.3 Sample Problem- Matrix Rotation Given a two dimension matrix measuring n $\times$ m, rotate the matrix in a clockwise direction.

INPUT

Input consists of one or more lines, with the first line being the size of the matrix to rotate in width and height. The width and height will be in a range between 0 and 79. The application should exit when the size given is "0 0".

The following lines denote the matrix to be rotated, with each element being a 1 or 0, separated by a space.

SAMPLE INPUT

```
3 3
1 0 1
0 0 0
1 0 0
2 2
1 0
0 1
0 0
```

OUTPUT

The output of each rotation should be the resultant matrix, without spaces between each column in the matrix. There must a blank line separating each resultant matrix.

SAMPLE OUTPUT

```
101
000
001

00
01
```

5.3.4.4 Sample Solution A solution to the sample problem utilising an array rotation can be seen in Figure 5.12. The solution performs the following:

1. Reads in the width and height values. If these are not 0 and 0, continue the main loop body, otherwise exit.
2. If either of the values is 0, (indicating a 0 width or height), skip attempting to read in a matrix, and go back to step 1.
3. Create a new array of size width, height to hold integers. Read in the values from stdin, and fill in these values into the respective position within the array.
4. Create a new array to hold the result of the rotation. Rotate the array, and print the results. Goto step 1.

As this challenge is very simple, the only issue to account for is if either the width or height is given as 0 (zero), in which case a blank line should be returned.

```
import java.util.Scanner;
public class ArrayRotation {

  /**
   * Rotate an Array in a clockwise direction
   *
   * @param source The source array
   * @param target The target array
   * @param width The width of the source array
   * @param height The height of the target array
   */
  public static void ArrayRotateClockWise(int[][] source, int[][] target, int width, int height) {
    int factor = height - 1;
    for (int y = 0; y < height; y++) {
      for (int x = 0; x < width; x++) {
        target[x][(factor - y)] = source[y][x];
      }
    }
  }

  /**
   * Main
   */
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);

    // get the size of the matrix to rotate.
    String matrixSizeLine = in.nextLine();
    Scanner matrixSize = new Scanner(matrixSizeLine);
    int width = matrixSize.nextInt();
    int height = matrixSize.nextInt();

    // Keep reading in a matrix until exit condition
   while (!((width == 0) && (height == 0))) {

      // Ignore the line if either width or height = 0.
      if (!((width == 0) || (height == 0))) {

        // Read in our matrix.
        int[][] matrix = new int[height][width];
        for (int count = 0; count < height; count++) {
          String[] matrixLine = in.nextLine().split("\\s+");
          for (int element = 0; element < matrixLine.length; element++) {
            matrix[count][element] = Integer.parseInt(matrixLine[element]);
          }
        }

        // Rotate the matrix and print.
        int[][] target = new int[width][height];
        ArrayRotateClockWise(matrix, target, width, height);

        // Print the resultant matrix.
        for (int row = 0; row < width; row++) {
          for (int column = 0; column < height; column++) {
            System.out.print(target[row][column]);
          }
          System.out.println();
        }

        // Print line between each matrix output
        System.out.println();
      }
      // get the size of the matrix to rotate.
      matrixSizeLine = in.nextLine();
      matrixSize = new Scanner(matrixSizeLine);
      width = matrixSize.nextInt();
      height = matrixSize.nextInt();
    }
  }
}
```

Figure 5.12: Solution to Array Rotation Problem (Java)

5.3.5 Array Mirroring or Flipping

Array mirroring or flipping is a very simple technique, that simply requires a offset to be calculated from the current height or width value. Typically, to calculate the mirror location, we simple subtract the current source column (or row) value from the width (or height) size to calculate the target location. Algorithms 8 and 9 depict mirroring along the vertical axis and horizontal axis respectively.

Algorithm 8 Mirror Array Along Vertical Axis

Input A source matrix S, with n by m elements, with target matrix T, with n by m elements.
Output Matrix T represents matrix S mirrored along a vertical axis

```
procedure MirrorMatrixVeritcal(S, T, n, m)
    for y = 0 to m − 1 do
        for x = 0 to n − 1 do
            T[m − y][x] ← S[y][x]                    ▷ S[row][column]
        end for
    end for
end procedure
```

Algorithm 9 Mirror Array Along Horizontal Axis

Input A source matrix S, with n by m elements, with target matrix T, with n by m elements.
Output Matrix T represents matrix S mirrored along a horizontal axis

```
procedure MirrorMatrixHorizontal(S, T, n, m)
    for y = 0 to m − 1 do
        for x = 0 to n − 1 do
            T[y][n − x] ← S[y][x]                    ▷ S[row][column]
        end for
    end for
end procedure
```

As both of these algorithms access both the source and target arrays row by row, there should be no (or few) performance penalities due to cache misses.

6 Advanced Algorithms

6.1 Simple Maths

6.1.1 Greatest common divisor

The Greatest Common Divisor or Eucilidean algorithm was originally developed by Euclid of Alexandria in 3^{rd} BC, and computes the greatest common divisor of two non-negative, not-both-zero integers `m` and `n`, that is the largest integer that divides both `m` and `n` *evenly*.

6.1.1.1 Description of working The Euclidean algorithm is based on the principle that the greatest common divisor of two numbers does not change if the smaller number is subtracted from the larger number. For if k, m, and n are integers, and k is a common factor of two integers A and B, then A=(n×k) and B=(m×k) implies A-B=(n-m)×k, therefore k is also a common factor of the difference. That k may also represent the greatest common divisor is proven below. For example, 21 is the GCD of 252 and 105 (252 = 12 × 21; 105 = 5 × 21); since 252 − 105 = (12 - 5) × 21 = 147, the GCD of 147 and 105 is also 21.

Since the larger of the two numbers is reduced, repeating this process gives successively smaller numbers until one of them is zero. When that occurs, the GCD is the remaining nonzero number. By reversing the steps in the Euclidean algorithm, the GCD can be expressed as a sum of the two original numbers each multiplied by a positive or negative integer, e.g., 21 = [5 × 105] + [(−2) × 252].

Iterative and recursive implementations of the algorithm are shown in Algorithms 10 and 11 respectively.

Algorithm 10 Euclidean Algorithm (Iterative)

Input Positives integers m and n which may share a common divisor
Output Greatest Common Divisor, being $n \geq 1$
  procedure EUCLID(m, n)
    while $n \neq 0$ **do** ▷ We have the answer if n is 0
      $r \leftarrow m \bmod n$
      $m \leftarrow n$
      $n \leftarrow r$
    end while
    return m ▷ The gcd is m
  end procedure

Algorithm 11 Euclidean Algorithm (Recursive)

  procedure EUCLID(m, n)
    if $n = 0$ **then**
      return m
    end if
    return gcd($n, m \bmod n$)
  end procedure

6.1.1.2 Implementation An implementation of the iterative form of the algorithm can be found in Figure 6.1.

```java
public static int gcd(int m, int n) {
  int r;
  while (n != 0) {
    r = m % n;
    m = n;
    n = r;
  }
  return m;
}
```

Figure 6.1: Euclidean Algorithm (Java)

6.1.1.3 Sample Problem - GCD Calculate the GCD of two integers.

INPUT

The first line of input will be N ($1 <= N <= 1000$), the number of test cases to run. On each of the next N lines will be two integers m and n ($0 <= m, n <= 2^{31} - 1$).

SAMPLE INPUT

```
3
12 60
60 24
3 5
```

OUTPUT

Output the greatest common denominator of m and n, one value on each line.

SAMPLE OUTPUT

```
12
12
1
```

6.1.1.4 Sample Solution The solution to this problem utilising Euclid's GCD algorithm can be seen in Figure 6.2.

Whilst this solution/challenge is rather simple, the following test cases must be considered:

1. One or both values are 0 (zero).
2. There will be longer run times if one or both numbers are primes. (The GCD is 1 in this case).
3. The input range is $0 < m < 2^{31} - 1$, which requires the use of at least a signed 32bit number (typically `int` on all platforms).

```java
import java.util.Scanner;
public class GCD {
  /**
   * Compute the Greatest Common Divisor
   *
   * @param m number 1, greater or equal to 0.
   * @param n number 2, greater or equal to 0.
   * @return the GCD
   */
  public static int EuclidGCD(int m, int n) {
    int r;
    while (n != 0) {
      r = m % n;
      m = n;
      n = r;
    }
    return m;
  }

  /**
   * Main
   */
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);

    // get first line and get the number of cases to test.
    int caseCount = Integer.parseInt(in.nextLine());

    // Keep reading each line while caseCount > 0
    while (caseCount-- > 0) {
      String line = in.nextLine();

      // extract numbers ints
      Scanner sc = new Scanner(line);
      int a = sc.nextInt();
      int b = sc.nextInt();

      // Output the gcd for the two given numbers.
      System.out.printf("%d%n", EuclidGCD(a,b));
    }
  }
}
```

Figure 6.2: Solution to GCD Problem (Java)

6.1.2 Sieve of Eratosthenes (prime number generation)

The brute force method of finding prime numbers to limit N, can be found by checking for divisibility of all numbers between 2 and N-1. There are however more efficient methods for finding prime numbers, including the use of Sieve's, in particulator the Sieve of Erathosthenes.

The Sieve of Erathosthenes, is a relatively simple algorithm, that performs prime number generation through iteration of composites based on the last prime found. This is done by maintaining an array of all numbers between 2 and N, and uses flags in each array slot to indicate if it's a prime, composite or unknown.

The Sieve of Erathosthenes is considered a **O(n log log n)** operation, with a **O(n)** memory requirement.

Algorithm 12 Sieve of Eratosthenes

Input Positives integer n denoting upper limit for prime number search, and vector S for list of prime numbers.
Output All prime number $\leq n$ stored in vector S.

  procedure ERATOSTHENES(n, S)
    define vector A of size n
    for $p \leftarrow 2$ to n **do**
      $A[p] \leftarrow p$
    end for
    for $p \leftarrow 2$ to $\lfloor\sqrt{n}\rfloor$ **do**
      if $A[p] \neq 0$ **then**
        $j \leftarrow p^2$
        while $j \leq n$ **do**
          $A[j] \leftarrow 0$
          $j \leftarrow j + p$
        end while
      end if
    end for

    $i \leftarrow 0$ ▷ Copy list of primes from A to S
    for $p \leftarrow 2$ to n **do**
      if $A[p] \neq 0$ **then**
        $S[i] \leftarrow A[p]$
        $i \leftarrow i + 1$
      end if
    end for
  end procedure

First generate a list of integers from 2 to 30:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

First number in the list is 2; cross out every 2nd number in the list after it (by counting up in increments of 2), i.e. all the multiples of 2:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25 ~~26~~ 27 ~~28~~ 29 ~~30~~

Next number in the list after 2 is 3; cross out every 3rd number in the list after it (by counting up in increments of 3), i.e. all the multiples of 3:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ 25 ~~26~~ ~~27~~ ~~28~~ 29 ~~30~~

Next number not yet crossed out in the list after 3 is 5; cross out every 5th number in the list after it (by counting up in increments of 5), i.e. all the multiples of 5:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~ ~~26~~ ~~27~~ ~~28~~ 29 ~~30~~

Next number not yet crossed out in the list after 5 is 7; the next step would be to cross out every 7th number in the list after it, but they are all already crossed out at this point, as these numbers (14, 21, 28) are also multiples of smaller primes because 7*7 is greater than 30. The numbers left not crossed out in the list at this point are all the prime numbers below 30:

2 3 4 5 7 11 13 17 19 23 29

Figure 6.3: Demonstration of Sieve of Eratosthenes

6.1.2.1 Description of working The algorithm utilises two arrays of numbers, one array that holds all numbers from 2 to N, and a second array that holds just the prime numbers from 2 to N.

The algorithm counts from 2 to $\lfloor \sqrt{N} \rfloor$, working on the array of all numbers from 2 to N. If the current field of the array is not 0 (that is, it's a known composite), it marks off all composites of the current number forward of the current position in the array of all numbers from 2 to N. Once it has cycled through the array, all the numbers that are left are primes. Figure 6.3 demonstrates this on a list of numbers from 1 to 30.

6.1.2.2 Implementation An implementation of the Sieve of Eratosthenes can be found in Figure .

,

```
public static int[] Eratosthenes(int maxPrimeNumber){

  // Define new vector A with all numbers from 2 to N
  int[] vectorA = new int[maxPrimeNumber+1];
  for(int index = 2; index <= maxPrimeNumber; index++){
    vectorA[index] = index;
  }

  int nonPrimeCount = 0;
  // Flag all non-primes in vector A.
  int sqrtMaxPrime = (int)Math.sqrt(maxPrimeNumber);
  for(int index = 2; index < sqrtMaxPrime; index++){
    // Skip this number if known composite
    if(vectorA[index] != 0){
      int composite = index * index;
      // Flag all composites of this number as non-prime
      while(composite <= maxPrimeNumber){
        vectorA[composite] = 0;
        nonPrimeCount++;
        composite = composite + index;
      }
    }
  }

  // Create new array from A, filled with just primes.
  int resultIndex = 0;
  int[] primes = new int[maxPrimeNumber-nonPrimeCount+3];
  for(int index = 2; index <= maxPrimeNumber; index++){
    if(vectorA[index] != 0){
      primes[resultIndex++] = vectorA[index];
    }
  }
  return primes;
}
```

Figure 6.4: Sieve of Eratosthenes Algorithm (Java)

6.1.2.3 Sample Problem - Light, More Light There is a man named "Mabu" for switching on-off lights in our University. He switches on-off the lights in a corridor. Every bulb has its own toggle switch. That is, if it is pressed then the bulb turns on. Another press will turn it off. To save power consumption (or may be he is mad or something else) he does a peculiar thing. If in a corridor there is 'n' bulbs, he walks along the corridor back and forth 'n' times and in i'th walk he toggles only the switches whose serial is divisable by i. He does not press any switch when coming back to his initial position. A i'th walk is defined as going down the corridor (while doing the peculiar thing) and coming back again.

Now you have to determine what is the final condition of the last bulb. Is it on or off?

INPUT

The input will be an integer indicating the n'th bulb in a corridor, which is less then or equals 2^{31} - 1. A zero indicates the end of input. You should not process this input.

SAMPLE INPUT

```
3
6241
8191
0
```

OUTPUT

Output "yes" if the light is on otherwise "no" , in a single line.

SAMPLE OUTPUT

```
no
yes
no
```

6.1.2.4 Sample Solution

6.2 String based algorithms

substring search