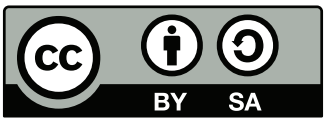
Guide for Advanced Algorithms for Australia and New Zealand Algorithmics & Computing League Competition.

October 12, 2012



Prepared By: Darran Kartaschew

Document Version: 1.0

Last Updated on: October 12, 2012

Contents

List of Figures

List of Algorithms

List of Source Code Implementations and Solutions

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 + 2TB 7200rpm HDD
- nVidia Quadro FX580 graphics card.
- Dual 20” LCD Monitors (1680x1050 resolution).

1.2.1.2 Software

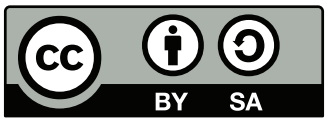
Oracle Solaris 11 11/11, with:

- 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.

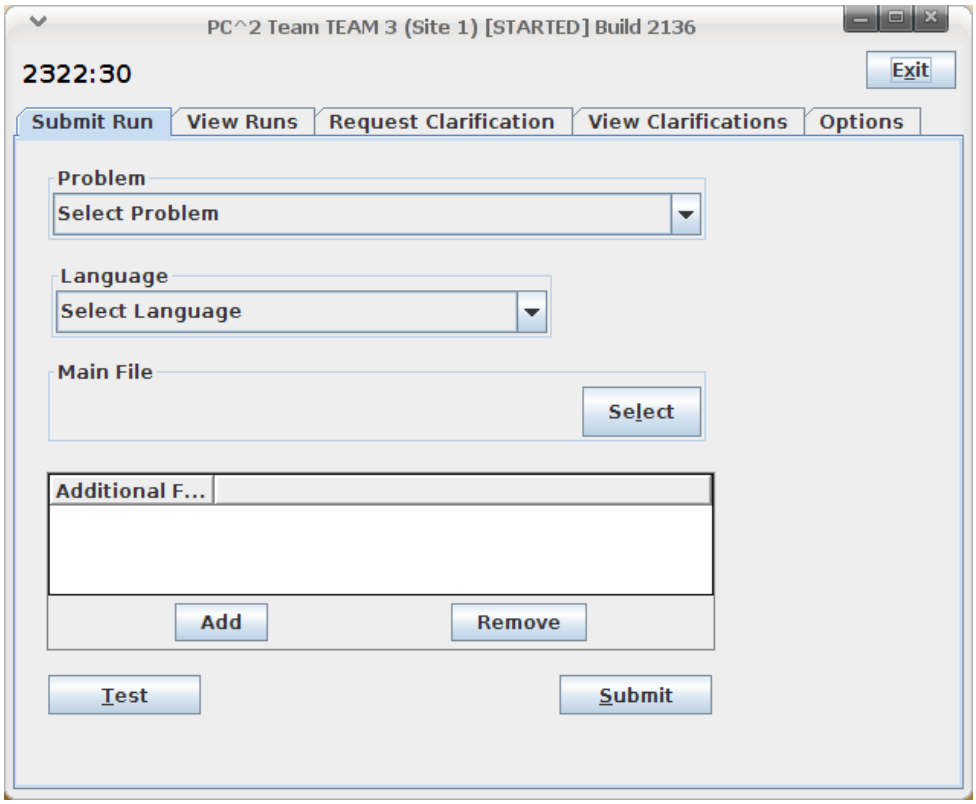


Figure 2.2: PC^2 Client Submit Run Tab

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.

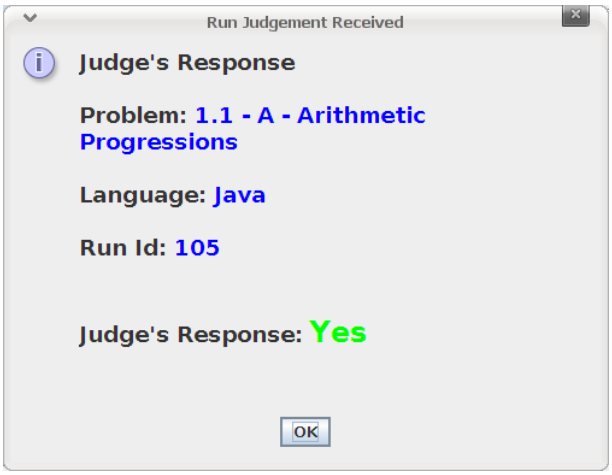


Figure 2.3: Sample Judge's Response

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".

2.1.3 Scoring

If you solve a problem, you get one point for solving the problem. The time of submission since the start of the competition is also recorded. A submission which is judged to be not correct attracts a time penalty, usually twenty minutes, but the penalty does not apply until the problem has been judged correct.

The ordering of the teams is based on the number of correct problems descending and then on the total amount of time used by the team ascending. An example of how the total time is calculated is given in the next section.

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.

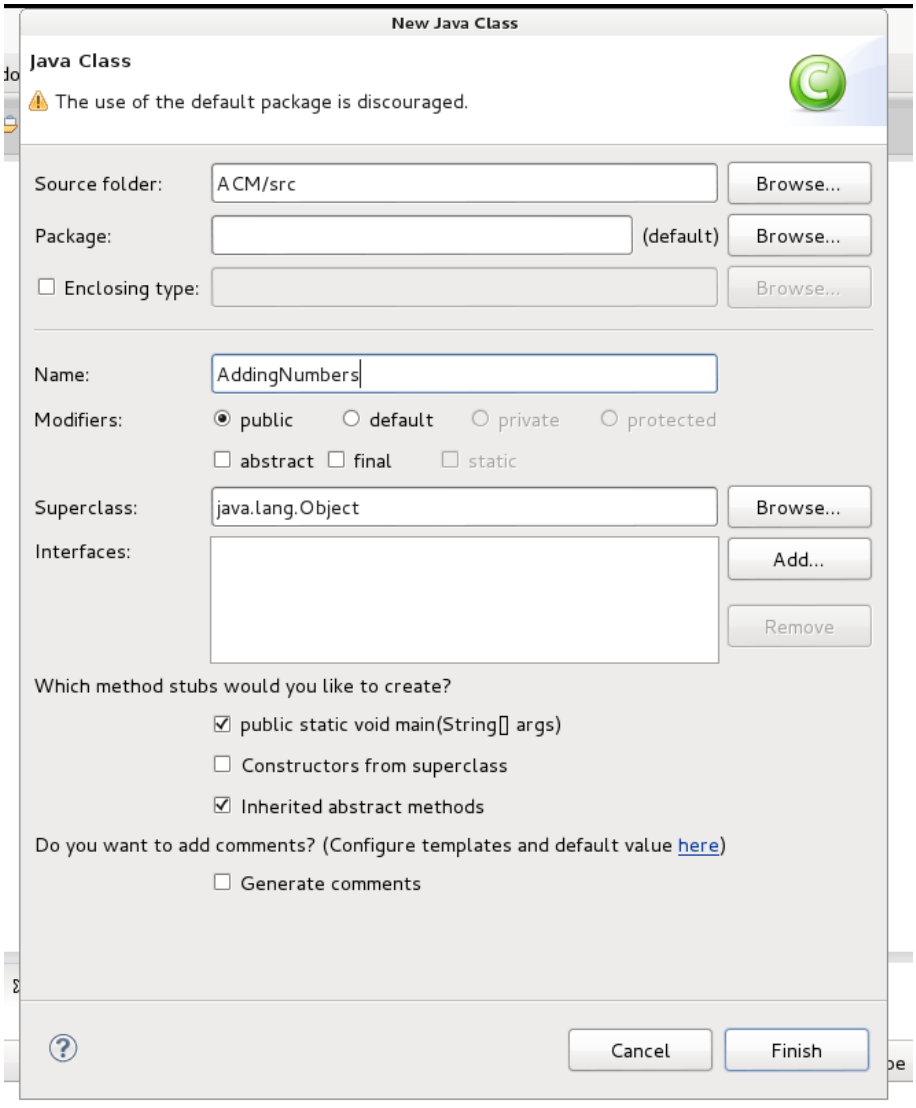


Figure 2.4: Eclipse - New Java Class

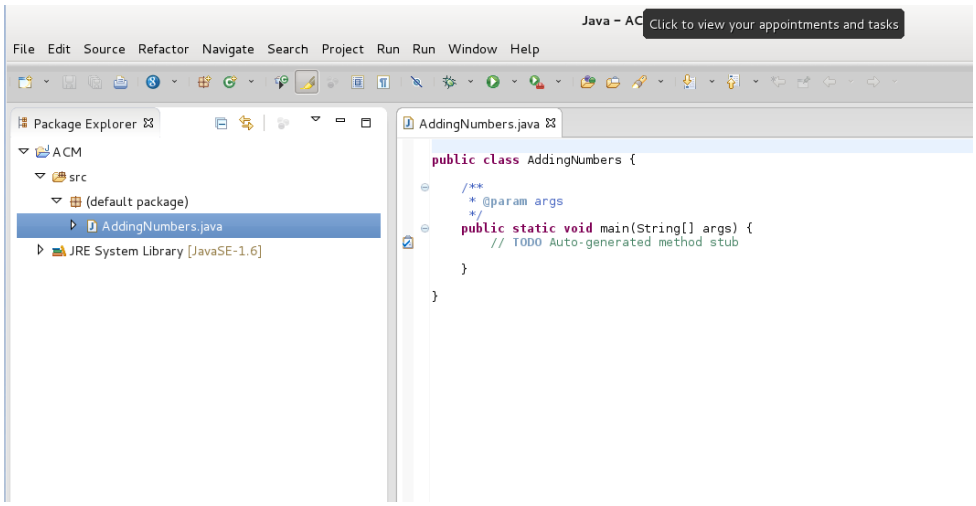


Figure 2.5: Eclipse - New Java Application

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.

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)

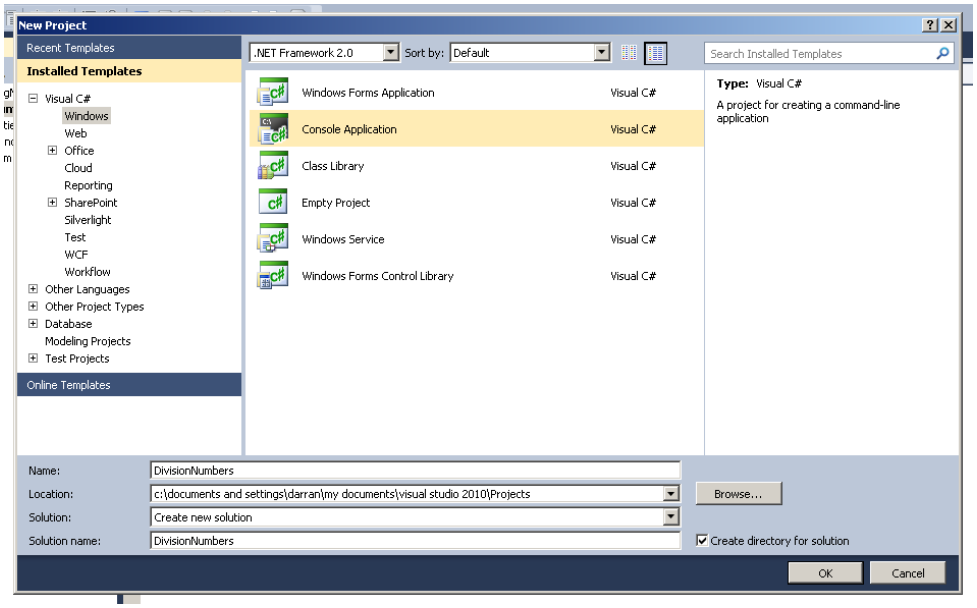


Figure 2.6: Visual Studio C# New Project

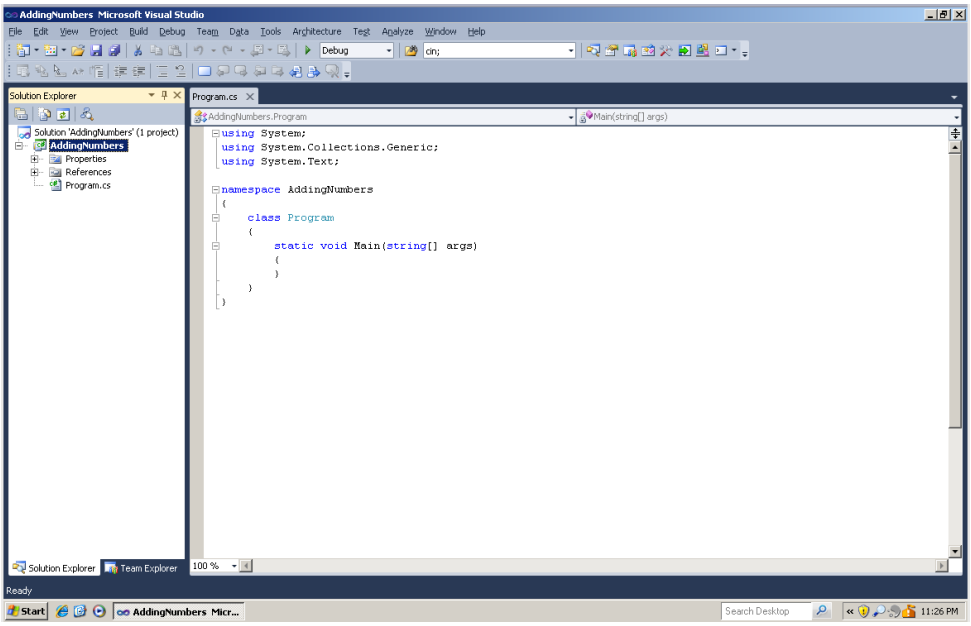


Figure 2.7: Visual Studio C# Program.cs

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.

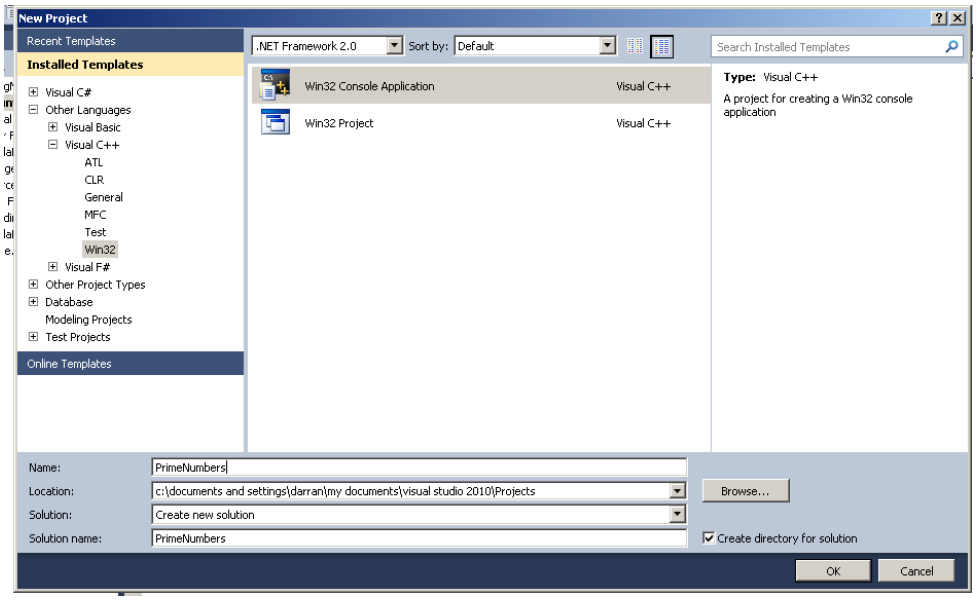


Figure 2.8: Visual Studio C++ New Project

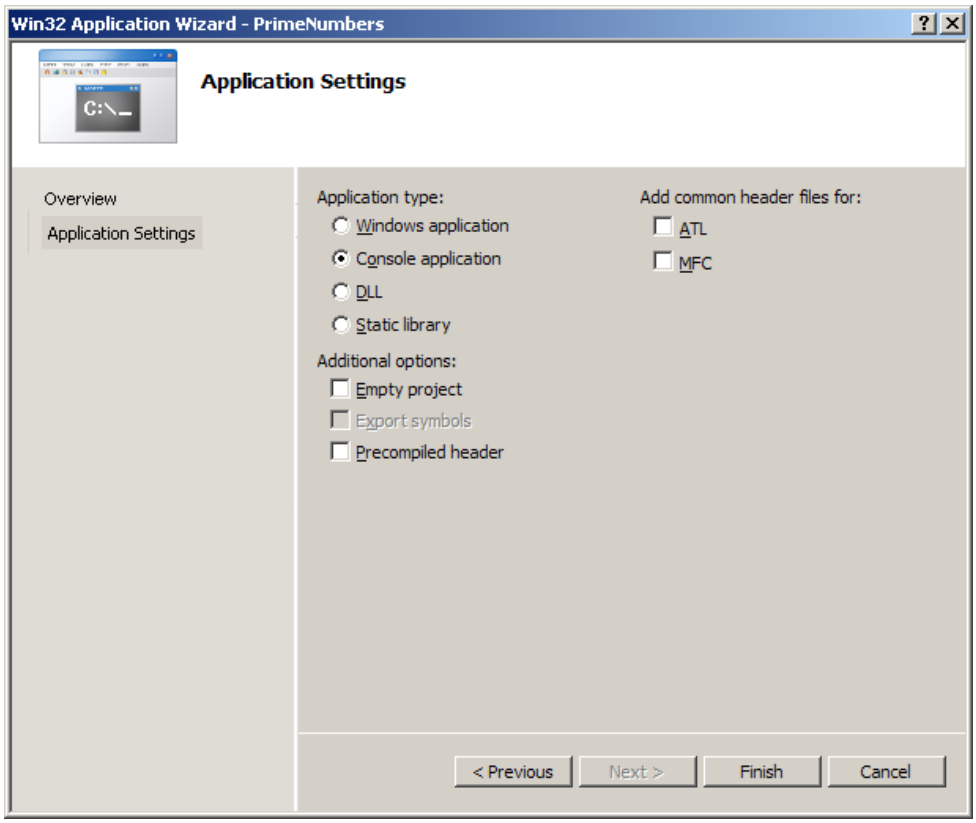


Figure 2.9: Visual Studio C++ Project Options

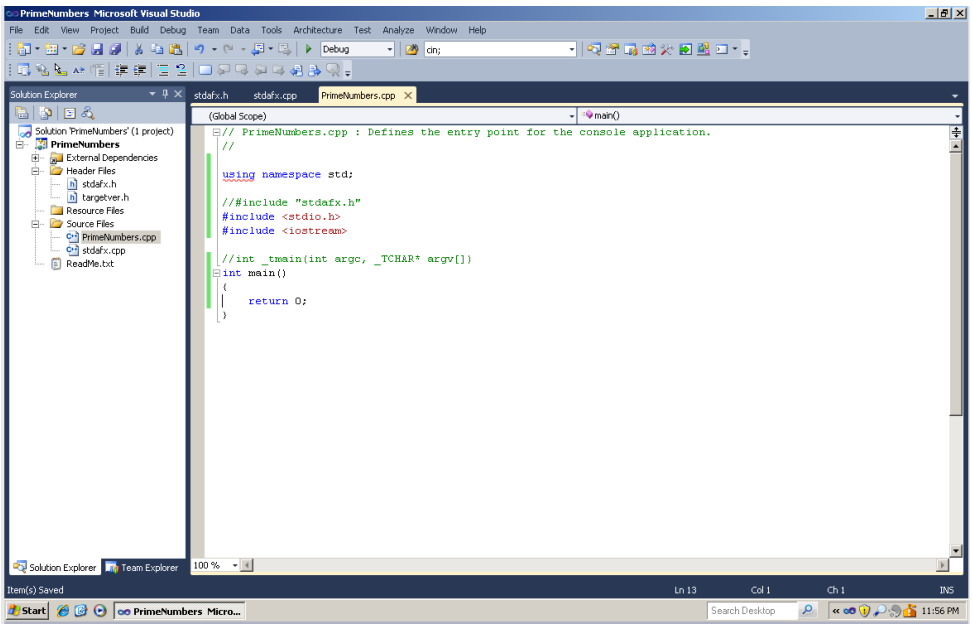


Figure 2.10: Visual Studio C++ Program.cpp

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.

2.5 Competition Strategy

It is important to identify the easier problems in the problem set and solve them correctly before moving on to problems that are more difficult. At the beginning of the competition, the team should quickly identify an easy problem and assign one of the team to solve this problem while the other two try to prioritize the remaining problems based on their understanding of the problems and their abilities to solve them.

Problems should be submitted as soon as the team believes that they have a correct solution. This is based on the scoring system where total time is used to break ties between teams who have solved the same number of problems.

Given that there are three people in the team, more than one problem should be worked on at a time.

At least one of the team members should be writing an algorithm for the next problem to be attempted. There are some benefits to be gained from pair programming while the third team member is working on algorithms.

Alternatively, one person can be writing source code while the other two are working on algorithms.

Teams should also build their own test cases to run against their solutions. This is a very important aspect of the competition and care must be taken to account for the constraints given in the problem specification. The sample data **will not** cover all test cases. The judges' data will.

Many problems are posed that will require the selection of a correct algorithm that runs inside the two minute time limit.

Another important aspect to be aware of is the range of values that may be required in calculations. Some problems are posed that have inputs in a 32 byte integer range (`int`) and an output in the same range 32 byte range but intermediate calculations will overflow the range of the data type, therefore requiring a 64 byte long integer range (`long`).

Output specifications can sometimes be quite picky. Some problems will require capitalization of some words and some will require specific punctuations. Problems are judged by matching string output, so if the characters are in the wrong case or the string lacks punctuation, it will be judged to be a Wrong Answer.

Some problems will be easier to solve if you create your own data types, classes or structs. These must all be included in the same source code file. Ensure you know how to include more than class or struct in a single file.

Ensure that you know how to read input from the keyboard (Standard Input device) using the language or languages that you are likely to use in the competition.

Ensure that you know how to format output to the monitor using the language or languages that you are likely to use.

As part of any preparation for programming competitions it would probably be a good idea to prepare implementations of some of the more complex algorithms that are likely to be encountered. For example, problems in the medium to difficult range can sometimes be transformed into a problem based on a well known graph theory problem e.g. minimum spanning tree, depth first search, shortest path or maximum matching. In order to recognize a problem description as something that can be solved using one of these well known algorithms, it is necessary to know these algorithms and have implemented them before.

Familiarise yourself with your language of choice, especially the associated libraries that are available. Also take careful note on the available collection classes that are available.

Identify the category of the problem, if it's mathematical / dynamic programming / graph theory etc. Ask yourself how familiar are you with it. After doing this you should make decisions regarding the order in which you'll solve them; this goes hand to hand with the next point...

You want to understand the problem completely before typing. Solve the right problem. In my first competitions I thought that if I wasn't typing, I was wasting my time. I later found that this was a mistake.

Don't think comments are a waste of time. At least in "clever" code, you don't want to go debugging line-by-line to see what went wrong (that is a real waste of time). Value clarity.

Have fun.

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

Based on the information in Table 1, should 2^n algorithms be avoided? Absolutely not, if the algorithm solves the problem quickly and easily on provision that the number of elements to be processed is relatively small. To illustrate, suppose the each operation can be done in 4 microseconds, but we only have 16 elements to be processed. The respective worst case runtimes for each function is shown in Table 2.

With this comparison, algorithm selection is not merely about the one with the least Order value, but also about understanding the number of items to be processed compared to the implementation complexity of the algorithm itself. (One would use an algorithm with **O(2^n)** complexity if implementation only takes 5 minutes, versus an algorithm with **O(n)** complexity if it takes hours to implement and test, if the number of input elements is small). Sometimes brute force is better than perfect elegance in the competition!

The information within this section is a very brief introduction to BigO notation, has been 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.

Function	Time
$log_2 n$	16 microseconds
n	64 microseconds
$nlog_2 n$	256 microseconds
n^2	1024 microseconds
n^3	16 milliseconds
2^n	256 milliseconds

Table 2: Big O runtimes

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. Listing 1 shows an example of the syntax.

```
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);
```

Listing 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. Listing 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);
```

Listing 2: Timing - C++

[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.

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]. Listing 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);
```

Listing 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 Listing 4 demonstrates the performance differences of using 'int', 'long' and 'double' for a simple add, multiple and divide sequence.

,

```
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;
    }
```

[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.

```
      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);
  }
}
```

Listing 4: Source code for Timing test

The results[4] for the above test in Listing 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.

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. Liberal 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.
8. Utilisation of the most efficient algorithm, even in cases where memory requirements may be pushed to the extreme. eg, building complete hashtables in memory for **O(1)** lookups, rather than recomputing as needed.

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.

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

[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. Listing 5 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 Listing 5 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);
      }
    }
  }
}
```

Listing 5: C# Input Example

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++). Listing 6 shows the same application written in C++, as shown in C# within Listing 5.

```
#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;
}
```

Listing 6: 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 displayed in Listing 7.

```
#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;
}
```

Listing 7: C++ getline() example

[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.

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.

4.1.2.2 Output The primary method of output is via the `cout` iostream, as shown in Listing 6. 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 citing that there is no performance difference. Utilising the source code in Listing 8, 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 Listing 8, yeilds the following results:

1. cout time = 67240 msec

2. printf() time = 280 msec

It should be heavily stressed, that the above times are indicative of worst cases only, and in practice and real-word the difference is less pronounced. For example, enabling all compiler optimisations that enable SSE4 and auto-parallelisation support reduces the difference to:

1. cout time = 700 msec

2. printf() time = 280 msec

Indicating a time penality of just over 2 times. While this penalty isn't as poor as earlier indicated, within the competition it may mean the difference between getting a "Yes" response and a "Time Overrrun" response.

```
#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;
}
```

Listing 8: 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 Listing 9 which solves the same problem as shown in Listing 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();
      }
   }
}
```

Listing 9: Java Input and Output Example

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 Listing 9.

4.1.3.3 Special Notes In addition to primitive data types found in other languages such as `int`, `double`, `float`, `string`, Java also has a `BigInteger` and `BigDecimal` data types that is capable of dealing with numbers that `double` and `long` cannot deal with. The use of these two data types may allow some problems to be easily solved, that would otherwise necessitate complex code when dealing with numbers that cannot be represented by either `long` or `double`.

4.2 Program Structure

For all solutions to ACM and ANZAC problems, the entire solution must be written in one file. It is suggested for clarity during the competition that you name your programs simply. Problems will usually be associated with a letter (A – I) or a number (1 – 9). Name your main class (if applicable) and therefore the source code file for the problem, something that represents the problem being presented.

4.2.1 C# and Java

For both Java and C#, due to the pure OOP paradigm being presented there are few recommendations[8] that may assist in development of yout solutions.

The main class will naturally require a main method. It is suggested that you instantiate an object of this class and write a non-static method that can access any global instance variables you may need for multiple test cases. (An example is shown in Listing 10). Attempt to avoid writing all the methods as static methods, as this can lead to some complications if not done carefully.

There should be no need to use public or private modifiers for your instance variables. There is no need for the strict OO rules to be followed, just as long as you understand what side effects you are writing in to your methods.

If the problem requires you to define a class it needs to be in the same file. If you make it an inner class of the class you write for your main program, all instance variables of that inner class will not need to be declared public for accessibility.

If your program requires you to do specific output from this inner data class, make sure you use a `toString()` (Java) or `ToString()` (C#) method to format that output as required. Because the `toString()` method would be overriding the `toString()` method from the Object base class, it must have the public modifier.

,

```
// imports go first
public class PA {
  public static void main(String[] args) {
    new PA().run()
  }

  public void run() {
    // start of solution goes here
  }
}
```

Listing 10: Java Program Structure

4.2.2 C++

Due to C++ being a language that allows both OOP and Structured Programming paradigms, there are no special considerations required. However some of the recommendations may be followed as those defined in Section 4.2.1 as needed.

[8] These are recommendations, not hard rules. Feel free to break them or ignore them completely as needed.

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 Listing 12 and Listing 12 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;
}
```

Listing 11: 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;
}
```

Listing 12: 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 Listing 13. 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.

```
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])));
    }
  }
}
```

Listing 13: Solution to Linear Search (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
    $mid \leftarrow \lfloor (low + high)/2 \rfloor$
    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 Listing 14 shows an implementation of the binary search in Java.

```
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);
    }
```

```
  }
}
```

Listing 14: Binary Search (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 Listing 15. 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.

```
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));
    }
  }
}
```

Listing 15: Solution to Binary Search Problem (Java)

5.3 Array Handling

When talking about arrays, we typical define one as a single string[9] 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[10] 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.

[9] 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.

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

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 data, and are typically 16kilobytes size in size[11]. 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[12]).

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[13], 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[14].

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[15]): "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 Listing 16 yeilds the results show in Table 3. 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.

[11] 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.

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

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

[14] 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.

[15] 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).

This clearly demonstrates the caches hits/misses taking place and confirms the latencies expected by the cache misses.

,

```
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;
}
```

Listing 16: 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 3: Array Access Performance

5.3.2 Array Traversal Methods

Any 2-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 Listing 16, 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[16].

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 **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 and 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:

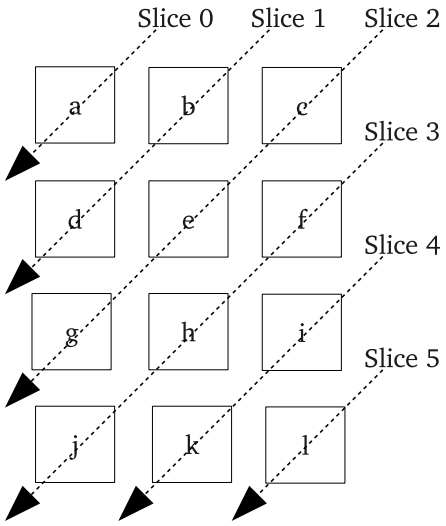
- Top-left origin to bottom-right traverse: print $S[w][v]$
- Bottom-left origin to top-right traverse: print $S[m - w - 1][v]$
- Top-right origin to bottom-left traverse: print $S[w][n - v - 1]$
- Bottom-right origin to top-left traverse: print $S[m - w - 1][n - v - 1]$

Figure 5.1 demonstrates the top-left origin to bottom-right traversel on a 3 $\times$ 4 array, with each slice (or starting location) shown.

[16]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`



S: 3 × 4 matrix

Figure 5.1: Diagonal Traversal of an Array with a Top-Left Origin

5.3.3.2 Implementation Listing 17 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;
    }
  }
}
```

Listing 17: 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(n^2)** 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 additional variable is needed to hold an offset, so that the new array offsets may be calculated correctly[17]. Figure 5.2 shows the clockwise rotation operation in effect.

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

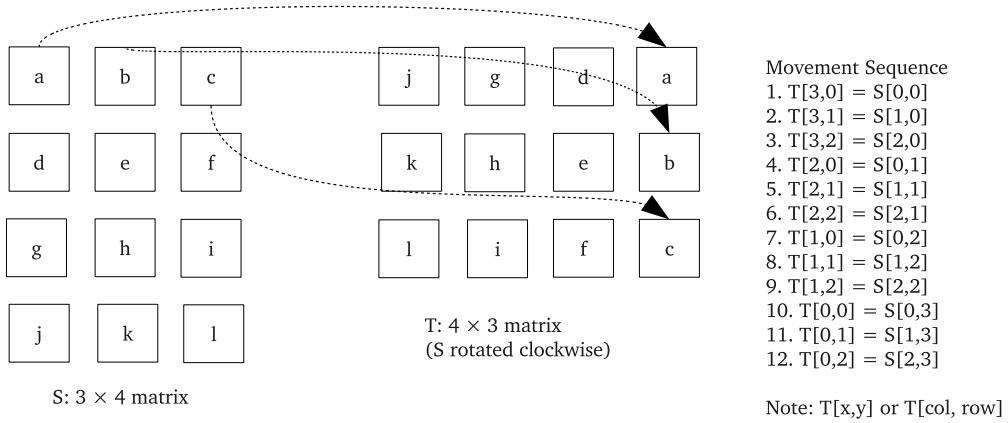


Figure 5.2: Array Rotation

5.3.4.2 Implementation Listing 18 and 19 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];
    }
  }
}
```

Listing 18: Array Clockwise Rotation (Java)

```
public void ArrayAntiRotateClockWise(E[][] source, E[][] target, 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];
    }
  }
}
```

Listing 19: Array Anti-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.

5.3.4.3 Sample Problem- Matrix Rotation Given a two dimension matrix measuring n × 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 Listing 23. 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();
    }
  }
}
```

Listing 20: 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 − 1][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 − 1] ← 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.

5.3.5.1 Description of working Algoritms 8 and 9 depict both vertical and horizontal flipping respectively. Simply, both perform substitution of x and y values for the target array. Figure shows the clockwise rotation operation in effect.

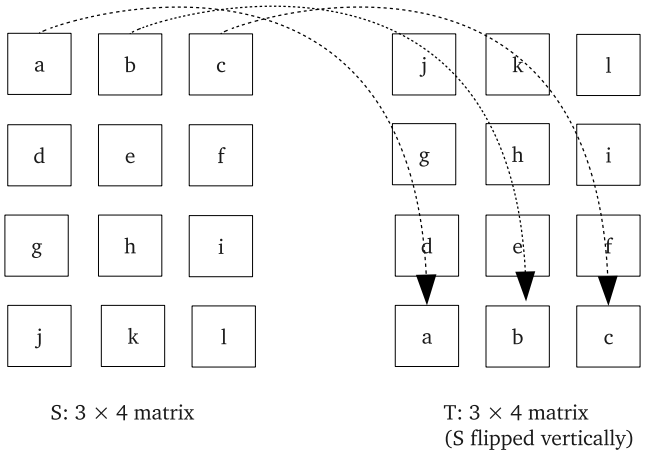


Figure 5.3: Array Flipping

5.3.5.2 Implementation Listing 21 and 22 show implementations of vertical and horizontal mirroing respectively.

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

Listing 21: Mirror Array Along Vertical Axis(Java)

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

Listing 22: Mirror Array Along Horizontal Axis(Java)

5.3.5.3 Sample Problem- Matrix Flipping Given a two dimension matrix measuring n $\times$ m, flip the matrix in a vertical direction, followed by a horizontal direction.

INPUT

Input consists of one or more lines, with the first line being the size of the matrix to flip 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 flipped, 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 flip operation should be the resultant matrix, without spaces between each column in the matrix. There must a blank line separating each resultant matrix.

SAMPLE OUTPUT

```
001
000
101

00
10
```

5.3.5.4 Sample Solution A solution to the sample problem utilising an array rotation can be seen in Listing 23. 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 flip. Flip the array vertically, then horizontally, 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 ArrayFlip {

  /**
   * Flip an Array in a vertical 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 source array
   */
  public static void ArrayVerticalFlip(int[][] source, int[][] target, int width,
                                       int height) {
    for (int y = 0; y < height; y++) {
      for (int x = 0; x < width; x++) {
        target[(height - y - 1 )][x] = source[y][x];
      }
    }
  }

  /**
   * Flip an Array in a horizontal 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 source array
   */
  public static void ArrayHorizontalFlip(int[][] source, int[][] target, int width,
                                         int height) {
    for (int y = 0; y < height; y++) {
      for (int x = 0; x < width; x++) {
        target[y][(width - x - 1)] = 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]);
          }
        }

        // Flip the matrix and print.
        int[][] target = new int[height][width];
        ArrayVerticalFlip(matrix, target, width, height);
        ArrayHorizontalFlip(target, matrix, width, height);

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

Listing 23: Solution to Array Flipping Problem (Java)

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 3rd 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 ≠ 0 do                          ▷ We have the answer if n is 0
        r ← m mod n
        m ← n
        n ← 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 mod n)
end procedure
```

6.1.1.2 Implementation An implementation of the iterative form of the algorithm can be found in Listing 24.

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

Listing 24: 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 Listing 25.

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).

,

```
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));
    }
  }
}
```

Listing 25: 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.

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 $\left\lfloor \sqrt{N} \right\rfloor$, 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*. During the main loop, we set the next composite value to be $index^2$, as all lower multiple of $index$, have already been flagged as being known. Once it has cycled through the array, all the numbers that are left are primes. Figure 6.1 demonstrates this on a list of numbers from 1 to 30.

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 ← 2 to n do
        A[p] ← p
    end for
    for p ← 2 to ⌊√n⌋ do
        if A[p] ≠ 0 then
            j ← p^2
            while j ≤ n do
                A[j] ← 0
                j ← j + p
            end while
        end if
    end for

    i ← 0                                   ▷ Copy list of primes from A to S
    for p ← 2 to n do
        if A[p] ≠ 0 then
            S[i] ← A[p]
            i ← 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 6.1: Demonstration of Sieve of Eratosthenes

6.1.2.2 Implementation An implementation of the Sieve of Eratosthenes can be found in Listing 26.

```
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){
        if(vectorA[composite] != 0){
          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-1];
  for(int index = 2; index <= maxPrimeNumber; index++){
    if(vectorA[index] != 0){
      primes[resultIndex++] = vectorA[index];
    }
  }
  return primes;
}
```

Listing 26: Sieve of Eratosthenes Algorithm (Java)

6.1.2.3 Sample Problem - Primes for Hashtable Student Programmer Joe is developing a dynamic hash table class in Java, and from his textbooks knows that prime numbers make the best values to use within the hashing function. He would like a program to tell him if the number he is using is a prime, and if not what's the closest prime to but below to the number he has given the program.

INPUT

The input will be an integer indicating the number Joe inputs to be tested for being a prime, which is within the range of $0 \leq N \leq 2^{16} - 1$. A zero indicates the end of input. You should not process this input.

SAMPLE INPUT

```
3
6243
8191
10
0
```

OUTPUT

Output "`prime`" if a prime number, otherwise the closest prime below the number given.

SAMPLE OUTPUT

```
prime
6421
prime
7
```

6.1.2.4 Sample Solution A sample solution for the Primes of Hash Table problem can be found in Listing 27. There are 6542 prime numbers in the specified number space.

There are a number of methods to solve this problem, including:

1. Testing each number input as a prime using the divisibility test, and if not a prime decrementing the number by 1 and retesting until a prime is found.
2. Create a list of all numbers within the test space in an array and set all array elements that respresent composites to 0. To test each number given, we simple look up the slot for that number in the array, and determine if it's a prime or not[18]. If it's not a prime, we use a reverse linear search through the array until a prime is found, and return the found prime.
   This offers, **O(1)** performance for testing if a number is a prime within the array, but up to **O(71)** if the number supplied is not a prime. The **O(71)** is determined by the largest gap between prime numbers in the number space (being 71), and would be the worst case for a linear search operation if needed.
3. Create a list of all prime numbers in the test space, and store only these in an array[17]. Utilising a binary search to search for the number in the list, we can determine if the number is prime based on $\mathbf{O(\log_2 n)}$ performance. With a slight modification to the binary search method, instead of returning -1 if not found, we can return the index of the last tested value, which incidently will the next prime below the number being tested. Therefore we can assume $\mathbf{O(\log_2 n)}$ performance in the worst case for each number test. (With 6542 primes, this would result in **O(13)** in the worst case).

For this problem, I utilised the third option listed above for three main reasons:

1. The memory space requirement is reduced after the lists of primes has been calculated[19], allowing more of the list of numbers to fit into CPU cache. (Option 2 will utilise 256KB of RAM vs Option 3 with 26KB of RAM). While 256KB vs 26KB may not seem a lot, 26KB will fit or mostly fit into the Level 1 cache on most CPUs thereby increasing performance. If we expand the address space to the maximum value being 2^{31}-1, Option 2 will utilise 8GB of RAM, whilst Option 3 will utilise 420MB of RAM once the lists have been created.
2. The time needed for determining if a prime is excellent with Option 2, but suffers some performance penalty if a linear search is needed to find the next lower prime. The largest gap in the required test space is 71 numbers, which may negatively effect performance. If we expand the address space to the maximum value being 2^{31}-1, in Option 2 the largest gap between primes becomes 354 numbers, therefore requiring up to 354 operations for the linear search. Option 3 suffers little performance penalty, as the search remains at $\mathbf{O(\log_2 n)}$, where n = 105,097,565, or **O(26)** to determine if a prime or the next lower prime.
3. Utilising the Sieve of Eratosthenes, the resulting list of primes is naturally sorted allowing the use of a binary search to find the prime number, without additional work.

Option 1 was not considered due to the lack of CPU efficiency within the proposed solution.

The implemented solution performs the following steps:

1. Create a list of primes in an array utilising the Sieve of Eratosthenes algorithm. (If a complete array can not fit into memory, an alternate Segmented Sieve of Eratosthenes may be used to build the list of primes).

[18] This allows the search for prime to become a lookup operation rather than a calculation operation improving performance.

[19] Whilst during the creation of the list of primes, large amounts of memory is needed, once the list has been created, the needed space to hld the results is vastly smaller.

2. Reads in the next number to be tested. (If '0', exit).
3. Utilise a binary search to find the index of the number in the list of primes, and test the value stored against the number to determine if prime or not. If it is not a prime, the index points to the next prime lower than the number entered.

To test the implementation, a range of number were randomly selected to test the basic operation. To ensure that the implementation was robust, the following numbers were also tested:

- Number 1. Number 1 is neither a prime, nor composite. In this case, I return 1 due to 1 being non-prime.
- Number 2.
- Less than 0, which returns 2. (Less than 0, is invalid input, so can't be considering incorrect).
- More than 2^{16}, which the first prime less than 2^{16}.

Since a binary search was used, I had to pay little attention to values outside the input space, as the results of the binary search would always return an index that was within the address space. The exception to this was the number 1, in which a special case is used.

,

```
import java.util.Scanner;
public class PrimesForHashtable {
  /* Largest value with input space */
  final static int PRIME_SPACE = (int) Math.pow(2, 16);

  /**
   * Utilise Sieve of Eratosthenes to create an array of prime numbers
   */
  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) {
          if (vectorA[composite] != 0) {
            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-1];
    for (int index = 2; index <= maxPrimeNumber; index++) {
      if (vectorA[index] != 0) {
        primes[resultIndex++] = vectorA[index];
      }
    }
    return primes;
```

```
  }

  /**
   * Perform binary search of array to locate prime number.
   */
  public static int BinarySearch(int[] vector, int key, int low, int high) {
    if (low > high) {
      return (high >= 0 ? high : 0);
            // Return the index of the current high value, as this
            // points to 1 below the key, if not found
    }
    int mid = (low + high) / 2;
    if (vector[mid] == key) {
      return mid;
    } else {
      if (vector[mid] > key) {
        return BinarySearch(vector, key, low, (mid - 1));
      } else {
        return BinarySearch(vector, key, (mid + 1), high);
      }
    }
  }

  /**
   * Returns a prime equal to or below to the number given.
   */
  public static int LookUpPrime(int[] primes, int number) {
    return primes[BinarySearch(primes, number, 0, primes.length-1)];
  }

  /**
   * Main
   */
  public static void main(String[] args) {

    // Create a list of primes for the given input space.
    int[] primes = Eratosthenes(PRIME_SPACE);

    Scanner in = new Scanner(System.in);
    String line = in.nextLine();

    // Keep processing input until 0.
    while (!line.equals("0")) {
      // Get our number, and it's closest prime.
      int number = Integer.parseInt(line);
      // Handle special case of number 1.
      if (number != 1) {
        int prime = LookUpPrime(primes, number);
        // Display if it's a prime, or closet prime.
        if (prime == number) {
          System.out.println("prime");
        } else {
          System.out.println(prime);
        }
      } else {
        System.out.println("1");
      }
      line = in.nextLine();
    }
  }
}
```

Listing 27: Solution to Primes for Hashtable (Java)

6.2 String based algorithms and data structures

String are a fundamental data type with a high importance in the following areas:

- Web crawlers utilised by search engines classify billions of text documents each day.
- Bioinformatics Research, particularly arounfg genome based research.

Strings are effectively a collection of characters, with each character being defined as part of an alphabet. This alphabet, may consist of only a single character, a small number of characters, (eg ASCII with 127 characters), a large number of characters (eg Unicode with 1,114,112 characters), or all possible values represented by either a byte (256 characters), word (65536 characters) or double-word(2^{32} characters). One common misconseption is that a string may only represent those letters and figures as defined in a human langauge, but from a Computers viewpoint any value that can be defined may be included (or even excluded) from the alphabet that defines a string.

Typically, we allow A to be an alphabet (finite set), with:

- *pattern* and *text* are vectors of elements of A.
- A may be:
  - usual human alphabet
  - binary alphabet = A = {0,1}
  - DNA alphabet = A = {A, C, G, T}
  - etc.

When dealing with strings in different languages, the meaning and implementation may vary.

- C99 - A string is defined as a vector of char (or byte), utilising ASCII[20] or UTF-8[21] encoding terminated by the character *NUL* (0).
- C++ - A string may be that as C99, or of an object wrapped around a C99 string. Some operating systems like Windows allow a Wide-Character (16bits) to be utilised in place of single byte characters.
- Java - A string is a vector of UTF-16 encoded characters, represented as an object, and is immutable.
- C# (with .NET) - A string is a vector of UTF-16 encoded characters, represented as an object, and is immutable.

There are numberous algorithms that perform string matching, including Brute Force, Rabin-Karp, Knuth-Morris-Pratt, Boyer-Moore, and Horspool. Regular Expressions are also considered a form of pattern matching, with the advantage that they allow an infinite number of patterns to be matched.

This guide will discuss the Brute Force and Knuth-Morris-Pratt algorithms.

6.2.1 Brute Force Substring Search

Brute Force Substring search is considered a naive based algorithm that utilises a very simple method to determine string or pattern matching.

The algorithm walks through the vector of text, attempting to match the pattern in sequence.

[20] Other non-PC systems may utilise alternate encoding systems such as EBCIDC in place of ASCII encoding.
[21] Modern UNIX implementations like FreeBSD, Solaris and GNU/Linux use UTF-8.

Algorithm 13 Brute Force String Matching

Input A Vector T of n characters representing a text, and vector P of m characters representing a pattern to find in T.

Output The index of the first character of P when a complete match is found in T, otherwise -1 if search was unsuccessful.

```
procedure BRUTEFORCESTRINGMATCH(T, P)
    for i ← 0 to n − m do
        j ← 0
        while j < m and P[j] = T[i + j] do
            j ← j + 1
        end while
        if j = m then
            return i
        end if
    end for
    return −1
end procedure
```

6.2.1.1 Description of working The Brute Force String Matching algorithm is a very simple algorithm that performs the following steps:

1. Align patternw at the beginning of the text.
2. Moving from left to right, compare each character of the pattern to the corresponding character in the text until:
   (a) All characters are found to match; or
   (b) A mismatch is detected.
3. While pattern is not found, and the text is not exhausted, relaign pattern one position to the right and repeat step 2.

Figure 6.2 has an example of Brute Force matching. (The pattern's characters that are compare with their text counterparts are in bold type).

Brute Force String Matching is considered a **O(n^2)** operation, however may offer worse performance if many clost matches are present within the text.

```
N  O  B  O  D  Y  _  N  O  T  I  C  E  D  _  H  I  M
N  O  T
   N  O  T
      N  O  T
         N  O  T
            N  O  T
               N  O  T
                  N  O  T
                     N  O  T
                        N  O  T        ←(Match Found)
```

Figure 6.2: Example of Brute Force String Matching

6.2.1.2 Implementation An implementation of the Brute Force String Matching can be found in Listing 28.

```
int BruteForceStringMatch(char* text, int textLength,
                          char* pattern, int patternLength){
  int patternIndex = 0;
  // Scan the text looking for the pattern
  for(int textIndex = 0; textIndex <= textLength-patternLength; textIndex++){
    patternIndex = 0; // Reset index into pattern
    // While the current text position matches the pattern, keep scanning it.
    while((patternIndex < patternLength)
             &&(pattern[patternIndex] == text[textIndex+patternIndex])){
      patternIndex++;
    }
    // If the patternIndex is equal to patternLength, we have found the pattern!
    if(patternIndex == patternLength){
      return textIndex;
    }
  }
  // Pattern not found, so return -1.
  return -1;
}
```

Listing 28: Brute Force String Matching (C++)

6.2.1.3 Sample Problem - Where's Waldorf Given a m by n grid of letters, ($1 \leq m, n \leq 50$), and a list of words, find the location in the grid at which the word can be found. A word matches a straight, uninterrupted line of letters in the grid. A word can match the letters in the grid regardless of case (i.e. upper and lower case letters are to be treated as the same). The matching can be done in any of the eight directions either horizontally, vertically or diagonally through the grid.

INPUT

The input begins with a single positive integer on a line by itself indicating the number of the cases following, each of them as described below. This line is followed by a blank line, and there is also a blank line between two consecutive inputs.

The input begins with a pair of integers, m followed by $n, 1 \leq m, n \leq 50$ in decimal notation on a single line. The next m lines contain n letters each; this is the grid of letters in which the words of the list must be found. The letters in the grid may be in upper or lower case.

Following the grid of letters, another integer k appears on a line by itself ($1 \leq k \leq 20$). The next k lines of input contain the list of words to search for, one word per line. These words may contain upper and lower case letters only (no spaces, hyphens or other non-alphabetic characters).

SAMPLE INPUT

```
1

8 11
abcDEFGhigg
hEbkWalDork
FtyAwaldORm
FtsimrLqsrc
byoArBeDeyv
Klcbqwikomk
strEBGadhrb
yUiqlxcnBjf
4
Waldorf
Bambi
Betty
Dagbert
```

OUTPUT

For each test case, the output must follow the description below. The outputs of two consecutive cases will be separated by a blank line.

For each word in the word list, a pair of integers representing the location of the corresponding word in the grid must be output. The integers must be separated by a single space. The first integer is the line in the grid where the first letter of the given word can be found (1 represents the topmost line in the grid, and m represents the bottommost line). The second integer is the column in the grid where the first letter of the given word can be found (1 represents the leftmost column in the grid, and n represents the rightmost column in the grid). If a word can be found more than once in the grid, then the location which is output should correspond to the uppermost occurence of the word (i.e. the occurence which places the first letter of the word closest to the top of the grid). If two or more words are uppermost, the output should correspond to the leftmost of these occurences. All words can be found at least once in the grid.

SAMPLE OUTPUT

```
2 5
2 3
1 2
7 8
```

6.2.1.4 Sample Solution A sample solution to the Where's Waldorf problem can be found in Listing 29.

```
#include <cstdlib>
#include <iostream>
#include <cctype>
#include <cstring>

using namespace std;

char** wordmap;
int testCases = 0;
int wordmapWidth = 0;
int wordmapHeight = 0;
char blankline[256];
int wordSearches = 0;
char* pattern = NULL;
int patternLength = 0;
char* text = NULL;
int location = -1;
int currentRow = 0;
int currentCol = 0;

/**
 * Attempt to find a pattern contained within the text.
 */
int BruteForceStringMatch(char* text, int textLength, char* pattern, int patternLength) {
  int patternIndex = 0;

  // Ensure the text is big enough to hold our pattern!
  if (patternLength > textLength) {
    return -1;
  }

  // Scan the text looking for the pattern
  for (int textIndex = 0; textIndex <= textLength - patternLength; textIndex++) {
    patternIndex = 0;
    // While the current text position matches the pattern, keep scanning it.
    while ((patternIndex < patternLength) && (pattern[patternIndex] ==
                                              text[textIndex + patternIndex])) {
      patternIndex++;
    }
    // If the patternIndex is equal to the patternLength, we have found the pattern!
    if (patternIndex == patternLength) {
      return textIndex;
    }
```

```
  }
  // Pattern not found, so return -1.
  return -1;
}

/**
 * Create a search pattern of length, from the word map.
 * @param col starting position in the word map (0 offset)
 * @param row starting position in the word map (0 offset)
 * @param length length of required string.
 * @param colOffset direction to move column index for each letter.
 * @param rowOffset direction to move row index for each letter.
 * @return Index of found string.
 */
char* buildTextString(int col, int row, int length, int colOffset, int rowOffset) {
  char* newString = new char[length + 1];
  int index = 0;
  while (length--) {
    newString[index++] = wordmap[row][col];
    row += rowOffset;
    col += colOffset;
    // Ensure we are not going outside map boundary
    if ((row < 0) || (row >= wordmapHeight) || (col < 0) || (col >= wordmapWidth)) {
      newString[index] = NULL;
      return newString;
    }
  }
  newString[index] = NULL;
  return newString;
}

/**
 * Convert string to lower case, and store in place
 * @param str ASCIIZ string to convert.
 */
void strToLower(char* str) {
  int i = 0;
  while (str[i]) {
    str[i] = (char) tolower(str[i]);
    i++;
  }
}

/**
 * Update the best found location, based on current row/col positions
 */
void UpdateFind(int row, int col, int rowOffset, int colOffset, int location) {
  if (currentRow > row + (rowOffset * location)) {
    currentRow = row + (rowOffset * location);
    currentCol = col + (colOffset * location);
  } else if (currentRow == row + (rowOffset * location)) {
    if (currentCol > col + (colOffset * location)) {
      currentRow = row + (rowOffset * location);
      currentCol = col + (colOffset * location);
    }
  }
}

/**
 * Main
 */
int main() {
  scanf("%d", &testCases);
  while (testCases--) {
    gets(blankline);
    // Read in the wordmap, and ensure in lower case.
    scanf("%d␣%d", &wordmapHeight, &wordmapWidth);
    wordmap = new char*[wordmapHeight];
    for (int row = 0; row < wordmapHeight; row++) {
      wordmap[row] = new char[wordmapWidth + 1];
      scanf("%s", wordmap[row]);
      strToLower(wordmap[row]);
    }
```

```
// Read in the number of words to search for.
scanf("%d", &wordSearches);
while (wordSearches--) {
  // Read in a single word.
  pattern = new char[wordmapWidth + 1];
  scanf("%s", pattern);
  strToLower(pattern);
  patternLength = strlen(pattern);

  currentRow = wordmapHeight;
  currentCol = wordmapWidth;

  //Attempt to scan for word, along the top of the grid.
  for (int col = 0; col < wordmapWidth; col++) {
    int row = 0;
    int rowOffset = 1;
    for (int colOffset = -1; colOffset < 2; colOffset++) {
      // Get our next text string, and attempt to match!
      text = buildTextString(col, row, 50, colOffset, rowOffset);
      location = BruteForceStringMatch(text, strlen(text), pattern, patternLength);
      if (location >= 0) {
        UpdateFind(row, col, rowOffset, colOffset, location);
      }
      delete(text);
    }
  }

  //Attempt to scan for word, along the bottom of the grid.
  for (int col = 0; col < wordmapWidth; col++) {
    int row = wordmapHeight - 1;
    int rowOffset = -1;
    for (int colOffset = -1; colOffset < 2; colOffset++) {
      // Get our next text string, and attempt to match!
      text = buildTextString(col, row, 50, colOffset, rowOffset);
      location = BruteForceStringMatch(text, strlen(text), pattern, patternLength);
      if (location >= 0) {
        UpdateFind(row, col, rowOffset, colOffset, location);
      }
      delete(text);
    }
  }

  //Attempt to scan for word, along the left of the grid.
  for (int row = 0; row < wordmapHeight; row++) {
    int col = 0;
    int colOffset = 1;
    for (int rowOffset = -1; rowOffset < 2; rowOffset++) {
      // Get our next text string, and attempt to match!
      text = buildTextString(col, row, 50, colOffset, rowOffset);
      location = BruteForceStringMatch(text, strlen(text), pattern, patternLength);
      if (location >= 0) {
        UpdateFind(row, col, rowOffset, colOffset, location);
      }
      delete(text);
    }
  }

  //Attempt to scan for word, along the right of the grid.
  for (int row = 0; row < wordmapHeight; row++) {
    int col = wordmapWidth - 1;
    int colOffset = -1;
    for (int rowOffset = -1; rowOffset < 2; rowOffset++) {
      // Get our next text string, and attempt to match!
      text = buildTextString(col, row, 50, colOffset, rowOffset);
      location = BruteForceStringMatch(text, strlen(text), pattern, patternLength);
      if (location >= 0) {
        UpdateFind(row, col, rowOffset, colOffset, location);
      }
      delete(text);
    }
  }
```

```
      printf("%d %d\n", currentRow + 1, currentCol + 1);
      // Free our word.
      delete(pattern);
    }

    // Free our wordmap;
    for (int row = 0; row < wordmapHeight; row++) {
      delete(wordmap[row]);
    }
    delete(wordmap);
    printf("\n");
  }
  return 0;
}
```

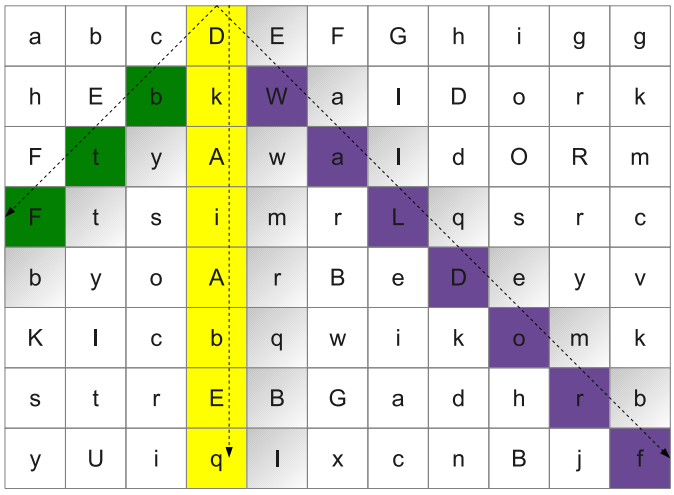
Listing 29: Solution to Where's Waldorf (C++)

The Where's Waldorf problem, can be solved a number of ways, with the two most common methods being:

1. Move through each letter in the grid, and attempt to find the word in one of the eight directions leading from the current character, until a match is found. (This requires $8n^2$ searches or **O(n²)** performance, not including the complexity of the string matching algorithm). An enhancement of this technique would be limit the search to places where the word would fit in the grid, reducing the number of tests.
2. Move along the outer most characters only, and attempt to find the word in three directions that look inward into the grid, utilising a pattern search algorithm. (This requires $12n$ searches or **O(n)** performance, not including the complexity of the string matching algorithm).

For solving this problem, I utilised the second method.

In order to mitigate the case from being an issue, when reading in either a grid line or pattern to look for, I convert both to lower case before storing/using them. The sample solution first initiates a scan along the top of the grid, creating 3 texts to search for each position along the top of the grid. Figure 6.3 demonstrates this technique. If found, I update the best known location (according to cuurent row and column values), and then continue to scan the rest of the grid (as shown as the gradients squares in Figure 6.3). Lastly at the end of the grid scan, I output the best found location.



Three scan lines starting from row 1, column 4. The resulting texts from these will be:

- dbtf
- dkaiabeq
- dwaldorf

The third line when searched will return 1 (being pattern found at index 1), so the found row and column values are updated accordingly.

By only having start positions around the edge of the grid, we don't duplicate text strings, thereby increasing performance of the search.

Figure 6.3: Scanning Grid for Where's Waldorf

For the sample implementation I utilised the Brute Force String Matching Algorithm simply to demonstarte it's use. However, in a real competition environment I would utilise a more efficient pattern

matching algorithm to ensure the best possible performance. One such algorithm that could be used is in place of the Brute Force String Matching Algorithm described in the next section.

Some of the tough test cases include:

- Grids being either 1 wide or 1 row deep.
- Current text to scan is smaller than the pattern length.
- Multiple words found in the grid.
- A grid being all of one letter, and pattern to find is a string of the one letter.

6.2.2 Knuth-Morris-Pratt Substring Search

When studying the worst-case performance of the brute force pattern matching, there is a major deficiency in that characters are potentially rescanned. Specifically, we may perform many comparisons while testing potential placement of the pattern against the text, yet if we discover a pattern character that does not match in the text, then we throw away all the information gained by these comparisons and start over again from scratch with the algorithm.

The Knuth-Morris-Pratt (or KMP) algorithm, avoids this waste of information, and in doing so, achieves **O(n+m)** performance, which is optimal in the worst case.

Algorithm 14 Knuth-Morris-Pratt Substring Search

Input A Vector T of n characters representing a text, and vector P of m characters representing a pattern to find in T.

Output The index of the first character of P when a complete match is found in T, otherwise -1 if search was unsuccessful.

```
procedure KMPMATCH(T, P)
    f ← KMPFAILURE(P)
    i ← 0
    j ← 0
    while i < n do
        if P[j] = T[i] then
            if j = m − 1 then
                return i − m + 1                          ▷ A match!
            end if
            i ← i + 1
            j ← j + 1
        else if j > 0 then                                ▷ No match, but advanced in P
            j ← f(j − 1)
        else
            i ← i + 1
        end if
    end while
    return −1
end procedure
```

6.2.2.1 Description of working The main idea of the KMP algorithm is to preprocess the pattern string P so as to compute a **failure function** *f* that indicates the proper shift of *P* so that, to the largest extent possible, we can reuse previously performed comparisons. Specifically, the failure function *f(j)* is defined as the length of the longest prefix of *P* that is a suffix of *P[1..j]*. We also use the convention that *f(0) = 0*. The importance of this failure function is that "encodes" repeated substrings inside the pattern itself.

The KMP pattern matching algorithm, shown in Algorithm 14 and 15, incrementally processes the text string *T* comparing it to the pattern string *P*. Each time there is a match, we increment the current indices.

Algorithm 15 Knuth-Morris-Pratt Substring Search Failure Function

Input A Vector P of m characters representing a pattern.
Output The failure function f for P, which maps j to the length of the longest prefix of P that is a suffix of $P[1..j]$

```
procedure KMPFAILURE(P)
    i ← 1
    j ← 0
    f(0) ← 0
    while i < m do
        if P[j] = P[i] then                    ▷ We have matched j + 1 characters
            f(i) ← j + 1
            i ← i + 1
            j ← j + 1
        else if j > 0 then                     ▷ j indexes just after a prefix of P that must match
            j ← f(j − 1)
        else                                   ▷ We have no matches
            f(i) ← 0
            i ← i + 1
        end if
    end while
end procedure
```

On the other hand, if there is a mismatch and we have previously made progress in *P*, then we consult the failure function to determine the new index in *P* where we need to continue checking *P* against *T*. Otherwise, we simplate increment the index for *T* (and keep the index variable for *P* at its beginning). We repeat this process until we find a match of *P* in *T* or the index of *T* reaches *n*, the length of *T*.

The main part of the KMP algorithm is the while-loop, which performs a comparison between a character in *T* and a character in *P* each iteration. Depending upon the outcome of this comparison, the algorithm either moves on to the next characters in *T* and *P*, consults the failure function for a new candidate character in *P*, or starts over with the next index in *T*. The correctness of this algorithm follows from the definition of the failure function. The skipped comparisons are actually unnecessary, for the failure function guarantess that all the ignored comparisons are redundan - they would involve comparing characters we already know match.

In Figure 6.4, we illustrate the execution of the KMP algorithm. The failure function f for the pattern displayed is given in Figure 6.5. The algorithm performs 19 character comparisons, which are indicated with numberical labels. Note the use of the failure function to avoid redoing one of the comparisons between a character of the pattern and a character of text. Also note that this algorithm performs fewer overall comparisons tha the brute force algorithm.

The failure function utilised within the KMP Algorithm is also shown in Algorithm 15. This algorithm is another example of a bootstrapping process quite similar to that used in the KMPMATCH procedure. We compare the pattern to itself as in the KMP Algorithm. Each time we have two characters that match, we set $f(i) = j + 1$. Note that since we have $i > j$ throughtout the execution of the algorithm, $f(j - 1)$ is always defined when we need to use it.

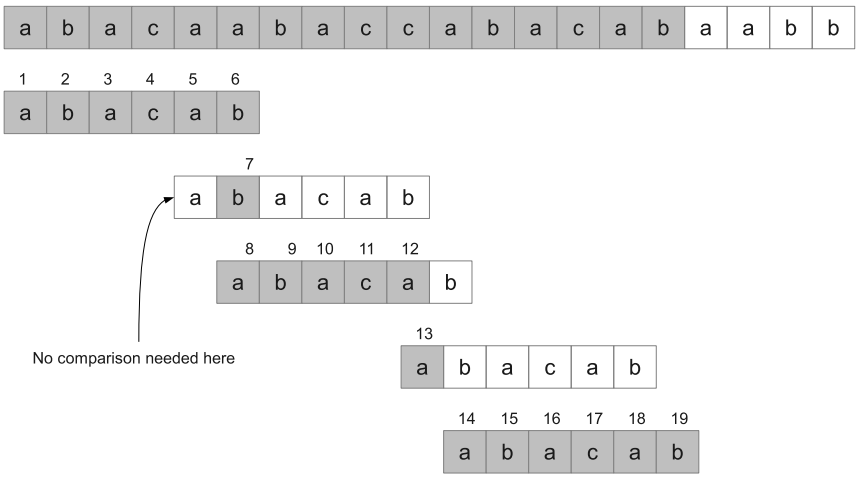


Figure 6.4: Illustration of the KMP algorithm.

Consider the pattern string *P*=”abacab”. The failure function f(j) for the string is shown as the following table.

j	0	1	2	3	4	5
P(j)	a	b	a	c	a	b
f(j)	0	0	1	0	1	2

Figure 6.5: Failure Function for Figure 6.4

6.2.2.2 Implementation An implementation of the KMP Algorithm can be found in Listing 30.

```
/**
 * KMP Failure function.
 * @param Pattern Pointer to the pattern to be processed to derive the failure
 *                function.
 * @param patternLength Length of pattern.
 * @return The failure function (array of int).
 */
int *KMPFailure(char *Pattern, int patternLength) {
  int indexI = 1;
  int indexJ = 0;
  int* failure = new int[patternLength]; // Create new function f.

  failure[0] = 0;
  while (indexI < patternLength) {
    if (Pattern[indexJ] == Pattern[indexI]) {
      failure[indexI++] = ++indexJ;
    } else if (indexJ > 0) {
      indexJ = failure[indexJ - 1];
    } else {
      failure[indexI++] = 0;
    }
  }
  return failure;
}
```

```
/**
 * Utilise the KMP Substring search algorithm to find a pattern in text.
 * @param Text Pointer to the text to be searched.
 * @param textLength Length of the text to be processed.
 * @param Pattern Pointer to the pattern to find within the text
 * @param patternLength Length of the pattern string
 * @return Index of pattern in text, otherwise -1 if not found.
 */
int KMPMatch(char* Text, int textLength, char* Pattern, int patternLength) {
  char* failure = KMPFailure(Pattern, patternLength);
  int indexI = 0;
  int indexJ = 0;
  while (indexI < textLength) {
    if (Pattern[indexJ] == Text[indexI]) {
      if (indexJ == (patternLength - 1)) {
        delete(failure); // Found a match, so return the index.
        return (indexI - patternLength + 1);
      }
      indexI++;
      indexJ++;
    } else if (indexJ > 0) { // No match but advanced in P.
      indexJ = failure[indexJ - 1]; // Move forward appropriate amount.
    } else {
      indexI++;
    }
  }
  delete(failure);
  return -1;
}
```

Listing 30: KMP String Match (C++)

6.2.2.3 Sample Problem - Big String Search Search some text to locate a particular pattern. If the pattern exists in the text, output the 0-based index of the pattern in the text.

INPUT

The first line of input will be *N (1 <= N <= 1000)*, the number of test cases to run. On each of the next *2N* lines will be two strings. The first string will be the pattern to search for. On the next line will be the text to search in. All pattern strings will have a length between 1 and 100,000 inclusive. All text strings will have a length between 1 and 500,000 inclusive.

SAMPLE INPUT

```
5
the
the quick brown fox jumps over the lazy dog
000111
010101010101000110000111
CGAT
TGATCTAGCTAGCTAGCTAGCTAGCATACGCATAGCTA
happy
I could be happy if I didn't have to work
```

OUTPUT

For each test case, output the test case number followed by a space, and then the 0-based index of the first letter of the pattern in the text if the pattern can be found, or NOT FOUND if the pattern does not exist in the text.

SAMPLE OUTPUT

```
1 0
2 18
3 NOT FOUND
4 11
```

6.2.2.4 Sample Solution A sample solution to the Big String Search problem can be found in Listing 31.

```
#include <cstdlib>
#include <cstdio>
#include <cstring>

using namespace std;

#define MAX_PATTERN_LENGTH 100000
#define MAX_TEXT_LENGTH 500000

int testCases = 0;
char pattern[MAX_PATTERN_LENGTH];
char text[MAX_TEXT_LENGTH];

int index = 0;
int testCount = 1;

/**
 * KMP Failure function.
 * @param Pattern Pointer to the pattern to be processed to derive the failure function.
 * @param patternLength Length of pattern.
 * @return The failure function (array of int).
 */
int *KMPFailure(char *Pattern, int patternLength) {
  int indexI = 1;
  int indexJ = 0;
  int* failure = new int[patternLength]; // Create new function f.

  failure[0] = 0;
  while (indexI < patternLength) {
    if (Pattern[indexJ] == Pattern[indexI]) {
      failure[indexI++] = ++indexJ;
    } else if (indexJ > 0) {
      indexJ = failure[indexJ - 1];
    } else {
      failure[indexI++] = 0;
    }
  }
  return failure;
}

/**
 * Utilise the KMP Substring search algorithm to find a pattern in text.
 * @param Text Pointer to the text to be searched.
 * @param textLength Length of the text to be processed.
 * @param Pattern Pointer to the pattern to find within the text
 * @param patternLength Length of the pattern string
 * @return Index of pattern in text, otherwise -1 if not found.
 */
int KMPMatch(char* Text, int textLength, char* Pattern, int patternLength) {
  int* failure = KMPFailure(Pattern, patternLength);
  int indexI = 0;
  int indexJ = 0;
  while (indexI < textLength) {
    if (Pattern[indexJ] == Text[indexI]) {
      if (indexJ == (patternLength - 1)) {
        delete [] failure;
        return (indexI - patternLength + 1);
      }
      indexI++;
      indexJ++;
    } else if (indexJ > 0) { // No match but advanced in P.
      indexJ = failure[indexJ - 1]; // Move forward appropriate amount.
    } else {
```

```
      indexI++;
    }
  }
  delete [] failure;
  return -1;
}

/*
 * Main Function
 */
int main() {
  scanf("%d\n", &testCases);
  while (testCases--) {
    // Read in our pattern and text
    gets(pattern);
    gets(text);

    // found our pattern in text and report the location.
    index = KMPMatch(text, strlen(text), pattern, strlen(pattern));
    if (index == -1) {
      printf("%d␣NOT␣FOUND\n", testCount++);
    } else {
      printf("%d␣%d\n", testCount++, index);
    }
  }
  return 0;
}
```

Listing 31: Solution to Big String Search (C++)

The Big String Problem is a relatively simple problem that relies on the string search algorithm to be correct. As can be seen in Listing 31, within the main() function, while-loop comprises of simply reading in the pattern and text strings, performing the search function and reporting the results of the search.

Sample test cases include:

- Pattern and Text length being 1.
- Pattern length being greater than the text length.

6.3 Graph Theory (Basic)

Graph Theory forms one of the principle areas of modern computer science due to its ability to be applied to a wide range of problems and areas of study.

When viewed abstractly, a **graph** G is simply a set of V of **vertices** and a collection of E of pairs of vertices from V, called **edges**. Thus, a graph is a way of representing connections or relationships between pairs of objects from some set V. Egdes in a graph are either **directed** or **undirected**, that is an edge may only allow connection in a single direction between vertices. Egdes may also have a **weight** applied that defines the cost[22] of utilising the edge for some defined operation.

Using the graph in Figure 6.6 as an example, each location (town/suburb) is a vertex, and the road is an edge between locations. The directions denote if the road is a one-way street or two-way, and the weight is the defined time to travel on that road to get between the vertices.

Using the graph as shown, some examples include:

- It's quicker to travel from Caboolture to Samford Valley via Brisbane, than it is to go via a direct route. (70 vs 90)
- For someone living in Logan, in order to travel to Brisbane, must travel either through Ipswich or the Gold Coast. (via Ipswich offers the shortest time).
- The only way to travel from the Gold Coast to Samford Valley is via Brisbane.

[22] A weight may have either a positive or negative value depending on what is being modelled.

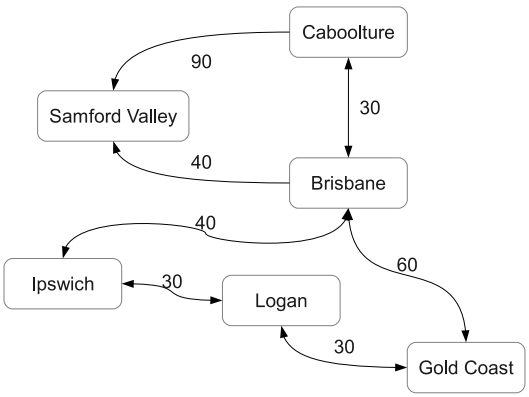


Figure 6.6: Basic Graph

- And if you arrive in Samford Valley, there is no road that leads out!

Other terminology that is often applied to Graph Theory includes:

- A **degree** of *V*, denoted *deg(v)*, is the number of edges incident to *V*. This may also be broken down into in-degree and ou-degree when applied with a directed graph. (The count of the edges leading to or from the vertex respectively).
- A **trail** in a graph, is a sequences of necessiarily distinct vertices, but distinct edges.
- A **path** is a trail in which all vertices are distinct.
- A **circuit** is a trail in which the starting vertex is the same as the ending vertex.
- A **cycle** is a path in which the starting vertex is the same as the ending vertex.
- A **Hamiltonian cycle** is a cycle that contains all the vertices of a graph.
- A **sparse** graph is a graph with few edges, conversely, a **dense** graph is a graph with many edges.

Other terms commonly used within Graph Theory will be introduced later through out this guide as needed, especially when utilised within description of an algorithm.

6.3.1 Representation as adjacency matrix and adjacency list.

Graphs may be represented in two forms, either as an adjacency matrix or as an adjacency list.

6.3.1.1 Adjacency List An Adjacency List is a method of representing a graph in a space efficient form, in that a vector (array) of vertices is maintained, and each element of the vector maintains a list (typically a linked list or ArrayList) of vertices that may be reached from the selected element. For weighted graphs, each element in the list also includes the edge weight in addition to the vertex.

Using the graph in Figure 6.7, the represented Adjacency List is shown in Table 4.

Two advantages of utilising the Adjacency List to represent a graph are:

- The space required to store the list is minimal in that no wasted space is present.
- Travesal of the graph is easy, by simply following the vertex information in the list.

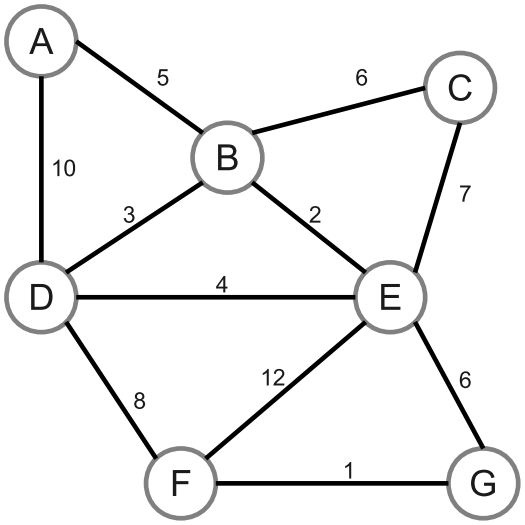


Figure 6.7: Basic Graph Example (weighted, undirected)

Vertex	List
A	B,5 → D,10
B	A,5 → C,6 → D,3 → E,2
C	B,6 → D,7
D	A,10 → B,3 → E,4 → F,8
E	B,2 → C,7 → D,4 → F,12 → G,6
F	D,8 → E,12 → G,1
G	E,6 → F,1

Table 4: Adjacency List

6.3.1.1.1 Description of working The adjacency list works primarily as it stores references and not copies to other vertices within the list, as well as the weights associated with the edge to the vertex. Additionally the data structure naturally supports both unweighted and weighted graphs, as well as directed and undirected graphs within the one implementation.

For graph travesal, you can simply make one of vertices within the list the current location, and therefore find it's edge list as well. This allows for a naturally recursive implementation for many graph traversal algorithms.

The disadvantages of the adjacency list implementation, is that in order to find if either an edge between vertices exist, or any cost associated, the list must be scanned to find the other vertex, which in worst case may be **O(n)**.

6.3.1.1.2 Implementation A storage class implemented in Java for representing an Adjacency List is shown in Listing 32.

```
import java.util.ArrayList;

public class AdjacencyList {

  public ArrayList<ArrayList<edge>> adjacencyList;

  public AdjacencyList(int vertexCount){
```

```
    // Create the arraylist of arraylist of edges.
    adjacencyList = new ArrayList<ArrayList<edge>>();
    for (int n = 0; n < vertexCount; n++) {
      adjacencyList.add(new ArrayList<edge>());
    }
  }

  /**
   * Generic Edge List to store our edge information.
   */
  public class edge implements Comparable<edge> {

    /**
     * destination vertex and weight.
     */
    public int vertex, weight;

    /**
     * Constructor
     *
     * @param v The destination vertex
     * @param w The weight of the edge.
     */
    public edge(int v, int w) {
      vertex = v;
      weight = w;
    }

    /**
     * Compare function for collection sort.
     */
    @Override
    public int compareTo(edge o) {
      if (this.vertex < o.vertex) {
        return -1;
      } else if (this.vertex > o.vertex) {
        return 1;
      } else {
        return 0;
      }
    }
  }
}
```

Listing 32: Adjacency List (Java)

6.3.1.1.3 Sample Problem - Adjacency List Construct the adjacency list representation of an unweighted undirected graph.

INPUT

The first line of input will be a number on a line by itself which is the number of test cases to run. For each test case, the first line will be two numbers separated by a space *N* and *M*, where *N (1 <= N <= 5000)* is the number of nodes in the graph and *M (1 <= M <= 10000)* is the number of edges. The graph nodes will be numbered 0 to *N*-1. Each of the next *M* lines contain 2 numbers *A* and *B* separated by a space representing an edge in the graph between *A (0 <= A < N)* and *B (0 <= B < N)*.

SAMPLE INPUT

```
2
4 4
0 1
1 2
2 0
2 3
4 4
0 3
1 0
2 0
2 3
```

OUTPUT

For each test case, output the test case number on a line followed by the adjacency list representation of the graph. For each node n in the graph output a line listing the number of the node followed by a colon and a space, then a comma and single space separated list of the nodes adjacent to n. The adjacent nodes should be listed in ascending order. See the example output below.

SAMPLE OUTPUT

```
1
0:  1, 2
1:  0, 2
2:  0, 1, 3
3:  2
2
0:  1, 2, 3
1:  0
2:  0, 3
3:  0, 2
```

6.3.1.1.4 Sample Solution A sample solution to the Adjacency List problem can be found in Listing 35.

,

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.Scanner;

public class AdjacencyList {

  static ArrayList<ArrayList<edge>> adjacencyList;
  static ArrayList<HashSet<Integer>> adjacencyEdges;

  /**
   * Main
   */
  public static void main(String[] args) {
    new AdjacencyList().run();
  }

  /**
   * Run interface to be called from main().
   */
  public void run() {
    Scanner in = new Scanner(System.in);

    // get first line and get the number of cases to test.
    int caseCount = Integer.parseInt(in.nextLine());
    int loopCount = 0;
```

```
    while (caseCount - loopCount > 0) {
      String line = in.nextLine();

      // extract numbers, this is the node count and edge count.
      Scanner sc = new Scanner(line);
      int vertexCount = sc.nextInt();
      int edgeCount = sc.nextInt();

      // Create the adjacency List.
      adjacencyList = new ArrayList<ArrayList<edge>>();
      adjacencyEdges = new ArrayList<HashSet<Integer>>();
      for (int n = 0; n < vertexCount; n++) {
        adjacencyList.add(new ArrayList<edge>());
        adjacencyEdges.add(new HashSet<Integer>());
      }

      while (edgeCount-- > 0) {
        // Get our next edge...
        line = in.nextLine();
        // extract numbers, this is the node count and edge count.
        sc = new Scanner(line);
        int source = sc.nextInt();
        int dest = sc.nextInt();
        edge edge1 = new edge(dest, 0);
        edge edge2 = new edge(source, 0);
        if (!adjacencyEdges.get(source).contains(edge1.hashCode())) {
          adjacencyEdges.get(source).add(edge1.hashCode());
          adjacencyList.get(source).add(edge1);
        }
        if (!adjacencyEdges.get(dest).contains(edge2.hashCode())) {
          adjacencyEdges.get(dest).add(edge2.hashCode());
          adjacencyList.get(dest).add(edge2);
        }
      }

      System.out.printf("%d\n", ++loopCount);
      // Print the resulting adjacency list.
      for (int i = 0; i < vertexCount; i++) {
        // Print the vertex number.
        System.out.printf("%d:␣", i);
        ArrayList<edge> edges = adjacencyList.get(i);
        // Sort the array list.
        Collections.sort(edges);
        // Print the contents of the array list.
        for (int node = 0; node < edges.size(); node++) {
          if (node + 1 == edges.size()) {
            System.out.print(edges.get(node).vertex);
          } else {
            System.out.printf("%d,␣", edges.get(node).vertex);
          }
        }
        System.out.println();
      }
    }
}

/**
 * Generic Edge List to store our edge information.
 */
public class edge implements Comparable<edge> {

  /**
   * destination vertex and weight.
```

```
     */
    public int vertex, weight;

    /**
     * Constructor
     *
     * @param v The destination vertex
     * @param w The weight of the edge.
     */
    public edge(int v, int w) {
      vertex = v;
      weight = w;
    }

    /**
     * Compare function for collection sort.
     */
    @Override
    public int compareTo(edge o) {
      if (this.vertex < o.vertex) {
        return -1;
      } else if (this.vertex > o.vertex) {
        return 1;
      } else {
        return 0;
      }
    }

    @Override
    public int hashCode() {
      int hash = 1;
      hash = 100007 * hash + this.vertex;
      hash = 100007 * hash + this.weight;
      return hash;
    }
  }
}
```

Listing 33: Solution to Adjacency List Problem (Java)

The Adjacency List Problem is a relatively simple problem that relies on the data structure implementation to be correct. As can be seen in Listing 35, within the run() method, while-loop comprises of simply reading in the vertex information for each graph, building a list in memory. The only main complication is the list being printed in an ascending manner, which is solved by sorting it with `Collections.sort()`[23], and printing the result.

Sample test cases include:

- Edge count for the entire graph being zero.
- The degree of an vertex being zero.
- Receiving the same edge more than once in the input strings.

6.3.1.2 Adjacency Matrix An Adjacency Matrix is a method of representing a graph within a two-dimension matrix which offers **O(1)** lookout performance to determine if either an edge exists, or obtain the weight of an edge.

Using the graph in Figure 6.7, the represented Adjacency Matrix is shown in Table 5.

[23] Collections.sort() utilises a merge sort, offering **O(n log n)** performance for sorting.

Vertex	A	B	C	D	E	F	G
A	0	5	0	10	0	0	0
B	5	0	6	3	2	0	0
C	0	6	0	0	7	0	0
D	10	3	0	0	4	8	0
E	0	2	7	4	0	12	6
F	0	0	0	8	12	0	1
G	0	0	0	0	6	1	0

Table 5: Adjacency Matrix

6.3.1.2.1 Description of working The Adjacency Matrix works well with either dense graphs (as the storage requirement between an Adjancey Matrix and Adjacency List would be roughly equal), or when performance is required when either utilising edge information or for performing edge manipulation operations.

The Adjacency Matrix utilises an $n \times n$ array A to store the edge information, such that $A[i, j]$ stores a reference to edge (i, j), if such an edge exists. If there is no edge (i, j), then $A[i, j]$ is null, with null typically being represented by 0 (zero). For weighted graphs each cell, holds the weight of the edge, and for un-weighted graphs, each cell will hold a boolean value either `true` (or 1) for an edge, or `false` (0) for no edge.

Using Adjacency Matrix A, we can perform edge lookup in **O(1)** time, as each lookup is the same as accessing $A[i, j]$. However, the the storage requirements for an Adjacency Matrix is **O(n^2)** which may representing very large graphs difficult.

6.3.1.2.2 Implementation A storage class implemented in Java for representing an Adjacency Matrix is shown in Listing 34. However it should be noted, that typically within the competition, a simple two dimension array of the corrrect type is used instead of the generic class implementation shown.

```
public class AdjacencyMatrix<E> {

  private E[][] matrix;
  private boolean directedGraph = false;

  /**
   * Create a new adjacency matrix of size n.
   *
   * @param numberOfVertex The number of vertices in the graph.
   * @param directedGraph If the graph is directed or undirected.
   */
  public void AdjacencyMatrix(int numberOfVertex, boolean directedGraph) {
    // Create a matrix
    matrix = (E[][]) new Object[numberOfVertex][numberOfVertex];
    // Set all egdes to 0.
    for (int i = 0; i < numberOfVertex; i++) {
      for (int j = 0; j < numberOfVertex; j++) {
        setEdge(i, j, null);
      }
    }
    // Set if graph is directed.
    this.directedGraph = directedGraph;
  }

  /**
   * Find if the specified edge exists.
   *
   * @param i Source vertex
   * @param j Destination vertex
   * @return true, if an edge is present otherwise false.
   */
```

```
  public boolean isEdge(int i, int j) {
    return (matrix[i][j] != null);
  }

  /**
   * Set the weight of an edge.
   *
   * @param i Source vertex
   * @param j Destination vertex
   * @param weight The new weight to be applied to this edge. (May be zero to
   * remove the edge from the graph).
   */
  public void setEdge(int i, int j, E weight) {
    matrix[i][j] = weight;
    if (!directedGraph) {
      // If undirected graph then set the back edge as well.
      matrix[j][i] = weight;
    }
  }

  /**
   * Get the weight of an edge.
   *
   * @param i Source vertex
   * @param j Destination vertex
   * @return The current weight of an edge, or null if no edge.
   */
  public E getEdge(int i, int j) {
    return matrix[i][j];
  }

  /**
   * Retrieve the matrix as a string suitable for printing.
   * @return String representation of the matrix.
   */
  @Override
  public String toString(){
    return matrix.toString();
  }
}
```

Listing 34: Adjacency Matrix (Java)

6.3.1.2.3 Sample Problem - Adjacency Matrix Construct the adjacency matrix representation of a weighted directed graph.

INPUT

The first line of input will be a number on a line by itself which is the number of test cases to run. For each test case, the first line will be two numbers separated by a space *N* and *M*, where *N (1 <= N <= 5000)* is the number of nodes in the graph and *M (1 <= M <= 10000)* is the number of edges. The graph nodes will be numbered 0 to *N*-1. Each of the next *M* lines contain 3 numbers *A B* and *C* each separated by a space representing an edge in the graph from *A (0 <= A < N)* to *B (0 <= B < N)* with weight *C (0 < C <= 1000)*.

SAMPLE INPUT

```
2
4 4
0 1 3
1 2 6
2 0 2
2 3 4
4 4
0 3 10
1 0 100
2 0 1000
2 3 2
```

OUTPUT

For each test case, output the test case number on a line followed by the adjacency matrix representation of the graph. The graph node numbers should appear as headings on the columns and rows of the matrix and the edge weights should appear as the contents of the matrix. Each number should be printed in a field width of 4 with a single space between all headings and values. See the example output below.

SAMPLE OUTPUT

,

```
1
         0    1    2    3
   0     0    3    0    0
   1     0    0    6    0
   2     2    0    0    4
   3     0    0    0    0
2
         0    1    2    3
   0     0    0    0   10
   1   100    0    0    0
   2  1000    0    0    2
   3     0    0    0    0
```

6.3.1.2.4 Sample Solution A sample solution to the Adjacency Matrix problem can be found in Listing 35.

```
import java.util.Scanner;

public class AdjacencyMatrixProblem {

  public static void main(String[] args) {
    int[][] adjacencyMatrix;

    Scanner in = new Scanner(System.in);

    // get first line and get the number of cases to test.
    int caseCount = Integer.parseInt(in.nextLine());
    int loopCount = 0;
    while (caseCount - loopCount > 0) {
      String line = in.nextLine();

      // extract numbers, this is the node count and edge count.
      Scanner sc = new Scanner(line);
      int vertexCount = sc.nextInt();
      int edgeCount = sc.nextInt();

      // Create the matrix, and clear the contents.
      adjacencyMatrix = new int[vertexCount][vertexCount];
      for (int i = 0; i < vertexCount; i++) {
        for (int j = 0; j < vertexCount; j++) {
          adjacencyMatrix[i][j] = 0;
        }
      }

      // Read in all the edges.
      while (edgeCount-- > 0) {
        // Get our next edge...
        line = in.nextLine();
        // extract numbers, this is the node count and edge count.
        sc = new Scanner(line);
        int source = sc.nextInt();
        int destination = sc.nextInt();
        int weight = sc.nextInt();
        adjacencyMatrix[source][destination] = weight;
      }
```

```
      // Print out the matrix.
      System.out.printf("%d\n␣␣␣␣", ++loopCount);
      // Print the row header.
      for(int i = 0; i < vertexCount; i++){
        System.out.printf("%5d", i);
      }
      System.out.println();
      // Print the matrix contents
      for (int i = 0; i < vertexCount; i++) {
        System.out.printf("%4d", i); // Row number
        for (int j = 0; j < vertexCount; j++) {
          System.out.printf("%5d",adjacencyMatrix[i][j]); // Cell
        }
        System.out.println();
      }
    }
  }
}
```

Listing 35: Solution to Adjacency Matrix Problem (Java)

The Adjacency Matrix Problem is a relatively simple problem that relies on the data structure implementation to be correct. Since the underlying data structure is a simple array, there is little that can go wrong.

The main challenges include:

- A graph with zero nodes.
- A graph with zero edges.
- Ensuring correct spacing when displaying the matrix.

6.3.2 Graph Traversal

A common operation on a graph is to visit every vertex and perform an operation on it. This can be performed systematically through either a Breadth First Search (BFS) or a Depth First Search (DFS). Both DFS and BFS will visit all vertices in a graph, with the difference being in the order in which they visit each vertex.

Breadth First Search, after visting a given vertex n, will visit every unvisited vertex adjacent to n before visiting any other nodes. Depth First Search, from a given vertex n, will proceed along a path from n as deeply into the graph as possible before backtracking. With both algorithms, at any point we need to know:

- Which nodes we have visited.
- Which nodes we have found but need to visit.

Essentially the same algorithm is used for both BFS and DFS, however the datastructures to store the above knowledge points allow the algorithm to work differently. A BFS will use a queue datastructure, whilst DFS will use a stack datastructure to implement those knowledge points. The similarities can be seen comparing both Algorithm 16 and Algorithm 17.

6.3.2.1 Description of working The Depth First Search is considered an adventurous algorithm, in that it simple wanders the graph from vertex to vertex until it reaches a vertex in which it has no unexplored vertices adjacent to the current vertex. Once it reaches this point, it will backtrack until it finds a vertex which has unexplored neighbours and then continue on from this point, until all vertices have been visited.

Figure 6.8 displays an example of the DFS travesal, where the edges incident of a vertex are explored by alphabetical order of the adjacent vertices, starting at Vertex A. (The direction of travel is displayed by

Algorithm 16 Breadth First Search

Input A Graph G and a vertex s.
Output A vertices are visited in a BFS fashion

```
procedure BFS(G, s)
    L ← QUEUE()
    L.enqueue ← s                                   ▷ enqueue s
    while L is not empty do
        v ← L.dequeue                               ▷ dequeue v
        visit v and mark as visited                 ▷ perform operation on v
        for all node n adjacent to v do
            if n is not visited and not on queue then
                L.enqueue ← n                       ▷ enqeue n
            end if
        end for
    end while
end procedure
```

Algorithm 17 Depth First Search

Input A Graph G and a vertex s.
Output A vertices are visited in a DFS fashion

```
procedure DFS(G, s)
    L ← STACK()
    L.push ← s                                      ▷ push s
    while L is not empty do
        v ← L.pop                                   ▷ pop v
        visit v and mark as visited                 ▷ perform operation on v
        for all node n adjacent to v do
            if n is not visited and not on stack then
                L.push ← n                          ▷ push n
            end if
        end for
    end while
end procedure
```

the arrows. In this example, no backtracking has taken place). The order of travesal in this instance is: A, B, C, E, D, F, G.

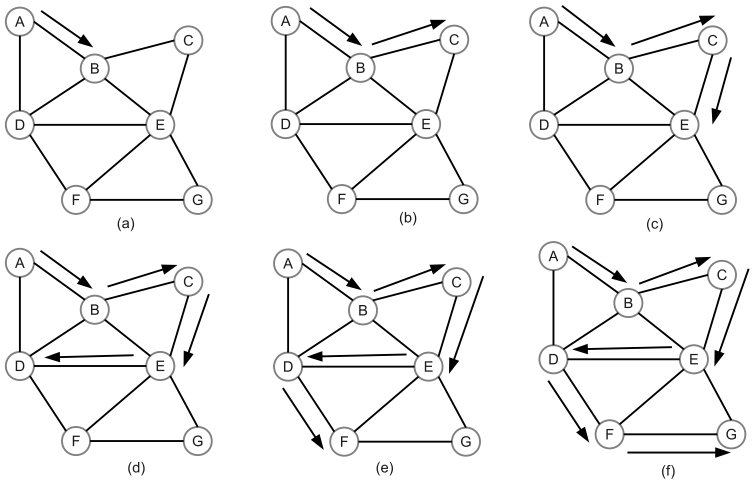


Figure 6.8: DFS Example

The Breadth First Search is less adventurous than DFS, however instead of wandering the graph, BFS proceeds in rounds and subdivides the vertices into levels. Breadth First Search, after visting a given vertex *n*, will visit every unvisited vertex adjacent to *n* before visiting any other nodes. For an unweighted graph, BFS therefore provides the shortests paths from the root vertex to all other vertices. BFS is also the basic implementation for a number of shortest paths algorithms.

Figure 6.9 displays an example of the BFS traversal, where the edges incident of a vertex are explored by alphabetical order of the adjacent vertices, starting at Vertex A. (The discovery edges are shown with bold lines and the crossed edges with a dash line: (a) graph before the traversal; (b) discovery at level 1; (c) discovery at level 2; (d) discovery at level 3.). The order of travesal in this instance is: A, B, D, C, E, F, G.

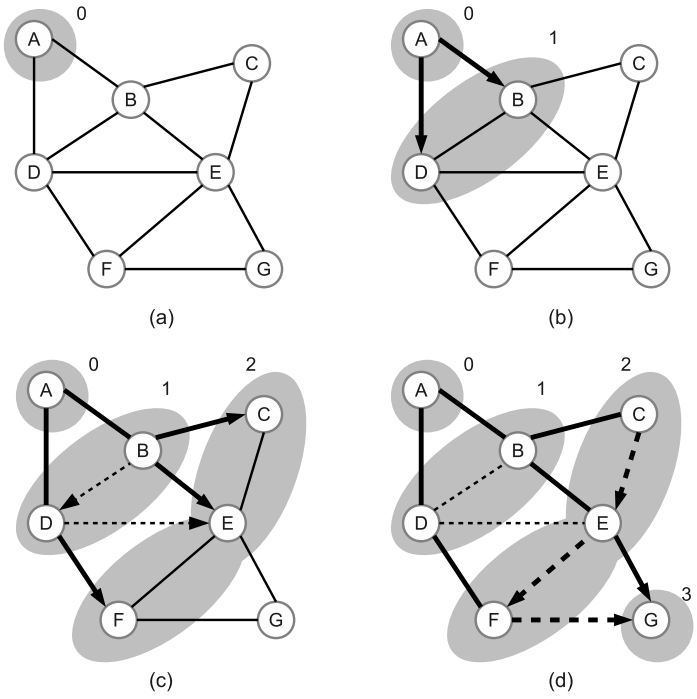


Figure 6.9: BFS Example

On comparing both implementations, it is considered that the DFS travesal is better for answering complex connectivity questions, such as determining if every pair of vertices in a graph can be connected by two disjoint paths. Whilst BFS is better for finding shortest path between a root vertex and a destination vertex.

6.3.2.2 Implementation An implementation of the DFS algorithm in Java can be seen in Listing 36.

```
import java.util.ArrayList;
import java.util.LinkedList;

public class BFS {

  /**
   * Traverse a graph (as represented by an Adjacency Matrix) starting at
   * root, printing the Vertex ID to console.
   * @param matrix The Adjacency Matrix representing the Graph
   * @param root The starting vertex
   */
  public void BFSTravese(int[][] matrix, int root){
    // Create a queue
```

```
    LinkedList<Integer> queue = new LinkedList<Integer>();
    // Create a visited list.
    ArrayList<Integer> visited = new ArrayList<Integer>();

    // Insert the start vertex into the queue.
    queue.offer(root);

    // While the queue is not empty
    while(!queue.isEmpty()){
      int node = queue.poll();
      visited.add(node);
      System.out.printf("%d ", node); // Print the node ID to console.

      for(int i = 0; i < matrix.length; i++){
        // If node not in queue or in visited add to queue.
        if(matrix[node][i] != 0){ // Make sure we are an edge
          if(!queue.contains(i) && !visited.contains(i)){
            queue.offer(i);
          }
        }
      }
    }
  }
}
```

Listing 36: Breadth Search First (Java)

An implementation of the Depth First Search can be seen in Listing 37.

```
import java.util.ArrayList;
import java.util.Stack;

public class DFS {

  /**
   * Traverse a graph (as represented by an Adjacency Matrix) starting at
   * root, printing the Vertex ID to console.
   * @param matrix The Adjacency Matrix representing the Graph
   * @param root The starting vertex
   */
  public void DFSTravese(int[][] matrix, int root){
    // Create a queue
    Stack<Integer> stack = new Stack<Integer>();
    // Create a visited list.
    ArrayList<Integer> visited = new ArrayList<Integer>();

    // Insert the start vertex into the queue.
    stack.push(root);

    // While the queue is not empty
    while(!stack.isEmpty()){
      int node = stack.pop();
      visited.add(node);
      System.out.printf("%d ", node); // Print the node ID to console.

      for(int i = 0; i < matrix.length; i++){
        // If node not in queue or in visited add to queue.
        if(matrix[node][i] != 0){ // Make sure we are an edge
          if(!stack.contains(i) && !visited.contains(i)){
            stack.push(i);
          }
        }
      }
```

```
    }
  }
}
```

Listing 37: Depth Search First (Java)

6.3.2.3 Sample Problem - Monks The order of Avidly Calculating Monks have their ashram high in the Sierra Nevada mountains east of Silicon Valley. After many years of training, each novice is faced with a final test before he can become a full member of the ACM.

In this test, the novice is led into a room containing three large urns, each containing a number of delicate glass beads. His task is to completely empty one of the urns using the following special procedure. Each day, the novice must select two of the urns, the source, and the destination. He must then carefully move beads from the source urn to the destination urn, never breaking one, until the original contents of the destination urn are doubled. No other beads may be moved.

So for example, if the number of beads in the three urns were respectively

115, 200 and 256 beads

then the novice might choose the second urn as the source and the first as the destination which would result in new contents of

230, 85 and 256 beads

at the end of the first day. Then on the second day the novice might choose the urn with 256 beads as the source, and that with 85 as the destination leaving

230, 170 and 171 beads

at the end of the second day. If the original contents were 12, 30, and 12 beads, then choosing the first as source and the third as destination would result in 0, 30, and 24 beads and completion of the task.

The chief guru of the ACM always likes to have available a crib sheet indicating how many days the novices should take if they use as few transfers as possible to empty one of the urns. Your task is to provide this crib sheet.

Given a sequence of scenarios, each with a triple of non-negative integers representing the urn contents, determine for each scenario the smallest possible number of days required to empty one of the urns. This problem is guaranteed to have a solution in each case.

INPUT

Input will consist of a sequence of lines, each line representing a scenario. Each line consists of three integers in the range from 0 to 500 representing the initial urn contents, separated by single spaces. Input is terminated by the line "`0 0 0`", which is not processed.

SAMPLE INPUT

```
0 3 5
5 0 3
1 1 1
2 3 4
12 3 8
0 0 0
```

OUTPUT

For each scenario "`a b c`", output a line of the form

```
a b c d
```

where `d` is the smallest possible number of days required to empty an urn beginning from contents `a`, `b`, and `c`.

SAMPLE OUTPUT

```
0 3 5 0
5 0 3 0
1 1 1 1
2 3 4 2
12 3 8 5
```

6.3.2.4 Sample Solution

A sample solution to the Monks problem can be found in Listing 38.

```
import java.util.Arrays;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Scanner;

/**
 * Jar Class to hold an instance of a set of Jars.
 */
class Jars {

  public final static int A = 0;
  public final static int B = 1;
  public final static int C = 2;
  private int[] jars = {0, 0, 0};
  public int count;

  @Override
  public int hashCode() {
    int hash = 1;
    hash = 503 * hash + this.jars[0];
    hash = 503 * hash + this.jars[1];
    hash = 503 * hash + this.jars[2];
    return hash;
  }

  @Override
  public boolean equals(Object obj) {
    final Jars other = (Jars) obj;
    return ((this.jars[A] == other.jars[A])
            && (this.jars[B] == other.jars[B])
            && (this.jars[C] == other.jars[C]));
  }

  /**
   * Constructor
   *
   * @param JarA the number of beads in Jar A.
   * @param JarB the number of beads in Jar B.
   * @param JarC the number of beads in Jar C.
   */
  public Jars(int JarA, int JarB, int JarC) {
    jars[A] = JarA;
    jars[B] = JarB;
    jars[C] = JarC;
    count = 0;
    Sort();
  }

  /**
   * Copy Constructor
   *
   * @param other The Jars to copy.
   */
  public Jars(Jars other) {
    this.jars[A] = other.jars[A];
    this.jars[B] = other.jars[B];
    this.jars[C] = other.jars[C];
    this.count = other.count;
    Sort();
  }

  /**
   * Sort the contents of the jars into numerical order. Thus: a < b < c
```

```
   */
  public final void Sort() {
    int tmp;
    if (jars[B] < jars[A]) {
      tmp = jars[B];
      jars[B] = jars[A];
      jars[A] = tmp;
    }
    if (jars[C] < jars[B]) {
      tmp = jars[C];
      jars[C] = jars[B];
      jars[B] = tmp;
    }
    if (jars[B] < jars[A]) {
      tmp = jars[B];
      jars[B] = jars[A];
      jars[B] = tmp;
    }
  }

  /**
   * Get if there is an empty Jar within the set.
   */
  public boolean EmptyJar() {
    return ((jars[A] == 0) || (jars[B] == 0) || (jars[C] == 0));
  }

  /**
   * Move beads from one jar to another.
   *
   * @param source The jar to take beads from.
   * @param destination The jar to add beads to.
   */
  public void moveBeads(int source, int destination) {
    jars[source] -= jars[destination];
    jars[destination] += jars[destination];
    Sort(); // The resulting jar counts MUST be sorted.
    count++; // Increment the move count.
  }
}

public class Monks {

  /**
   * Main.
   */
  public static void main(String[] args) {

    // Read in lines as needed.
    Scanner in = new Scanner(System.in);
    String line = in.nextLine();
    // Quit when our input is "0 0 0"
    while (!line.equals("0␣0␣0")) {

      Scanner sc = new Scanner(line);
      // Output the minimum moves required.
      System.out.printf("%s␣%d\n", line,
              LeastMoves(new Jars(sc.nextInt(), sc.nextInt(), sc.nextInt())));
      line = in.nextLine(); // Get next line.
    }
  }

  /**
   * Find the minimum number of moves required to move beads until at least 1
   * jar is empty. This function is a modified BFS algorithm, with each vertex
   * to search next being derived from the current vertex contents.
   *
   * @param jars The starting jar contents.
   * @return The minimum number of moves.
   */
  static int LeastMoves(Jars jars) {
    // Check for empty jar
    if (jars.EmptyJar()) {
```

```
      return 0; // If our start set of jars has an empty jar, then exit.
    } else {
      int leastMoves = 0; // result
      LinkedList<Jars> moves = new LinkedList<Jars>(); // create queue
      HashSet<Integer> visited = new HashSet<Integer>();
      moves.offer(jars); // add initial jars state to queue

      while ((!moves.isEmpty()) && (leastMoves == 0)) {
        Jars current = new Jars(moves.poll()); // remove first in queue
        visited.add(current.hashCode());

        // Process the current set of Jars, first check for empty jar
        if (current.EmptyJar()) {
          leastMoves = current.count; // store result

        } else {

          // Visit all neighbouring vertices (or Jars derived from the
          // current Jar).
          // Since all Jars are stored in numerical order (a<b<c), we
          // can reduce the number of tests to perform, and this also
          // ensures that any subtraction doesn't result in a negative
          // bead count.
          // We maintain a hash set with all visited states to ensure
          // don't revisit them in future
          Jars moveBtoA = new Jars(current);
          moveBtoA.moveBeads(Jars.B, Jars.A);
          int hash = moveBtoA.hashCode();
          if (!visited.contains(hash)) {
            moves.offer(moveBtoA);
            visited.add(hash);
          }

          Jars moveCtoA = new Jars(current);
          moveCtoA.moveBeads(Jars.C, Jars.A);
          hash = moveCtoA.hashCode();
          if (!visited.contains(hash)) {
            moves.offer(moveCtoA); // add result to queue
            visited.add(hash);
          }

          current.moveBeads(Jars.C, Jars.B);
          hash = current.hashCode();
          if (!visited.contains(hash)) {
            moves.offer(current); // add result to queue
            visited.add(hash);
          }
        }
      }
      // We have found a empty jar set
      return leastMoves;
    }
  }
}
```

Listing 38: Solution to Monks Problem (Java)

The use of the BFS algorithm highlights the breadth of the application of the algorithm. Figure 6.10 demonstrates the core method LeastMoves() of the sample solution. You will notice immediately that the resulting calculations that define new jar values form a tree (which is a type of grah), that allows us to the BFS to solve the problems as presented. The key difference, is that new vertices (or Jars) to explore are derived at runtime, and not part of a set graph to explore. This allows the solution to scale to very large graphs of possible jar combinations.

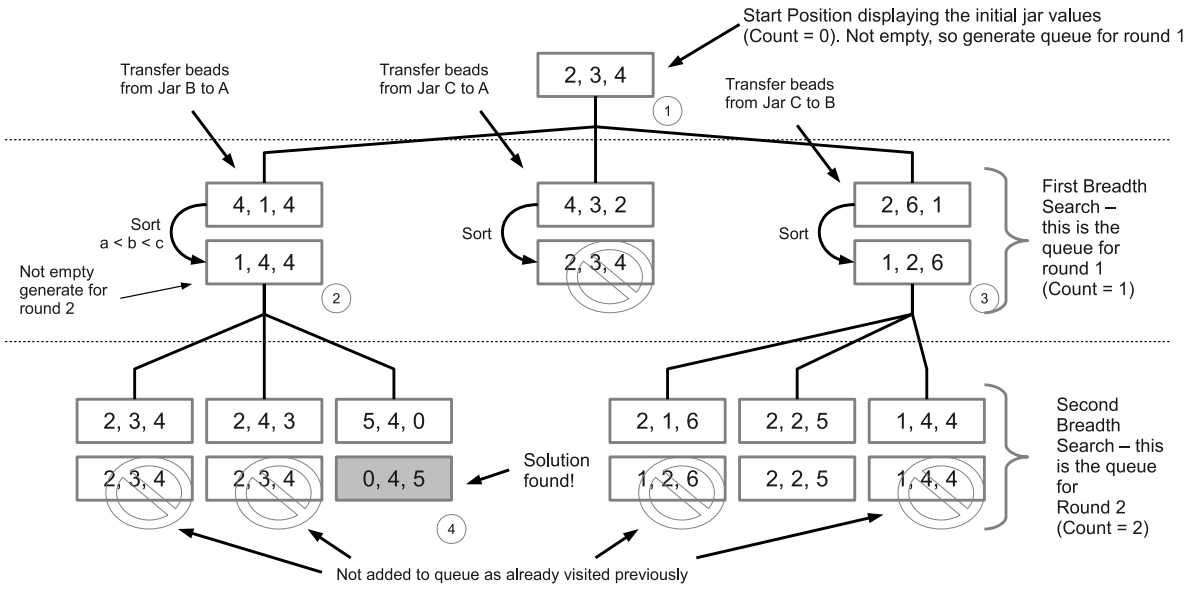


Figure 6.10: Monks Solution Diagram

At Step 1, there are no empty jars, so add the three resulting calculations for the new jar values to the queue. At Step 2, take this jar from the queue, inspect, and since there are no empty jars, add the three resulting calculations for the new jar values to the queue, if and only if the new values to be tested have not already been tested (and found in the `visited HashSet`). The process repeats for steps 3. Since Step 4 has an empty jar we terminate algorithm and return the current count value.

There are few challenging cases within the Monks problem. Ensuring that the jar bead counts are stored in numerical order allows a reduction in the number of edges to explore, as well as removing the requirement to ensure a subtraction from one jar does not result in a negative bead count within a jar. The main tricky case, is that when the starting bead count is such that there is no solution, that even after an infinite number of moves no solution is found. (The problem statement indicates that this is not an issue with the supplied sets to test, but the implementation still handles this). The implementation handles the possibility of a no-solution problem be ensuring that we only test jar states that have not been tested prior, and if run out of states to test, then there is no solution to the problem.

One of the main points within the implementation is the use of the HashSet to hold the visited jar states, as the `contains()` method as implemented within `HashSet` offers **O(1)** lookup. If a normal `List` was used, this would result is significantly reduced performance as the `contains()` method is typically **O(n)** performance.

6.3.3 Minimum Spanning Tree

Minimum spanning trees are a useful operation on a graph that allow you to construct a tree[24] that only includes the minimum number of egdes that also has the property that the sum of all edge weights is the minimum possible.

This is useful for when we wish to find the quickest or cheapest way to implement a network (be it Local Area Network, or Wide Area Network), when we wish to minimise the amount of network cabling needed to connect all computers in a location, etc.

For some graphs there may be multiple minimum spanning trees possible[25], which may be derived differently based on which algorithm is used to derive the tree. The two common algorithms utilised today are Kruskal's Algorithm and The Prim-Jarník Algorithm, whilst the original MST algorithm Barůvka's Algorithm is not utilised as much due to lesser performance compared to the other two.

[24] A tree is a graph, where there are no cycles

[25] This is true for unweighted graphs, or for graphs were most edges share the same weight

6.3.3.1 The Prim-Jarník Algorithm For this section I will describe and implement The Prim-Jarník Algorithm, as it offers good performance while remaining relatively simple to implement.

The Prim-Jarník Algorithm works by walking the graph, following the the edges with the least weight until all vertices have been visited. The full algorithm can be found in Algorithm 18.

Algorithm 18 The Prim-Jarník Algorithm

Input A Graph G which is a fully connected graph.
Output A tree T whose properties match those of a Minimum Spanning Tree.

```
procedure PRIMJARNÍKMST(G)
    Pick any vertex v of G
    D[v] ← 0                                  ▷ D[u] stores the weight of the best current edge
    for all vertex u ≠ v do
        D[u] ← +∞
    end for
    Initialise T ← ∅
    Initialise Priority Queue Q with an item ((u,null, D[u]) for each vertex u.
    while Q is not empty do
        (u, e) ← Q.removeMin()
        Add vertex u and edge e to T.
        for all vertex z adjacent to u such that z is in Q do
            if w((u, z)) < D[z] then
                D[z] ← w((u, z))
                Change to (z, (u, z)) the element of vertex z in Q
                Change to D[z] the key of vertex z in Q
            end if
        end for
    end while
    return T
end procedure
```

6.3.3.1.1 Description of working The algorithm essentially builds the minimum spanning tree based on the least weight of edges that lead to vertices that are not part of the tree.

This is achieved through the use of a priority queue, that holds the next least cost edge of all current vertices within the tree. On each iteration, it will remove the next least cost edge, add it to the tree, then add any new edges to the priority queue for processing. The algorithm terminates when all vertices are present in the tree.

Figure 6.11demonstrates the algorithm, based on the initial graph in Figure 6.7, starting at vertex D.

Starting at Vertex D, all edges that are incident are added to the priority queue, and the edge with the least weight is removed from the queue. At step (a), this is the edge (d,b). This edge is then added to the resulting MST, as shown in the bold line. This process is then repeated at Vertex B (step b), in which all edges are added to the queue (or if in the queue is adjusted if a new lower weight has been found). The edge with the least weight is then removed from the queue. This edge may be adjacent to either Vertex D or Vertex B, as the algorithm only cares about the minimum weight, not the current vertex. This process is repeated, until all vertices are within the MST T (step (f)).

The Prim-Jarník Algorithm depending on the implemetation will have a running time of **O(m log n)** when an adjacency list is utilised or **O(n^2)** when an adjacency matrix is utilised.

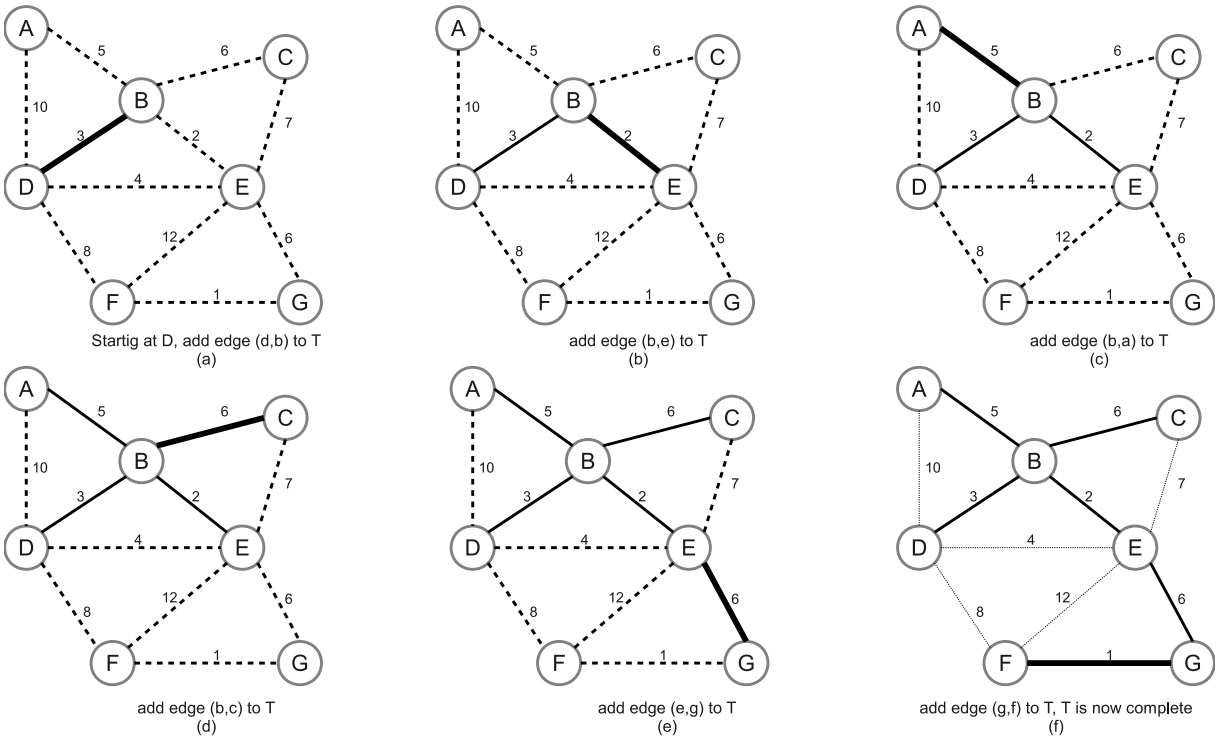


Figure 6.11: The Prim-Jarník Algorithm

6.3.3.1.2 Implementation An implementation of The Prim-Jarník Algorithm can be seen in Listing 39.

```
import java.util.ArrayList;

public class PrimMST {

  /**
   * Construct a MST based on the Prim-Jarnik Algorithm. As this implementation
   * is based on an Adjacency Matrix, offers O(n^2) performance.
   *
   * @param matrix An adjacency matrix defining a graph
   * @param root The starting vertex
   * @return An array with the connecting vertex ID in each element
   */
  public static int[] PrimJarnikMST(int[][] matrix, int root) {
    // shortest known distance to MST, typically this may be a priority
    // queue, but here we use an array, and linear scan the array to find
    // the lowest value.
    int[] distance = new int[matrix.length];

    // preceeding vertex in tree
    int[] pred = new int[matrix.length];

    boolean[] visited = new boolean[matrix.length]; // all false initially

    // Set all distances to inf, except root vertex, which we set to 0,
    // so that the main loop starts at this vertex.
    for (int i = 0; i < distance.length; i++) {
      distance[i] = Integer.MAX_VALUE;
    }
    distance[root] = 0;

    // Main loop, which terminates when all vertices have been visited.
    for (int i = 0; i < distance.length; i++) {
      int next = minVertex(distance, visited); // Get next lowest cost edge
      visited[next] = true;

      // The edge from pred[next] to next is in the MST (if next != root)
```

```
      int[] neighbours = getNeighbours(matrix,next);
      for (int j = 0; j < neighbours.length; j++) {
        int vertex = neighbours[j];
        int weight = matrix[next][vertex];
        if (distance[vertex] > weight || distance[vertex] == 0) {
          distance[vertex] = weight; // Update best weight
          pred[vertex] = next; // Set best vertex to reach this vertex
        }
      }
    }
    return pred;
  }

  /**
   * Find the minimum distance in the current best distance vector
   *
   * @param dist The vector of current minimum distance.
   * @param v An array of vertices that have been visited.
   * @return The shortest known distance to an unvisited vertex.
   */
  private static int minVertex(int[] distance, boolean[] visited) {
    int weight = Integer.MAX_VALUE;
    int vertex = -1;   // graph not connected, or no unvisited vertices
    for (int i = 0; i < distance.length; i++) {
      if (!visited[i] && distance[i] < weight) {
        vertex = i; // set lowest weight vertex.
        weight = distance[i]; // get the weight, for later comparison
      }
    }
    return vertex;
  }

  private static int[] getNeighbours(int[][] matrix, int row){
    ArrayList<Integer> neighbours = new ArrayList<Integer>();
    for(int i = 0; i < matrix.length; i++){
      if(matrix[row][i] > 0){
        neighbours.add(i);
      }
    }
    int[] temp = new int[neighbours.size()];
    int i = 0;
    for(Integer vertex: neighbours){
      temp[i++] = vertex;
    }
    return temp;
  }
}
```

Listing 39: Prim-Jarník Algorithm (Java)

6.3.3.1.3 Sample Problem - Minimum Spanning Tree Construct the minimum spanning tree for a weighted undirected graph.

INPUT

The first line of input will be a number on a line by itself which is the number of test cases to run. For each test case, the first line will be two numbers separated by a space N and E, where N ($1 <= N <= 5000$) is the number of nodes in the graph and E ($1 <= E <= 10000$) is the number of edges. The graph nodes will be numbered 0 to N-1. Each of the next E lines contain three numbers S, D and W each separated by a space representing an edge in the graph from S ($0 <= S < N$) to D ($0 <= D < N$) with weight W ($0 < W <= 1000$).

SAMPLE INPUT

```
2
5 7
0 1 10
1 3 2
1 4 6
3 4 3
2 4 6
0 4 20
0 2 30
5 6
0 1 8
0 3 2
0 4 5
1 2 6
2 3 4
3 4 3
```

OUTPUT

For each test case, output the following message on a new line:

`Test x, minimum spanning tree weight = y`

where `x` is the test case number and y is the total weight of the edges of the minimum spanning tree. See the example output below.

SAMPLE OUTPUT

```
Test 1, minimum spanning tree weight = 21
Test 2, minimum spanning tree weight = 15
```

6.3.3.1.4 Sample Solution A sample solution to the Minimum Spanning Tree problem can be found in Listing 40.

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Scanner;

public class SpanningTreeProblem {

  static int vertexCount;
  static int edgeCount;
  static PriorityQueue<edge> edges;
  static ArrayList<ArrayList<edge>> adjacencyList;
  static Boolean[] visited;
  private final static String OUTPUT_STRING = "Test␣%d,␣minimum␣spanning␣tree␣weight␣=␣%d\n";

  /**
   * Main
   */
  public static void main(String[] args) {
    new SpanningTreeProblem().run();
  }

  /**
   * Run interface to be called from main().
   */
  public void run() {
    Scanner in = new Scanner(System.in);
    int caseCount = Integer.parseInt(in.nextLine());
    int loopCount = 0;

    // Keep looping while we have cases to test.
    while (caseCount - loopCount > 0) {
      int minimumWeight = 0;
      vertexCount = in.nextInt();
      edgeCount = in.nextInt();
```

```
    // Construct visited map and adjacency list.
    visited = new Boolean[vertexCount];
    adjacencyList = new ArrayList<ArrayList<edge>>();

    // Create priority queue.
    edge comp = new edge( 0, 0);
                      edges = new PriorityQueue<edge>(edgeCount, comp);

    // Set all vertices to not visited and ensure Adjacency list has items.
    for (int n = 0; n < vertexCount; n++) {
      visited[n] = false;
      adjacencyList.add(new ArrayList<edge>());
    }

    // Read all edges into the adjacency list.
    while (edgeCount-- > 0) {
      int source = in.nextInt();
      int dest = in.nextInt();
      int weight = in.nextInt();
      adjacencyList.get(source).add(new edge(dest, weight));
      adjacencyList.get(dest).add(new edge(source, weight));
    }

    // Set our source vertex to be visited.
    visited[0] = true;

    // Add all vertex 0 edges to the priority queue
    addEdges(0);

    // While not visited all vertices, continue to loop
    while (!edges.isEmpty()) {
      edge e = edges.poll();

      // if vertex is not yet connected
      if (visited[e.vertex] == false) {

        // take edge and process all edges incident to this edge
        // each edge is in priority queue only once
        minimumWeight += e.weight;
        addEdges(e.vertex);
      }
    }

    System.out.printf(OUTPUT_STRING, ++loopCount, minimumWeight);
  }
}

/**
 * Add edges of vertex to the priority queue, whose edge has not been visited.
 *
 * @param vertex
 */
public void addEdges(int vertex) {
  visited[vertex] = true;
  for (edge e : adjacencyList.get(vertex)) {
    // Add in edge, if neighbour has NOT been visited, to avoid cycles.
    if (visited[e.vertex] == false) {
      edges.add(e);
    }
  }
}

/**
 * Edge container class.
 */
public class edge implements Comparator<edge> {

  /**
   * destination vertex and weight.
   */
  public int vertex, weight;
```

```
    /**
     * Constructor
     * @param v The destination vertex
     * @param w The weight of the edge.
     */
    public edge(int v, int w) {
      vertex = v;
      weight = w;
    }

    /**
     * Compare function for priority queue.
     */
    @Override
    public int compare(edge e1, edge e2) {
      if (e1.weight < e2.weight) {
        return -1;
      } else if (e1.weight > e2.weight) {
        return 1;
      } else {
        return 0;
      }
    }
  }
}
```

Listing 40: Solution to Minimum Spanning Tree Problem (Java)

The solution to the Minimum Spanning Tree problem is implemented slightly different to the algorithm implementation shown in 39, in that the problem solution utilises an adjacency list to represent the graph rather than an adjacency matrix. This suits the problem well, as we are interested in the sum of the tree and not the tree itself.

The implementation, utilises a priority queue[26] of edges that are located incident to the edges that have been explored, and removes the head of the queue. If this edge has not been visited, we add it's weight to the sum of the tree, and note that we have visited the vertex. We then add all edges incident to the vertex just visited to the priority queue. We continue looping until the priority queue has been exhausted (which indicates that we have visited all vertices in the graph that are reachable from vertex 0).

The few edge cases to consider would be very dense graphs, very sparse graphs, or graphs that contain zero edges. An graph with all edges having the same weight, whilst will derive a different minimum spanning tree, is not a problem with the solution implementation, due to it's use of the priority queue. Another item to consider is how to handle cycles within the graph representation, that is a new edge, that has both source and destination vertex already in the resulting tree. (These should simply be discarded, and are within the implementation).

6.3.3.2 Kruskal's with Union Find Algorithm Kruskal's Minimum Spanning Tree Algorithm works in a very similar manner to that described of The Prim-Jarník Algorithm, with one exception, that egdes are added to the tree in order of lowest cost irrespective of the edge they incident with, unless addition of the edge results in a cycle forming.

6.3.3.2.1 Description of working The algorithm like Prim's essentially builds the minimum spanning tree based on the least weight of edges that lead to vertices that are not part of the tree.

This is achieved through the use of a single priority queue, that holds all edges of the graph, with the least weighted egdes being at the start/head of the queue. On each iteration, it will remove the next least cost edge, add it to the tree, if and only if the addition of the edge does not result in a cycle within the tree.

Figure 6.12 demonstrates the algorithm, based on the initial graph in Figure 6.7.

Based on the weights, of edges within the graph depicted in Figure 6.12(a), will be stored in the priority queue in the following order: {(f,g), (b,e), (b,d), (d,e), (a,b), (b,c), (e,g), (c,e), (d,f), (a,d), (e,f)}. In (a),

[26]The priority queue is primed with edges incident to vertex 0.

Algorithm 19 Kruskal's Algorithm

Input A Graph G which is a fully connected graph.
Output A tree T whose properties match those of a Minimum Spanning Tree.

```
procedure KRUSKALSMST(G)
    for all vertex v in G do
        Define an elementary cluster C(v) ← v
    end for
    Initialise priority queue Q to contains all edges in G, using weights as keys.
    T ← ∅
    while T has fewer than n − 1 egdes do
        (v, u) ← Q.removeMin()
        Let C(v) be the cluster containing v, and let C(u) be the cluster containing u.
        if C(v) ≠ C(u) then
            Add edge (v, u) to T
            Merge C(v) and C(u) into one cluster. (C(v) ∪ C(u))
        end if
    end while
    return T
end procedure
```

the first edge is automatically added to the Tree (as there are no other edges currently in the tree). (b), (c) add additional edges to form a second sub-tree. In (d), the next edge in the priority queue is (d,e), however this is not added as it would create a cycle within the tree. The rest of the edges are added into the tree, until the number of edges in the resulting Tree is equal to the number of vertices - 1.

One of the key aspects of the algorithm is the method used to determining if $C(v) \neq C(u)$ is true or false as well as performing the merge operation between the two clusters (or sub-trees). To achieve this we utilise a union find data structure, that offers high performance in both checking if $C(v) \neq C(u)$ and also performing the merge operation.

Kruskal's Algorithm will typically have a performance complexity of **O(m log n)**.

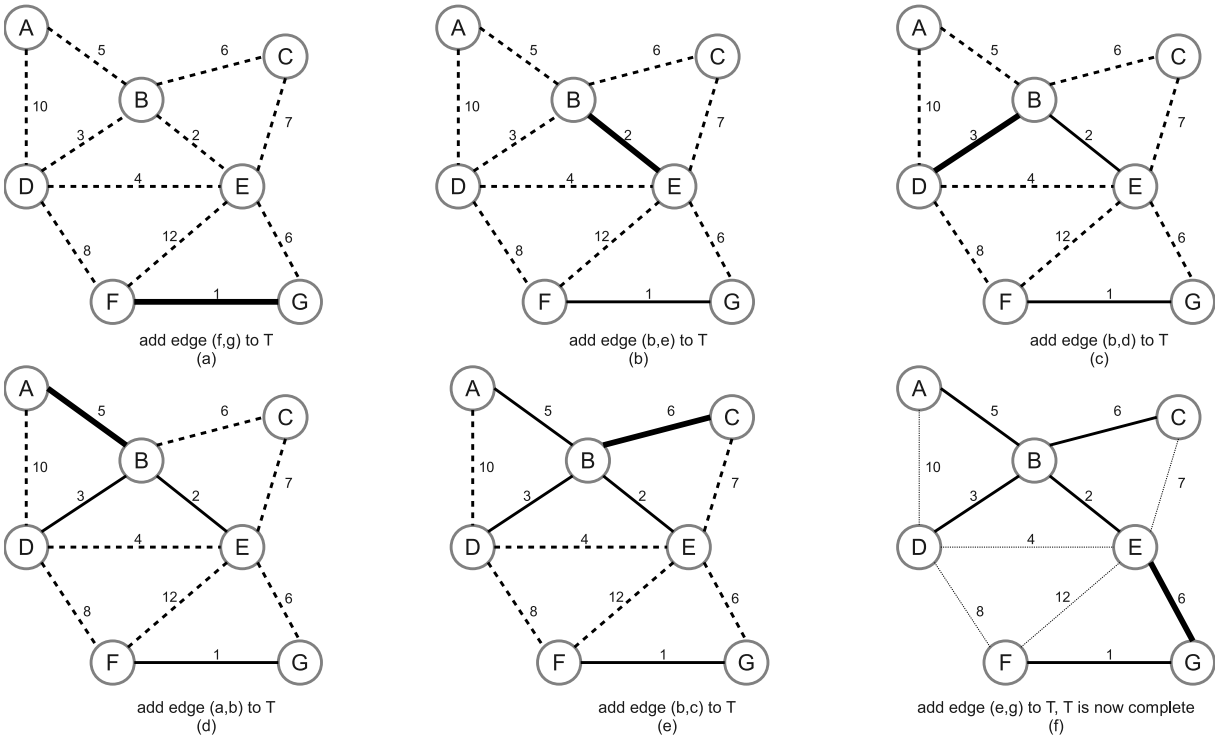


Figure 6.12: Kruskal's Algorithm

6.3.3.2.2 Implementation An implementation of Kruskal's Algorithm can be seen in Listing 41.

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.PriorityQueue;

public class Kruskals {

  /**
   * Find Minimum Spanning Tree using Kruskal's Algorithm
   * @param edges Priority queue of all edges available within the graph
   * @param vertexCount The number of vertices in the graph.
   * @return Edges that make up the minimum spanning tree.
   */
  public ArrayList<edge> Kruskals(PriorityQueue<edge> edges, int vertexCount) {
    int minimumWeight = 0;
    ArrayList<edge> tree = new ArrayList<edge>();
    // Build connectivity map for Union-Find operation.
    unode[] vertexMap = new unode[vertexCount];
    for (int i = 0; i < vertexCount; i++) {
      // Set all vertices to be in their own trees.
      vertexMap[i] = new unode(null, 0);
    }

    // Begin Kruskal's Algorithm with Union-Find
    while (!edges.isEmpty()) {
      edge currentEdge = edges.poll();
      unode p = find(vertexMap[currentEdge.source]); // Get parent of tree that source is in
      unode q = find(vertexMap[currentEdge.destination]); // Get parent of tree that dest is in.
      if (q != p) {
        // If different parents, add the edge in.
        tree.add(currentEdge);
        minimumWeight += currentEdge.weight;
        union(p, q); // Join the two, ensuring minimum depth.
      }
    }
    return tree;
  }

  /**
   * Edge container class.
   */
  public class edge implements Comparator<edge> {

    /**
     * destination vertex and weight.
     */
    public int source, destination, weight;

    /**
     * Constructor
     *
     * @param v The destination vertex
     * @param w The weight of the edge.
     */
    public edge(int source, int destination, int weight) {
      this.source = source;
      this.destination = destination;
      this.weight = weight;
    }

    /**
     * Compare function for priority queue.
     */
    @Override
    public int compare(edge e1, edge e2) {
      if (e1.weight < e2.weight) {
        return -1;
      } else if (e1.weight > e2.weight) {
        return 1;
      } else {
        return 0;
      }
    }
```

```
  }

  /**
   * Parent container class for Union-Find data structure.
   */
  public class unode {

    public unode parent;
    public int depth;

    public unode(unode parent, int depth) {
      this.parent = parent;
      this.depth = depth;
    }
  }

  /**
   * FIND method.
   */
  public unode find(unode current) {
    if (current.parent == null) {
      return current;
    }
    return (current.parent = find(current.parent));
  }

  /**
   * UNION method.
   */
  public void union(unode p, unode q) {
    if (p.depth > q.depth) {
      q.parent = p;
    } else if (p.depth < q.depth) {
      p.parent = q;
    } else {
      p.parent = q;
      p.depth += 1;
    }
  }
}
```

Listing 41: Kruskal's Algorithm (Java)

6.3.3.2.3 Sample Problem- Minimum Spanning Tree For this sample problem I will reuse the same problem from Section 6.3.3.1.3.

6.3.3.2.4 Sample Solution A sample solution to the Minimum Spanning Tree problem can be found in Listing 42.

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Scanner;

public class KruskalsMinimumSpanningTree {

  int vertexCount;
  int edgeCount;
  PriorityQueue<edge> edges;
  ArrayList<edge> tree;
  private final static String OUTPUT_STRING = "Test␣%d,␣minimum␣spanning␣tree␣weight␣=␣%d\n";

  /**
   * Main
   */
  public static void main(String[] args) {
    new KruskalsMinimumSpanningTree().run();
  }

  /**
```

```
  * Run interface to be called from main().
  */
public void run() {
  Scanner in = new Scanner(System.in);
  int caseCount = Integer.parseInt(in.nextLine());
  int loopCount = 0;

  // Keep looping while we have cases to test.
  while (caseCount - loopCount > 0) {
    int minimumWeight = 0;
    vertexCount = in.nextInt();
    edgeCount = in.nextInt();

    // Create priority queue and MST.
    edge comp = new edge(0, 0, 0);
    edges = new PriorityQueue<edge>(edgeCount, comp);
    tree = new ArrayList<edge>();

    // Read all edges into the adjacency list.
    for (int i = 0; i < edgeCount; i++) {
      edges.add(new edge(in.nextInt(), in.nextInt(), in.nextInt()));
    }

    // Build connectivity map for Union-Find operation.
    unode[] vertexMap = new unode[vertexCount];
    for(int i = 0; i < vertexCount; i++){
      // Set all vertices to be in their own trees.
      vertexMap[i] = new unode(null, 0);
    }

    // Begin Kruskal's Algorithm with Union-Find
    while (!edges.isEmpty()) {
      edge currentEdge = edges.poll();
      unode p = find(vertexMap[currentEdge.source]); // Get parent of tree that source is in
      unode q = find(vertexMap[currentEdge.destination]); // Get parent of tree that dest is in.
      if(q != p){
        // If different parents, add the edge in.
        tree.add(currentEdge);
        minimumWeight += currentEdge.weight;
        union(p,q); // Join the two, ensuring minimum depth.
      }
    }
    System.out.printf(OUTPUT_STRING, ++loopCount, minimumWeight);
  }
}

/**
 * Edge container class.
 */
public class edge implements Comparator<edge> {

  /**
   * destination vertex and weight.
   */
  public int source, destination, weight;

  /**
   * Constructor
   *
   * @param v The destination vertex
   * @param w The weight of the edge.
   */
  public edge(int source, int destination, int weight) {
    this.source = source;
    this.destination = destination;
    this.weight = weight;
  }

  /**
   * Compare function for priority queue.
   */
  @Override
  public int compare(edge e1, edge e2) {
```

```
      if (e1.weight < e2.weight) {
        return -1;
      } else if (e1.weight > e2.weight) {
        return 1;
      } else {
        return 0;
      }
    }
  }

  /**
   * Parent container class for Union-Find data structure.
   */
  public class unode {

    public unode parent;
    public int depth;

    public unode(unode parent, int depth) {
      this.parent = parent;
      this.depth = depth;
    }
  }

  /**
   * FIND method.
   */
  public unode find(unode current) {
    if (current.parent == null) {
      return current;
    }
    return (current.parent = find(current.parent));
  }

  /**
   * UNION method.
   */
  public void union(unode p, unode q) {
    if (p.depth > q.depth) {
      q.parent = p;
    } else if (p.depth < q.depth) {
      p.parent = q;
    } else {
      p.parent = q;
      p.depth += 1;
    }
  }
}
```

Listing 42: Solution to Minimum Spanning Tree Problem (Kruskal's) (Java)

One of the key aspects of the implementation is the method utilised in determining if $C(v) \neq C(u)$ is true or false as well as performing the merge operation between the two clusters (or sub-trees). To achieve this we utilise a union find data structure, that offers high performance in both checking if $C(v) \neq C(u)$ and also performing the merge operation. The Union Find Data structure is very simple, in that an array of vertices in maintained whose data contains the root of the subtree in which it exists, as well as the depth of the tree in which it exists. To determine if two vertices are in the same subtree, we simply compare their root vertex, and if the same, both vertices are in the same subtree, otherwise they are not and should be merged. (Both of these are **O(1)** operations, except when path compression is needed). Figure 6.13 illustrates how the Union Find datastructure works, utilising the graph shown in 6.12.

1. At the start of the main loop, all vertices are placed into their own sub-trees (parent is null, that is they are the root of the tree). (a)
2. On the first iteration, we determine if the two vertices share the same parent, and in this case they do not. We simply merge both trees into 1 tree. (b)
3. With iterations (c) through (g), we perform the same action as above, except in step (e) where both B and E share a common root vertex within the sub-tree. Also, note that path compression doesn't occur in (e) as vertex B is not processed in the `find()` method.

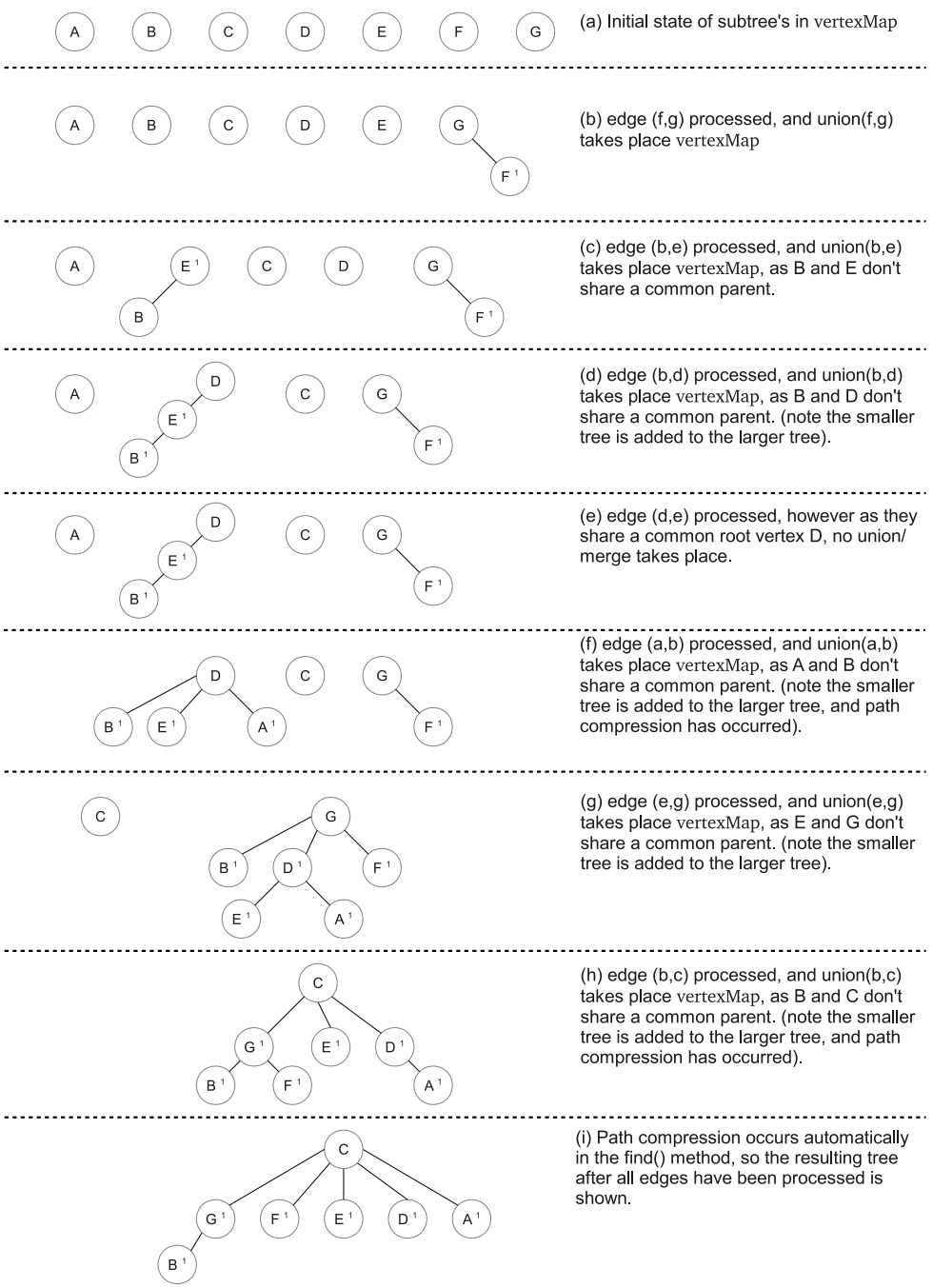


Figure 6.13: Kruskal's with Union Find

4. On the last iteration, however we have to merge two subtrees, which results in the figure shown in (h). However the path compression within the method `find()`, modifies the respective parent pointers so that the resultant graphs looks like that shown in (i). This flatterning of the Union Find data structure ensures minimum time is needed within the operation to determine is two sub-trees share the same parent. If another find option occurs involving vertex B, then vertex B will have it's parent changed to vertex C as well.

As can be seen in Figure 6.13, the depth of the Union Find data structure is minimised due to the path compression, so that future searches for a vertices root vertex within it's sub-tree is very quick.

The few edge cases to consider would be very dense graphs, very sparse graphs, or graphs that contain zero edges. An graph with all edges having the same weight, whilst will derive a different minimum

spanning tree, is not a problem with the solution implementation, due to it's use of the priority queue. Another item to consider is how to handle cycles within the graph representation, that is a new edge, that has both source and destination vertex already in the resulting tree. (These should simply be discarded, and are within the implementation).

6.3.4 Single Source Shortest Path

As seen previously, the Breadth-First Search is capable of finding the shortest path within an unweighted graph, however in the real world edges within a graph almost always have a weight associated. For weighted graphs (being either directed or undirected) we require a different algorithm. The most popular single source shortest path algorithm was developed by Edsger Dijkstra, and is commonly known as Dijkstra's Algorithm.

Dijsktra's algorithm is commonly used with network routing protocols, specifically Open Shortest Path First and IS-IS. It also has many other applications including path-finding in Artificial Intelligence (as utilised in computer games) or Robotics. One advantage of Dijkstra's algorithm, is that it will also calculate the shortest path from a source vertex to all other vertices within the operation of the algorithm.

Algorithm 20 Dijkstra's Algorithm

Input A simple undirected Graph G with non-negative edge weights, and a source vertex v to calculate path from.
Output A label $D[u]$, for each vertex u of G, such that $D[u]$ is the distance from v to u in G.

```
procedure DijkstrasShortestPaths(G, v)
    D[v] ← 0
    for all vertex u ≠ v in G do
        D[u] ← +∞
        P[u] ←undefined                          ▷ Vector P to hold previous vertex
    end for
    Let a priority queue Q contain all vertices of G using the D labels as keys.
    while Q is not empty do
        u ← Q.removeMin()
        for all vertex z adjacent to u such that z is in Q do
            if D[u] + w((u, z)) < D[z] then
                D[z] ← D[u] + w((u, z))
                P[z] ← u
                Change to D[z] the key of vertex z in Q
            end if
        end for
    end while
    return D
end procedure
```

6.3.4.1 Description of working The algorithm applies a greedy method in that starting at vertex v in graph G, we simple find the shortest path to the next vertex which has not been visited previously. Using this new information, we update all distances to vertices via this new vertex using a relaxation procedure, in that it takes the old distance estimate and checks if it can be improved to get closer to its real distance. This is performed by the following section of the algorithm:

```
if D[u] + w((u, z)) < D[z] then
    D[z] ← D[u] + w((u, z))
end if
```

Note that if the newly discovered path to z is no better than the old way, then we do not change $D[z]$.

Figure 6.14 illustrates how Dijkstra's Algorithm works utilising the graph in Figure 6.7, starting at vertex A.

Utilising Adjacency Matrix as starting point as shown in Table 5 for graph displayed in 6.7.

Step 1.
Initially, D contains the Adjacency Matrix Values from Vertex A.

		A	B	C	D	E	F	G
z	Visited	D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]
	{A}	∞	5	∞	10	∞	∞	∞

Step 2.
Select the smallest weighted edge from D, not in Visited, (in this case B), and now update the shortest path weight. Note, the weight from A to D, has been updated.

		A	B	C	D	E	F	G
z	Visited	D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]
B	{A,B}	∞	5	11	**8**	7	∞	∞

Step 3.
Select the smallest weighted edge from D, not in Visited, (in this case E), and now update the shortest path weight.

		A	B	C	D	E	F	G
z	Visited	D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]
E	{A,B,E}	∞	5	11	8	7	19	13

Step 4.
Select the smallest weighted edgefrom D, not in Visited, (in this case D), and now update the shortest path weight. Note, the weight from A to F has been updated, as path {A, B, D, F} = 16 is shorter than {A, B, E, F} = 19.

		A	B	C	D	E	F	G
z	Visited	D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]
D	{A,B,D,E}	∞	5	11	8	7	**16**	13

Step 5.
Select the smallest weighted edge from D, not in Visited, (in this case C), and now update the shortest path weight. No weights are updated.

		A	B	C	D	E	F	G
z	Visited	D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]
C	{A,B,C,D,E}	∞	5	11	8	7	16	13

Step 6.
Select the smallest weighted edge from D, not in Visited, (in this case G), and now update the shortest path weight. Note, the weight from A to F has been updated.

		A	B	C	D	E	F	G
z	Visited	D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]
G	{A,B,C,D,E,G}	∞	5	11	8	7	**14**	13

Step 7.
Select the smallest weighted edge from D, not in Visited, (in this case F), and now update the shortest path weight.

		A	B	C	D	E	F	G
z	Visited	D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]
F	{A,B,C,D,E,F,G}	∞	5	11	8	7	14	13

Step 8.
Since all vertices have been visited, we exit the algorithm. D now contains the shortest weight/cost from vertex A to all other vertices.

Figure 6.14: Dijsktra's Algorithm

As shown, Dijkstra's Algorithm gives you the weight of the shortest path to each other vertice, not the actual path itself. To find the actual path, Algorithm 21 is needed.

Algorithm 21 Dijkstra's Algorithm (Find Path)

Input Vector P of previously visited vertices (constructed in Dijkstra's Algorithm), source vertex v and target vertex t
Output Vector $Path$ with path from v to u

procedure DIJKSTRASSHORTESTPATH(P, v, t)
  $Path \leftarrow \emptyset$ ▷ stack datatype
  $u \leftarrow t$
  while $P[u] \neq$ undefined **do**
    $Path$.push() $\leftarrow u$ ▷ push u onto stack
    $u \leftarrow P[u]$
  end while
  return $Path$
end procedure

This additional part to the algorithm simply walks backwards through the Previous Vertex Vector, until the next path is undefined (meaning we are at the start or origin vertex).

Dijkstra's algorithm when implemented correctly with an efficient priority queue implementation, will offer **O(m log n)** performance when used with an adjacency list representation of a graph.

6.3.4.2 Implementation An implementation of Dijkstra's Algorithm can be seen in Listing 43. This implementation utilises an adjacency matrix as the graph represention data structure, thus offers **O(n^2)** performance. I've utilised an adjacency matrix in this example, as will allows easier/quicker understanding of the algorithm, as well as making to easily translatable to other programming languages such as C[27].

```
import java.util.Stack;

public class DijkstrasAlgorithm {

  /**
   * Find the shortest path from source vertex to destination vertex
   *
   * @param adjacencyMatrix The graph represented as an adjacency matrix.
   * @param sourceVertex The vertex to start from.
   * @param destinationVertex The destination vertex
   * @return A stack with the path from source to destination.
   */
  public Stack<Integer> DijkstraShortestPath(int[][] adjacencyMatrix, int sourceVertex,
                                             int destinationVertex) {
    final int UNDEFINED = -1;

    // Define data structures needed for implementation.
    Stack<Integer> path = new Stack<Integer>();
    boolean[] visited = new boolean[adjacencyMatrix.length];
    int[] previousVertex = new int[adjacencyMatrix.length];
    int[] pathWeights = new int[adjacencyMatrix.length];

    // Clear all arrays being used.
    for (int i = 0; i < adjacencyMatrix.length; i++) {
      pathWeights[i] = Integer.MAX_VALUE; // Infinity
      previousVertex[i] = UNDEFINED;
      visited[i] = false;
    }

    // Set the starting vertex.
    pathWeights[sourceVertex] = 0;

    // Follow all vertices, calculating paths/distances.
```

[27] The only modification to be usable in C, would be to implement the Stack<E> datastructure, which can be easily done with a linked list datastructure, or even just a normal array (and maintaining a pointer/index to the last element in the array).

```
    for (int k = 0; k < adjacencyMatrix.length; k++) {
      int min = -1;

      // Find shortest path to vertex not yet visited.
      for (int i = 0; i < adjacencyMatrix.length; i++) {
        if (!visited[i] && (min == -1 || pathWeights[i] < pathWeights[min])) {
          min = i;
        }
      }

      // Flag we are at a new vertex.
      visited[min] = true;

      // perform edge relaxation
      for (int i = 0; i < adjacencyMatrix.length; i++) {
        if (adjacencyMatrix[min][i] > 0) {
          if (pathWeights[min] + adjacencyMatrix[min][i] < pathWeights[i]) {
            pathWeights[i] = pathWeights[min] + adjacencyMatrix[min][i];
            previousVertex[i] = min;
          }
        }
      }
    }

    // All vertices have been visited, so now construct a path
    path.push(destinationVertex);
    while (previousVertex[destinationVertex] != UNDEFINED) {
      path.push(previousVertex[destinationVertex]);
      destinationVertex = previousVertex[destinationVertex];
    }

    return path;
  }
}
```

Listing 43: Dijkstra's Algorithm (Java)

6.3.4.3 Sample Problem - Single Source Shortest Path Find the shortest path between a given node and all the other nodes in a weighted directed graph.

INPUT

The first line of input will be a number on a line by itself which is the number of test cases to run. For each test case, the first line will be three each numbers separated by a space N, E and X, where N ($1 <= N <= 5000$) is the number of nodes in the graph, E ($1 <= E <= 10000$) is the number of edges and X ($0 <= X < N$) is the source node for the shortest paths. The graph nodes will be numbered 0 to N-1. Each of the next E lines contain three numbers S, D and W each separated by a space representing an edge in the graph from S ($0 <= S < N$) to D ($0 <= D < N$) with weight W ($0 < W <= 1000$).

SAMPLE INPUT

```
2
5 7 0
0 1 10
1 3 2
1 4 6
3 4 3
2 4 6
0 4 20
0 2 30
5 6 1
0 1 8
0 3 2
0 4 5
1 2 6
2 3 4
3 4 3
```

OUTPUT

For each test case, output the test case number on a line followed by a space and then a space separated list shortest path weights for each node in ascending node number order. If there is no path from the source node to another node print 'NP'. See the example output below.

SAMPLE OUTPUT

```
1
0 10 30 12 15
2
NP 0 6 10 13
```

6.3.4.4 Sample Solution A sample solution to the Single Source Shortest Path problem can be found in Listing 44.

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Scanner;

public class ShortestPathProblem {

  private static int INFINITY = Integer.MAX_VALUE;

  public static void main(String[] args) {
    new ShortestPathProblem().run();
  }

  public void run() {

    Vertex[] vertexList;  // List of all vertices in graph
    PriorityQueue<Vertex> verticesQueue;

    Scanner scn = new Scanner(System.in);
    int testCount = scn.nextInt();

    // Loop through all test cases.
    for (int testItem = 1; testItem <= testCount; testItem++) {
      int numVertices = scn.nextInt();
      int numEdges = scn.nextInt();
      int sourceVertex = scn.nextInt(); // source vertex

      // create array to store N vertices
      vertexList = new Vertex[numVertices];
      for (int n = 0; n < numVertices; n++) {
        vertexList[n] = new Vertex(numVertices);
      }

      // set source vertex (distance = 0 at source!)
      vertexList[sourceVertex].distance = 0;

      for (int edge = 0; edge < numEdges; edge++) {
        int s = scn.nextInt(); // source vertex
        int d = scn.nextInt(); // destination vertex
        int w = scn.nextInt(); // distance from s to d

        // add to adjacency list
        vertexList[s].adjacencyList[d] = w;
      }

      // Create priority queue of N elements, and queue all elements.
      Vertex comp = new Vertex(numVertices);
      verticesQueue = new PriorityQueue<Vertex>(numVertices, comp);
      for (Vertex v : vertexList) {
        verticesQueue.add(v);
      }

      // Start main Dijkstra'a Algorithm
      while (!verticesQueue.isEmpty()) {
        Vertex uVertex = verticesQueue.poll();
```

```
      if (uVertex.distance == INFINITY) // vertex can't be reached
      {
        break;
      }

      // Loop through all neighbouring vertices
      for (int n = 0; n < numVertices; n++) {
        if (uVertex.adjacencyList[n] != INFINITY
              && verticesQueue.contains(vertexList[n])) {
          verticesQueue.remove(vertexList[n]); // Remove old D[n] value.
          vertexList[n].setDist(uVertex.distance + uVertex.adjacencyList[n]);
          verticesQueue.add(vertexList[n]); // Add updated D[n] value to Queue
        }
      }
    }

    // Output the resulting Distance from source.
    System.out.printf("%d\n", testItem);
    for (Vertex vertex : vertexList) {
      vertex.printDistance();
    }
    System.out.println();
  }
}

/**
 * Class to define a vertex, including it's neighbouring vertices.
 */
public class Vertex implements Comparator<Vertex> {

  public int distance;  // known distance from source vertex
  public int[] adjacencyList; // weights to neighbouring vertices.

  /**
   * Default Constructor.
   * @param numNeighbours The number of vertices in the graph.
   */
  public Vertex(int numNeighbours) {
    distance = INFINITY;

    // Create a list of weights to neighbours and set the weight to INFINITY.
    adjacencyList = new int[numNeighbours];
    for (int i = 0; i < numNeighbours; i++) {
      adjacencyList[i] = INFINITY;
    }
  }

  /**
   * Update the distance for this vertex.
   * @param newDistance
   */
  public void setDist(int newDistance) {
    if (newDistance != INFINITY) {
      if (distance == INFINITY || newDistance < distance) {
        distance = newDistance;
      }
    }
  }

  /**
   * Print the distance from source vertex to this vertex.
   */
  public void printDistance() {
    if (distance == INFINITY) {
      System.out.print("NP␣");
    } else {
      System.out.printf("%d␣", distance);
    }
  }

  /**
   * Comparator method for priority queue.
   */
```

```
    @Override
    public int compare(Vertex o1, Vertex o2) {
      if (o1.distance < o2.distance) {
        return -1;
      } else if (o1.distance > o2.distance) {
        return 1;
      } else {
        return 0;
      }
    }
  }
}
```

Listing 44: Solution to Single Source Shortest Path Problem (Java)

The few edge cases to consider would be very dense graphs, very sparse graphs, or graphs that contain zero edges. Another edge case that should be considered is the maximum weight between vertices, and ensuring that the data type that holds the weight from source to destination is capable of representing the value without overflow. (That is any possible weight from source to destination is less than the maximum value used to hold the weight).

As the problem focuses around the algorithm itself, the implementation of the algorithm is the key factor in success with this problem. Two items to note about the implemention of the sample solution compared to the reference implementation, is that we don't maintain a data structure to find a path from a source to destination vertex, and the use of the priority queue.

The priority queue itself offers a significant improvement over the sample implementation in Listing 43, by providing a single datastructure that maintains a list of unvisited vertices and also sorts the vertices by weight, so the head of the queue is the next unvisited vertex with the least weight. This implementation as shown should offer **O((n+m) log n)** performance.

6.3.5 All Pairs Shortest Path

While is it extremely useful to find the shortest path from a source vertex to any other vertex within a graph, an alternative is to find the shortest path from any vertex to any other vertex in a graph utilising a single algorithm. This is extremely useful if the paths and path options will remain static throughout the operation of the software system, as is typically found within computer games. Thereby, rather than having to recalculate the path for each entity within the game, we can simply lookup the information as it has been precomputed. (Effectively a **O(1)** operation vs a **O((n+m) log n)** operation).

A popular algorithm to perform this is the Floyd-Warshall Algorithm, as shown in Algorithm 22.

6.3.5.1 Description of working The Floyd-Warshall Algorithm is an example of dynamic programming that compares all possible paths through the graph between each pair of vertices. The remarkable aspect of the algorithm is the simplicity of the design, as well as having a runtime performance of only **O(n^3)**.

If we define $f(i, j, k)$ as the shortest distance from i to j, using $1 \ldots k$ as intermediated vertex, we find that:

- $f(i, j, n)$is the shortest distance from i to j, and
- $f(i, j, 0)$is the cost of(i, j).

Understanding that the optimal part for $f(i, j, k)$, may or may not have k as an intermediate vertex, we find:

- If it does, $f(i, j, k) = f(i, k, k - 1) - f(k, j, k - 1)$
- Otherwise, $f(i, j, k) = f(i, j, k - 1)$

Algorithm 22 Floyd-Warshall Algorithm

Input A directed Graph $\vec{G}$ with non-negative edge weights, with n vertices.
Output A numbering $v_1, v_2, \ldots, v_n$ of vertices of $\vec{G}$ and a matrix D, such that $D[i, j]$ is the distance from v_i to v_j in $\vec{G}$.

```
procedure AllPairsShortestPaths(G⃗)
    let v_1, v_2, ..., v_n be an abitrary numbering of the vertices of G⃗
    for i ← 1 to n do
        for j ← 1 to n do
            if i = j then
                D^0[i, i] ← 0
            end if
            if (v_i, v_j) is an edge in G⃗ then
                D^0[i, j] ← w((v_i, v_j))
            else
                D^0[i, j] ← +∞
            end if
        end for
    end for
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D^k[i, j] ← min{D^{k-1}[i, j], D^{k-1}[i, k] + D^{k-1}[k, j]}
            end for
        end for
    end for
    return D^n
end procedure
```

Which leads to $f(i, j, k)$ being the minimum of the two quantities above. If the above properties are extended to all vertices within the graph, we may find all paths in one easy operation!

Like Dijkstra's Algorithm, the Floyd-Warshall Algorithm only finds the minimum weight of the path, however with a small modification it is possible to determine the path as well.

In order to determine the path between two vertices, we are required to maintain an additional matrix that defines the path P. If the new new weight is less than the previous weight, then we store k at $P(i, j)$. Once all weights have been determined we use Alogrithm 23 to determine the path.

Algorithm 23 Floyd-Warshall Algorithm (Find Path)

Input Matrix P of previously visited vertices (constructed in Floyd-Warshall Algorithm), source vertex i and target vertex j
Output Vector $Path$ with path from i to j

```
define Path ← ∅                                   ▷ Global data, Queue datatype

procedure FloydWarshallShortestPath(P, i, j)
    if P[i, j] = null then
        Path.add() ← P[i, j]                      ▷ add k onto end of queue
    else
        FloydWarshallShortestPath(P, i, P[i, j])
        FloydWarshallShortestPath(P, P[i, j], j)
    end if
end procedure
```

6.3.5.2 Implementation An implementation of the Floyd-Warshall Algorithm is shown in Listing 45. This implementation expects that the adjacency matrix is complete with information, and that path

information is required.

```
import java.util.ArrayList;

public class FloydWarshall {

  private int[][] path;
  private int[][] pathWeights;
  ArrayList<Integer> sdpath;
  private final static int NULL = -1;

  /**
   * Perform Floyd-Warshall Algorithm on weighted directed graph.
   * @param matrix Adjacency Matrix representation of grpah.
   * @return Matrx of costs for graph.
   */
  public int[][] AllPairsShortestPath(int[][] matrix) {
    int numberVertices = matrix.length;

    // create new storage container for path and weight information
    path = new int[numberVertices][numberVertices];
    pathWeights = new int[numberVertices][numberVertices];

    // Initialise containers;
    for (int i = 0; i < numberVertices; i++) {
      for (int j = 0; j < numberVertices; j++) {
        // If no direct path, set weight to Infinity.
        pathWeights[i][j] = matrix[i][j] > 0 ? matrix[i][j] : Integer.MAX_VALUE;
        if(i == j){
          pathWeights[i][j] = 0;
        }
        path[i][j] = NULL;
      }
    }

    // Main loop.
    for (int k = 0; k < numberVertices; k++) {
      for (int i = 0; i < numberVertices; i++) {
        for (int j = 0; j < numberVertices; j++) {
          // Cast these to long to avoid overflow!
          if ((long)pathWeights[i][j] > (long)pathWeights[i][k]+(long)pathWeights[k][j]) {
            // Store new min weight
            pathWeights[i][j] = pathWeights[i][k] + pathWeights[k][j];
            // Store new path through k.
            path[i][j] = k;
          }
        }
      }
    }
    return pathWeights;
  }

  /**
   * Find the shortest Path from source to destination, based on a previous
   * run of the Floyd-Warshall algorithm.
   * @param source Source Vertex
   * @param destination Destination Vertex
   * @return Path from source to destination
   */
  public ArrayList<Integer> ShortestPath(int source, int destination) {
    sdpath = new ArrayList<Integer>();
    SPath(source, destination);
    return sdpath;
  }

  /**
   * Recursively follow the path from source to destination.
   */
  private void SPath(int source, int destination) {
    if (path[source][destination] == NULL) {
      sdpath.add(path[source][destination]);
    } else {
      SPath(source, path[source][destination]);
      SPath(path[source][destination], destination);
```

```
    }
  }
}
```

Listing 45: Floyd-Warshall Algorithm (Java)

6.3.5.3 Sample Problem - All Pairs Shortest Path Problem Find the shortest path between every pair of nodes in a weighted directed graph.

INPUT

The first line of input will be a number on a line by itself which is the number of test cases to run. For each test case, the first line will be two numbers separated by a space N and E, where N ($1 <= N <= 1000$) is the number of nodes in the graph and E ($1 <= E <= 10000$) is the number of edges. The graph nodes will be numbered 0 to N-1. Each of the next E lines contain three numbers S, D and W each separated by a space representing an edge in the graph from S ($0 <= S < N$) to D ($0 <= D < N$) with weight W ($0 < W <= 1000$).

SAMPLE INPUT

```
2
5 7
0 1 10
1 3 2
1 4 6
3 4 3
2 4 6
0 4 20
0 2 30
5 6
0 1 8
0 3 2
0 4 5
1 2 6
2 3 4
3 4 3
```

OUTPUT

For each test case, output the test case number on a line by itself. The following lines will contain the shortest paths matrix. That is the next n ($0 <= n < N$) lines will each contain a space separated list of the shortest path weight between node n and each other node in ascending node number order. If no path exists between a pair of node print 'NP'. See the example output below.

SAMPLE OUTPUT

```
1
0 10 30 12 15
NP 0 NP 2 5
NP NP 0 NP 6
NP NP NP 0 3
NP NP NP NP 0
2
0 8 14 2 5
NP 0 6 10 13
NP NP 0 4 7
NP NP NP 0 3
NP NP NP NP 0
```

6.3.5.4 Sample Solution A sample solution to the All Pairs Shortest Path problem can be found in Listing 46.

```
import java.util.Scanner;

public class AllPairsShortestProblem {

  private int[][] pathWeights;
  private int[][] adjacencyMatrix;

  public static void main(String[] args) {
    new AllPairsShortestProblem().run();
  }

  public void run() {
    Scanner scn = new Scanner(System.in);
    int testCount = scn.nextInt();

    // Loop through all test cases.
    for (int testItem = 1; testItem <= testCount; testItem++) {
      int numVertices = scn.nextInt();
      int numEdges = scn.nextInt();

      // create adjacency Matrix
      adjacencyMatrix = new int[numVertices][numVertices];
      for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
          adjacencyMatrix[i][j] = 0;
        }
      }

      for (int edge = 0; edge < numEdges; edge++) {
        int s = scn.nextInt(); // source vertex
        int d = scn.nextInt(); // destination vertex
        int w = scn.nextInt(); // distance from s to d

        // add to adjacency matrix
        adjacencyMatrix[s][d] = w;
      }

      // Create the matrix with wieght values.
      AllPairsShortestPath(adjacencyMatrix);

      // Output the resulting Distance from source.
      System.out.printf("%d\n", testItem);
      for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
          if(pathWeights[i][j] == Integer.MAX_VALUE){
            System.out.print("NP␣");
          } else {
            System.out.printf("%d␣", pathWeights[i][j]);
          }
        }
        System.out.println();
      }
    }
  }

  public void AllPairsShortestPath(int[][] matrix) {
    int numberVertices = matrix.length;

    // create new storage container for path and weight information
    pathWeights = new int[numberVertices][numberVertices];

    // Initialise containers;
    for (int i = 0; i < numberVertices; i++) {
      for (int j = 0; j < numberVertices; j++) {
        // If no direct path, set weight to Infinity.
        pathWeights[i][j] = matrix[i][j] > 0 ? matrix[i][j] : Integer.MAX_VALUE;
        if(i == j){
          pathWeights[i][j] = 0;
        }
      }
    }

    // Main loop.
```

```
    for (int k = 0; k < numberVertices; k++) {
      for (int i = 0; i < numberVertices; i++) {
        for (int j = 0; j < numberVertices; j++) {
          // Cast these to long to avoid overflow!
          if ((long)pathWeights[i][j] > (long)pathWeights[i][k]+(long)pathWeights[k][j]) {
            // Store new min weight
            pathWeights[i][j] = pathWeights[i][k] + pathWeights[k][j];
          }
        }
      }
    }
  }
}
```

Listing 46: Solution to All Pairs Shortest Path Problem

The few edge cases to consider would be very dense graphs, very sparse graphs, or graphs that contain zero edges. Another edge case that should be considered is the maximum weight between vertices, and ensuring that the data type that holds the weight from source to destination is capable of representing the value without overflow. (That is any possible weight from source to destination is less than the maximum value used to hold the weight).

6.3.6 A* Shortest Path

Another shortest path and graph traversal algorithm is the A*[28] Algorithm which offers improved runtime performance in the cases that both the source and destination vertex within the graph are known prior to running the algorithm. It is an extension of Dijkstra's Algorithm and A* achieves better performance (with respect to time) by using heuristics.

Algorithm 24 A* Algorithm

Input A vertex $start$, and vertex $goal$ defining the source and destination vertices within an undirected graph. (The vertex data structure holds an adjacency list internally).
Output Weight of the shortest path

```
procedure A*SEARCH(start, goal)
    visited ← ∅
    Q ← start                                          ▷ Q Priority queue
    while Q is not empty do
        current ← Q.removeMin()                        ▷ Remove next best from queue
        if current = goal then
            return goal.gScore
        end if
        visited.add() ← current                        ▷ Add current to the visited set
        for all vertex adjacent to current do
            if adj not in visited then
                gScore ← current.gScore+distance(current, adj)
                if adj not in Q or gScore < adj.gScore then
                    adj.gScore ← gScore
                    adj.hScore ← distance(adj, goal)
                    adj.fScore ← adj.gScore + adj.hScorce
                    update adj on Q
                end if
            end if
        end for
    end while
    return FAILURE
end procedure
```

[28] The A* algorithm is pronounced as "A star".

A* is commonly utilised in real-time robotics for path finding on dynamic graphs, or within navigation aids (such as in car GPS devices).

6.3.6.1 Description of working A* utilises additional information within each vertex data structure to able to perform a heuristic search of the graph. In addition to the vertex location (x,y co-ordinates (Euclidean or Manhattan)) and list of vertices adjacent, these are:

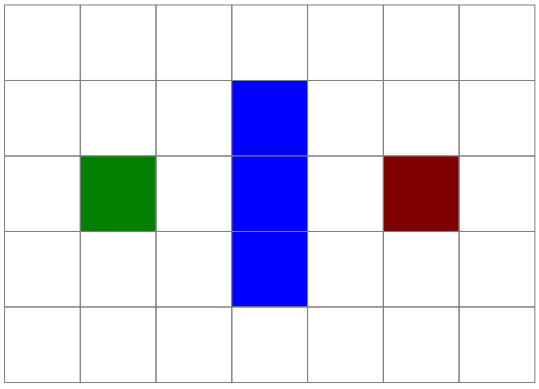
- gScore - cost from the start along the best path.
- hScore - heuristic estimate from the current location to the goal.
- fScore - gScrore + hScore

The algorithm also uses a priority queue of Vertices, that the lowest fScore has priority to be used for searching for the next vertex to traverse. And a Set of visited vertices, to ensure we don't search a vertex already searched.

The key variable in the algorithm, is the fScore, in that this is used to select the next vertex to traverse to in order to find the optimal path to the goal vertex. As noted above the vertex with the lowest fScore should lead to the shortest path, as it is an estimate of the current path cost, and the estimated cost left to the goal.

A demonstration of the A* Shortest Path Algorithm can be seen in Figures 6.15 to 6.21.

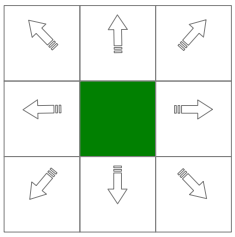
For the purposes of the demonstration, we have divided the search area into a square grid, with the start location being the Green Square (A) and the goal location being the Red Square (B) (Figure 6.15). Simplifying the search area, as we have done here, is the first step in pathfinding with A*. This particular method reduces our search area to a simple two dimensional array. Each item in the array represents one of the squares on the grid, and its status is recorded as walkable or unwalkable. The path is found by figuring out which squares we should take to get from A to B. Once the path is found, our entity or object moves from the center of one square to the center of the next until the target is reached.



Starting Search Area.
(Green being the starting point, red being the goal, and blue being an obstacle)

Figure 6.15: A* Initial Map

Once we have simplified the search area, the next step is to start searching the grid to find our shortest path. We simply search all adjacent vertices, placing them onto the priority queue of vertices. (Figure 6.16)

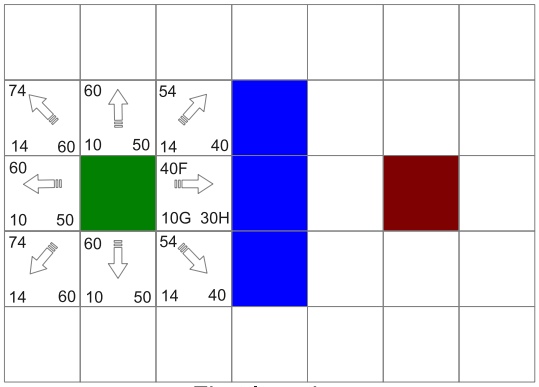


Default Search Pattern.
(Green being the starting point, arrows indicate adjacent cells)

Figure 6.16: A* Default Search Pattern

We start the first iteration by path scoring our adjacent vertices, then place them onto the priority queue. Figure 6.17 demonstrates the gScores, hScores and fScores given to each vertex based on the initial search. (Figure 6.17 is prior to removing the first vertex from the priorty queue). It should be noted, that the hScore can be estimated in a number of ways. The method we use here is called the Manhattan method, where you calculate the total number of squares moved horizontally and vertically to reach the target square from the current square, ignoring diagonal movement, and ignoring any obstacles that may be in the way. We then multiply the total by 10, our cost for moving one square horizontally or vertically. This is (probably) called the Manhattan method because it is like calculating the number of city blocks from one place to another, where you can't cut across the block diagonally.

While typically the hScore is the best guess, it does not represent the minimum number of squares in a direct line of sight from the current vertex to the goal vertex, but rather a best guess at the number of squares between the current and goal vertices. (The better the guess, the quicker the algorithm works).



First Iteration
(Numbers indicate fScore, gScore and hScore. Each move is worth 10 units, with diagonal being 14 units)

Figure 6.17: A* First Iteration

Figure 6.18 displays the first vertex to move to as part of the first iteration. This vertex, has the lowest fScore of 40 (shown with black border). On arriving, all adjacent vertices are added to the priority queue, however in this instance none are added. (All new vertices are obstacle vertices which can't be traversed). When add this vertex to the visited Set, and remove the vertex from the priority queue.

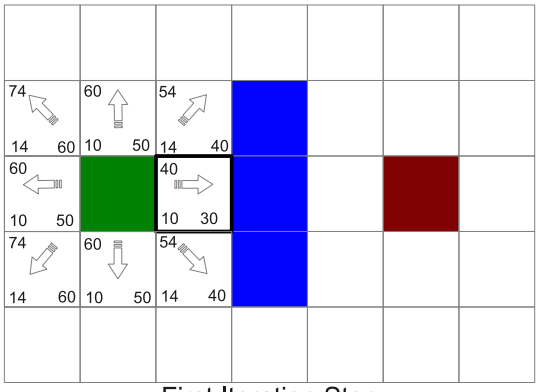


Figure 6.18: A* First Iteration Step

Figure 6.19 shows the next vertex being removed from the proirity queue (below the first vertex in this case), with it's adjacent vertices added to the priority queue, with the various Scores being shown. This time, when we check the adjacent squares we find that the one to the immediate right is a wall square, so we ignore that. The same goes for the one just above that. We also ignore the square just below the wall. Why? Because you can't get to that square directly from the current square without cutting across the corner of the nearby wall. You really need to go down first and then move over to that square, moving around the corner in the process. (Note: This rule on cutting corners is optional. Its use depends on how your vertices are placed.)

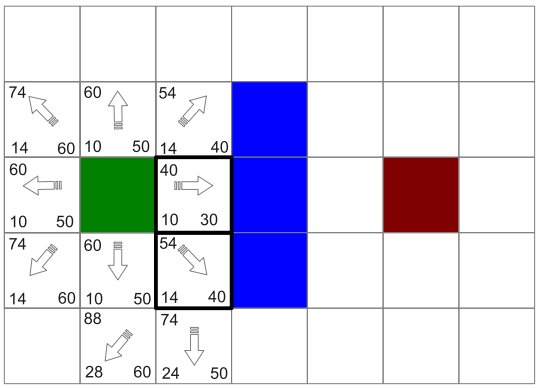


Figure 6.19: A* Second Iteration

We simply repeat this process, until we have a map as displayed in Figure 6.20. As can been seen, we have added some of the squares to the priority queue that are above starting point as well. This is due to the vertices already in the priority queue. After adding the two new vertices in Figure 6.19, the next vertex with the lowest fScore, is actually above and right of the start point! So we add those vertices to the priority queue as well. However on selection of the next vertex, we end up below the last vertex visited and continue on that path. (The final path could go either below or above the obstacle depending on how the priority queue is implementated, but for the sake of the demonstration we assume the oldest item is taken first even if two items have the same priority).

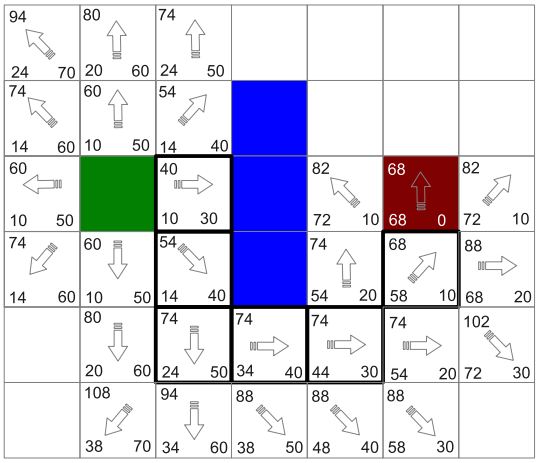


Figure 6.20: A* Final Iteration

Figure 6.20, displays the map after the final iteration of the algorithm. Take note that the path follows the lowest fScore through out the traversal. (Now that the current vertex is the same as the goal vertex, we terminate the algorithm.

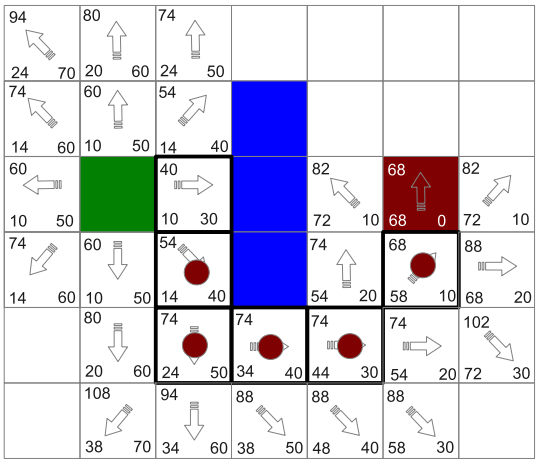


Figure 6.21: A* Path

To determine the actual path followed, we start at the goal square and work backwards moving from one square to its parent, following the arrows. This will eventually take you back to the starting square, and that's your path. It should look like the the illustration in Figure 6.21. Moving from the starting square A to the destination square B is simply a matter of moving from the center of each square (the vertex) to the center of the next square on the path, until you reach the target.

The time complexity of the A* Shortest Path algorithm is deterministic on the hueristic function choosen, and how well it can guess the hScore value. This is due to the number of cells or paths that will be tested during the course of the algorithm running will vary accordingly to how accurate the estimation is. The absolute worst case will see A* function equivalent to Dijkstra's Algorithm. (However this rarely is the case).

6.3.6.2 Implementation

An implementation of the A* Shortest Path Algorithm is shown in Listing 47.

```
import java.util.HashSet;
import java.util.PriorityQueue;
```

```
/**
 * A* Search for Vertices that exist in 2D space.
 */
public class aStar {

  /**
   * Perform a search from start vertex to goal vertex, and return the cost of
   * the path
   * @param vertices An Array of Vertices within the graph.
   * @param start The ID of the starting vertex.
   * @param end The ID of the goal vertex.
   * @return The weight of the path, or Double.POSITIVE_INFINITY if no path!
   */
  public Double aStarSearch(Vertex[] vertices, int start, int goal) {
    PriorityQueue<Vertex> vertexQueue = new PriorityQueue<Vertex>();

    vertices[start].gScore = 0.0;
    vertices[goal].isGoal = true;
    vertexQueue.add(vertices[start]);

    while (!vertexQueue.isEmpty()) {

      Vertex current = vertexQueue.poll();
      if (current.isGoal) {
        return vertices[goal].gScore;
      }

      current.visited = true;
      vertices[current.vertexID].visited = true;

      // for all vertex adjacent to current do
      for (int n = 0; n < vertices.length; n++) {
        if (current.adjList.contains(n) && !vertices[n].visited) {
          Vertex adj = vertices[n];
          Double gScore = current.gScore + euclidianDistance(current, adj);

          // if adj not in Q or gScore < adj.gScore
          if (!vertexQueue.contains(adj) || gScore < adj.gScore) {
            // Update the Scores for the adjacent vertex.
            vertexQueue.remove(adj);
            adj.gScore = gScore;
            adj.hScore = euclidianDistance(adj, vertices[goal]);
            adj.fScore = adj.gScore + adj.hScore;
            // Add this vertex back into the priority queue with updated information.
            vertexQueue.add(adj);
          }
        }
      }
    }

    return Double.POSITIVE_INFINITY; // failed
  }

  /**
   * Return the distance between vertex A and vertex B, using euclidian distance
   *
   * @param vertexA The first vertex
   * @param vertexB The second vertex
   * @return The distance between to two vertices.
   */
  private  Double euclidianDistance(Vertex vertexA, Vertex vertexB) {
    Double dist = Math.sqrt(
              Math.pow((double)(vertexA.xCoordinate - vertexB.xCoordinate), 2.0)
            + Math.pow((double)(vertexA.yCoordinate - vertexB.yCoordinate), 2.0));
    return dist;
  }

  /**
   * Class definition of the Vertex Information, with vertices in 2D space.
   */
  public class Vertex implements Comparable<Vertex> {

    public int xCoordinate; // Coordinate in 2D space.
```

```
    public int yCoordinate; // Coordinate in 2D space.
    public int vertexID;  // The id of the Vertex
    public Double gScore;
    public Double fScore;
    public Double hScore;
    public HashSet<Integer> adjList;
    public Boolean visited; // Has this vertex been visited.
    public Boolean isGoal; // Is this the goal vertex.

    /**
     * Default constructor.
     *
     * @param xCoord X Coordinate in 2D space.
     * @param yCoord Y Coordinate in 3D space.
     * @param entityId The id of the Vertex
     */
    public Vertex(int xCoord, int yCoord, int entityId) {
      xCoordinate = xCoord;
      yCoordinate = yCoord;
      vertexID = entityId;
      gScore = Double.POSITIVE_INFINITY;
      fScore = Double.POSITIVE_INFINITY;
      hScore = Double.POSITIVE_INFINITY;
      adjList = new HashSet<Integer>();
      visited = false;
      isGoal = false;
    }

    @Override
    public int compareTo(Vertex o) {
      if (fScore < o.fScore) {
        return -1;
      } else if (fScore > o.fScore) {
        return 1;
      } else {
        return 0;
      }
    }
  }
}
```

Listing 47: A* Shortest Path (Java)

6.3.6.3 Sample Problem - Shortest Road Trip Road trip! It's time to hit the road from your home town and travel to some randomly selected town on the map. Your map has x, y co-ordinates for all towns and the government has done an amazing job of laying perfectly straight roads between some of the towns as shown on the map.

The distance, d, between two towns, a and b is defined as the Euclidean distance between the coordinates (x_a, y_a) and (x_b, y_b).

$$d = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

INPUT

The first line of input will be a number on a line by itself which is the number of test cases to run. For each test case, the first line will have four numbers. The first number, *N* ($2 \leq N \leq 10{,}000$), is the number of towns on the map. The second number, *E* ($1 \leq E \leq 20{,}000$), is the number of roads on the map. The third and fourth numbers, *S* ($0 \leq S \leq N - 1$) and *D* ($0 \leq D \leq N - 1$) are the numbers of the start and destination towns.

Each of the next *N* lines has two integers, x_i and y_i (-50,000 $\leq x_i, y_i \leq$ 50,000), giving the coordinates of the i^{th} town on the map.

Each of the next *E* lines of input has two integers, n_i and n_j ($0 \leq n_i, n_j \leq N - 1$) giving the end towns of the roads. Roads only connect their start and destination towns.

SAMPLE INPUT

```
2
5 7 0 4
0 0
0 10
5 7
5 3
5 -3
0 1
3 0
3 1
0 2
2 3
4 3
1 2
3 1 0 2
0 0
0 10
10 0
1 0
```

OUTPUT

For each road trip in the input, output the shortest distance between the start and destination towns in the format shown below. If the destination town is not reachable output IMPOSSIBLE.

SAMPLE OUTPUT

```
Road Trip 1:  11.83
Road Trip 2:  IMPOSSIBLE
```

6.3.6.4 Sample Solution A sample solution to the Shortest Road Trip problem can be found in Listing 48.

```
import java.util.HashSet;
import java.util.PriorityQueue;
import java.util.Scanner;

public class ShortestRoadTrip {

  /**
   * Main
   */
  public static void main(String[] args) {
    new ShortestRoadTrip().run();
  }

  public void run() {
    int numTowns, numRoads, startTown, destinationTown;
    Scanner scn = new Scanner(System.in);
    int numTests = scn.nextInt();

    for (int test = 1; test <= numTests; test++) {
      numTowns = scn.nextInt();
      numRoads = scn.nextInt();
      startTown = scn.nextInt();
      destinationTown = scn.nextInt();

      // Array of all towns.
      Vertex[] towns = new Vertex[numTowns];

      // Get Town Co-ordinates.
      for (int town = 0; town < numTowns; town++) {
        int x = scn.nextInt();
        int y = scn.nextInt();
        towns[town] = new Vertex(x, y, town);
      }
```

```
    // Get our road information.
    for (int road = 0; road < numRoads; road++) {
      int source = scn.nextInt(); // input roads
      int destination = scn.nextInt();
      towns[source].adjList.add(destination); // set towns adjacency
      towns[destination].adjList.add(source); // i.e. connected by road
    }

    // Perform Search
    Double tripDistance = aStarSearch(towns, startTown, destinationTown);

    // Output Results
    System.out.printf("Road␣Trip␣%d:␣", test);
    if (!tripDistance.isInfinite()) {
      System.out.printf("%.2f\n", tripDistance);
    } else {
      System.out.println("IMPOSSIBLE");
    }
  }
}

/**
 * Perform a search from start vertex to goal vertex, and return the cost of
 * the path
 *
 * @param vertices An Array of Vertices within the graph.
 * @param start The ID of the starting vertex.
 * @param end The ID of the goal vertex.
 * @return The weight of the path, or Double.POSITIVE_INFINITY if no path!
 */
public Double aStarSearch(Vertex[] vertices, int start, int goal) {
  PriorityQueue<Vertex> vertexQueue = new PriorityQueue<Vertex>();

  vertices[start].gScore = 0.0;
  vertices[goal].isGoal = true;
  vertexQueue.add(vertices[start]);

  while (!vertexQueue.isEmpty()) {

    Vertex current = vertexQueue.poll();
    if (current.isGoal) {
      return vertices[goal].gScore;
    }

    current.visited = true;
    vertices[current.vertexID].visited = true;

    // for all vertex adjacent to current do
    for (int n = 0; n < vertices.length; n++) {
      if (current.adjList.contains(n) && !vertices[n].visited) {
        Vertex adj = vertices[n];
        Double gScore = current.gScore + euclidianDistance(current, adj);

        // if adj not in Q or gScore < adj.gScore
        if (!vertexQueue.contains(adj) || gScore < adj.gScore) {
          // Update the Scores for the adjacent vertex.
          vertexQueue.remove(adj);
          adj.gScore = gScore;
          adj.hScore = euclidianDistance(adj, vertices[goal]);
          adj.fScore = adj.gScore + adj.hScore;
          // Add this vertex back into the priority queue with updated information.
          vertexQueue.add(adj);
        }
      }
    }
  }

  return Double.POSITIVE_INFINITY; // failed
}

/**
 * Return the distance between vertex A and vertex B, using euclidian distance
 *
```

```
   * @param vertexA The first vertex
   * @param vertexB The second vertex
   * @return The distance between to two vertices.
   */
  private Double euclidianDistance(Vertex vertexA, Vertex vertexB) {
    Double dist = Math.sqrt(
              Math.pow((double)(vertexA.xCoordinate - vertexB.xCoordinate), 2.0)
            + Math.pow((double)(vertexA.yCoordinate - vertexB.yCoordinate), 2.0));
    return dist;
  }

  /**
   * Class definition of the Vertex Information, with vertices in 2D space.
   */
  public class Vertex implements Comparable<Vertex> {

    public int xCoordinate; // Coordinate in 2D space.
    public int yCoordinate; // Coordinate in 2D space.
    public int vertexID;  // The id of the Vertex
    public Double gScore;
    public Double fScore;
    public Double hScore;
    public HashSet<Integer> adjList;
    public Boolean visited; // Has this vertex been visited.
    public Boolean isGoal; // Is this the goal vertex.

    /**
     * Default constructor.
     *
     * @param xCoord X Coordinate in 2D space.
     * @param yCoord Y Coordinate in 2D space.
     * @param entityId The id of the Vertex
     */
    public Vertex(int xCoord, int yCoord, int entityId) {
      xCoordinate = xCoord;
      yCoordinate = yCoord;
      vertexID = entityId;
      gScore = Double.POSITIVE_INFINITY;
      fScore = Double.POSITIVE_INFINITY;
      hScore = Double.POSITIVE_INFINITY;
      adjList = new HashSet<Integer>();
      visited = false;
      isGoal = false;
    }

    @Override
    public int compareTo(Vertex o) {
      if (fScore < o.fScore) {
        return -1;
      } else if (fScore > o.fScore) {
        return 1;
      } else {
        return 0;
      }
    }
  }
}
```

Listing 48: Solution to Shortest Road Trip (Java)

The few edge cases to consider would be very dense graphs, very sparse graphs, or graphs that contain zero edges. Another edge case that should be considered is the maximum weight between vertices, and ensuring that the data type that holds the weight from source to destination is capable of representing the value without overflow or even underflow. (That is when calculating the distance, we don't experience either overflow or underflow in regards when using the `pow()` or `sqrt()` methods).

6.3.7 Topological Sort

Directed graphs wihtou direct cycles are encountered in many applications. Such a diagrap is often referred to as a **directed acyclic graph** or **dag**, for short. Applications of such graphs include:

- Inheritence between C++ classes or Java Interfaces.
- Prerequistes between courses of a degree program.
- Scheduling constraints between tasks or a project.

Being able to sort a directed graph based on hierarchies or dependencies allows us to determine the order in which items may be done, or if there is a circular dependency in which we won't be able to complete our work. (That is task A relies on Task B to be completed, but Task B relies on Task A to be completed first. This is also known as the Chicken and Egg Problem).

Algorithm 25 Topological Sort Algorithm

Input A directed graph $\vec{G}$ with n vertices.
Output A topological ordering $v_1, \ldots, v_n$ of $\vec{G}$ or an indication that $\vec{G}$ has a directed cycle.

```
procedure TOPOLOGICALSORT(G⃗)
    S ← ∅                                              ▷ Let S be an empty Stack
    for all vertex u of G⃗ do
        incounter(u) ← indeg(u)
        if incounter(u) = 0 then
            S.push ← u                                 ▷ Push u onto stack S
        end if
    end for
    i ← 1
    while S is not empty do
        u ← S.pop                                      ▷ Pop Stack
        number u as the i-th vertex v_i
        i ← i + 1
        for all egde e ∈ G⃗.outIncidentEdges(u) do
            w ← G⃗.opposite(u, e)
            incounter(w) ← incounter(w)−1
            if incounter(w) = 0 then
                S.push ← w                             ▷ Push w onto stack S
            end if
        end for
    end while
    if i > n then
        return v_1, ..., v_n
    else
        return "digraph G⃗ has a directed cycle"
    end if
end procedure
```

6.3.7.1 Description of Working The alogrithm itself is based on the same principles as DFS, in that we first find all the vertices with an indegree of 0, and use these as our starting point. (We add these to a stack). The main `while()` loop runs until this stack is empty. Within the main loop, we record the current vertex we are at, and then add all vertices incident to the current that have an indegree of 0 (when the current edge between the current vertex and adjacent vertex is removed) to stack for further processing.

At the end of the algorithm, if the number of edges processed is less than the total edge count, then we have detected a directed cycle in the graph (in which case we return an error), otherwise return a vector of order in which vertices were visited.

Figure 6.22 demonstrates the algorithm in action. (a) demonstrates the initial configuration; (b-h) after each while-loop iteration. The vertex labels give the vertex number and the current incounter value. The edges tarversed in previous iterations are drawn with thick solid lines. The edges traversed in the current iteration are drawn in dotted lines.

Within the example, the topological sort order would be: {A, D, F, B, C, E, G}.

A described Topological Sort algorithm has an expected runtime complexity of **O(n + m)**.

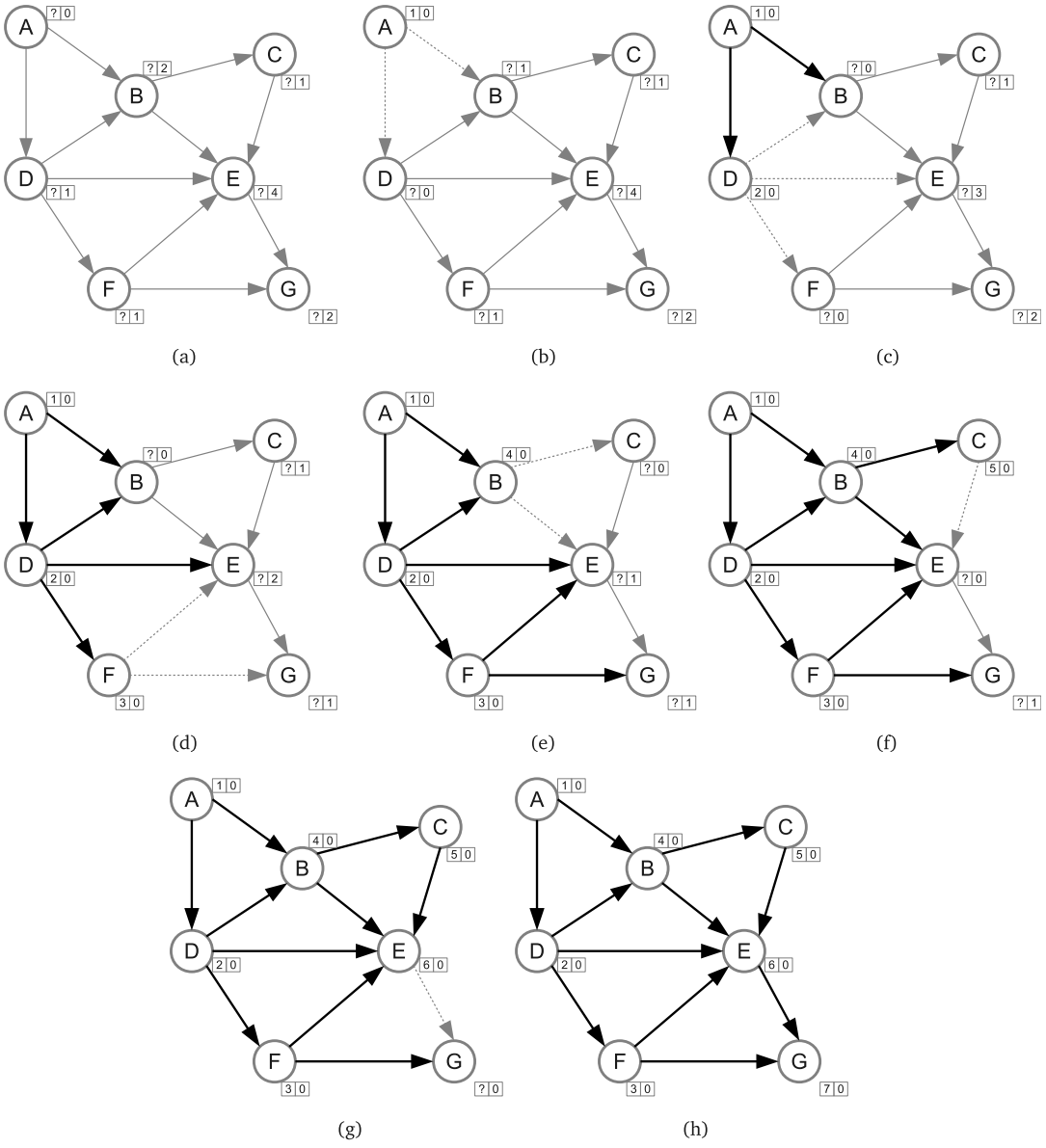


Figure 6.22: Topological Sort

6.3.7.2 Implementation An implementation of the Topological Sort Algorithm can be found in Listing 49.

```
import java.util.ArrayList;
import java.util.Stack;

public class TopologicalSort {

  /**
   * Perform a topological sort of the graph.
   * @param graph The graph to sort.
   * @return A List of vertices processed in order, or null if a cycle was detected.
   */
  public ArrayList<Vertex> TSort(ArrayList<Vertex> graph){

    Stack<Vertex> vertexStack = new Stack<Vertex>();
    ArrayList<Vertex> vertexVisitedOrder = new ArrayList<Vertex>();

    // Add all vertices with an indegree of 0 to the stack.
    for(Vertex vertex: graph){
      if(vertex.inCount == 0){
```

```
        vertexStack.push(vertex);
      }
    }

    //initialise the visited index.
    int i = 1;

    while(!vertexStack.empty()){
      // get our next vertex from the stack, and add it to the visited array.
      Vertex currentVertex = vertexStack.pop();
      currentVertex.order = i++;
      vertexVisitedOrder.add(currentVertex);

      // Get all edges from this vertex.
      for(Vertex w: currentVertex.adjList){
        w.inCount--;
        // Decrement the in degree count, and if 0, add to the processing list.
        if(w.inCount == 0){
          vertexStack.add(w);
        }
      }
    }
    // If we have processed all edges, then return the visited order array.
    if(i > graph.size()){
      return vertexVisitedOrder;
    }
    return null; // return an error.
  }

  /**
   * Class definition of the Vertex Information, with vertices in 2D space.
   */
  public class Vertex {

    public int id = 0;
    public int order = 0;
    public ArrayList<Vertex> adjList;
    public int inCount = 0;

    /**
     * Default constructor.
     */
    public Vertex(int vertexID, int indeg) {
      this.id = vertexID;
      this.inCount = indeg;
      adjList = new ArrayList<Vertex>();
    }
  }
}
```

Listing 49: Topological Sort Algorithm (Java)

6.3.7.3 Sample Problem - Spreadsheet In 1979, Dan Bricklin and Bob Frankston wrote VisiCalc, the first spreadsheet application. It became a huge success and, at that time, was the killer application for the Apple II computers. Today, spreadsheets are found on most desktop computers.

The idea behind spreadsheets is very simple, though powerful. A spreadsheet consists of a table where each cell contains either a number or a formula. A formula can compute an expression that depends on the values of other cells. Text and graphics can be added for presentation purposes.

You are to write a very simple spreadsheet application. Your program should accept several spreadsheets. Each cell of the spreadsheet contains either a numeric value (integers only) or a formula, which only support sums. If it is possible to compute all of the values of all of the formulas, then your program should output the resulting spreadsheet where all formulas have been replaced by their value.

A1	B1	C1	D1	E1	F1	...
A2	B2	C2	D2	E2	F2	...
A3	B3	C3	D3	E3	F3	...
A4	B4	C4	D4	E4	F4	...
A5	B5	C5	D5	E5	F5	...
A6	B6	C6	D6	E6	F6	...
...	...	...	...	...	...	...

Naming of the top left cells.

INPUT

The first line of the input file contains the number of spreadsheets to follow. A spreadsheet starts with a line consisting of two integer numbers, separated by a space, giving the number of columns and rows. The following lines of the spreadsheet each contain a row. A row consists of the cells of that row, separated by a single space.

A cell consists either of a numeric integer value or of a formula. A formula starts with an equal sign (=). After that, one or more cell names follow, separated by plus signs (+). The value of such a formula is the sum of all values found in the referenced cells. These cells may again contain a formula. There are no spaces within a formula.

The name of a cell consists of one to three letters for the column followed by a number between 1 and 999 (including) for the row. The letters for the column form the following series: A, B, C, ..., Z, AA, AB, AC, ..., AZ, BA, ..., BZ, CA, ..., ZZ, AAA, AAB, ..., AAZ, ABA, ..., ABZ, ACA, ..., ZZZ. These letters correspond to the number from 1 to 18278. The top left cell has the name A1 (see table above).

SAMPLE INPUT

```
2
4 3
10 34 37 =A1+B1+C1
40 17 34 =A2+B2+C2
=A1+A2 =B1+B2 =C1+C2 =D1+D2
4 3
10 34 37 =A1+B1+C1
40 17 34 =A2+B2+C2
=A1+A2 =B1+B2 =C1+C2 =D1+D2+D3
```

OUTPUT

For each test case, output the test case number followed by a colon on a line by itself. If the spreadsheet can be completely computed then, the following lines should contain the values of the cells of the spreadsheet printed in the same format as that of the input with all formulas replaced by values. If the spreadsheet cannot be completely computed because of cyclic references, then the line following the test case number should contain “Not computable!”. See the sample below.

SAMPLE OUTPUT

```
1:
10 34 37 81
40 17 34 91
50 51 71 172
2:
Not computable!
```

6.3.7.4 Sample Solution A sample solution to the Spreadsheet problem can be found in Listing 50.

```
import java.util.ArrayList;
import java.util.Scanner;
import java.util.Stack;

public class SpreadSheet {

  static Vertex[][] cells;

  public static void main(String[] args) {
    new SpreadSheet().run();
  }

  private void run() {
    int numCols;
    int numRows;


    Scanner sc = new Scanner(System.in);
    int numSheets = sc.nextInt();

    for (int sheet = 1; sheet <= numSheets; sheet++) {
      numCols = sc.nextInt();
      numRows = sc.nextInt();
      sc.nextLine();

      // Create a new spread sheet.
      cells = new Vertex[numRows][numCols];
      ArrayList<Vertex> graph = new ArrayList<Vertex>();

      // Read in our rows.
      for (int row = 0; row < numRows; row++) {
        String rowStr = sc.nextLine();
        String[] columns = rowStr.split("␣");
        for (int col = 0; col < numCols; col++) {
          if (cells[row][col] == null) {
            cells[row][col] = new Vertex(row, col, columns[col]);
          } else {
            cells[row][col].setContents(columns[col]);
          }
          // Add in all the vertices into the main graph.
          graph.add(cells[row][col]);
        }
      }

      // Get our path around the spread sheet.
      //ArrayList<Vertex> path = TopologicalSort(graph);
      System.out.printf("%d:\n", sheet);
      if (!TopologicalSort(graph)) {
        // No path found, so print we have no path.
        System.out.println("Not␣computable!");
      } else {
        // print the output.
        for (int row = 0; row < numRows; row++) {
          for (int col = 0; col < numCols; col++) {
            System.out.printf("%d␣", cells[row][col].value);
          }
          System.out.println();
        }
      }
    }
  }

  /**
   * Perform a topological sort of the graph.
   *
   * @param graph The graph to sort.
   * @return true if all edges processed and a order was found.
   */
  public boolean TopologicalSort(ArrayList<Vertex> graph) {

    Stack<Vertex> vertexStack = new Stack<Vertex>();
    //ArrayList<Vertex> vertexVisitedOrder = new ArrayList<Vertex>();
```

```
    // Add all vertices with an indegree of 0 to the stack.
    for (Vertex vertex : graph) {
      if (vertex.inCount == 0) {
        vertexStack.push(vertex);
      }
    }

    //initialise the vistied index.
    int i = 1;

    while (!vertexStack.empty()) {
      // get our next vertex from the stack, and add it to the visited array.
      Vertex currentVertex = vertexStack.pop();
      currentVertex.order = i++;
      //vertexVisitedOrder.add(currentVertex);

      // Process the dependencies immediately, rather than delaying processing
      // Based on a derived path.
      if (!currentVertex.adjList.isEmpty()) {
        // have dependencies, so calculate
        currentVertex.calculate();
      }
      // Get all edges from this vertex.
      for (Cell cell : currentVertex.adjList) {
        Vertex w = cells[cell.row][cell.column];
        w.inCount--;
        // Decrement the in degree count, and if 0, add to the processing list.
        if (w.inCount == 0) {
          vertexStack.add(w);
        }
      }
    }
    // If we have processed all edges, then return the visited order array.
    if (i > graph.size()) {
      return true; // We found a parh, and processed all edges
    }
    return false; // return an error.
  }

  /**
   * Basic class to hold the location of a Cell (used in the adjacent list).
   */
  public class Cell {

    public int row = 0;
    public int column = 0;

    /**
     * Basic constructor to build via row, col values.
     *
     * @param row
     * @param col
     */
    public Cell(int row, int col) {
      this.row = row;
      this.column = col;
    }

    /**
     * Basic constructor to build via a spreadsheet cell reference.
     *
     * @param cellID The reference to this cell in A0 format.
     */
    public Cell(String cellID) {
      setAddress(cellID);
    }

    /**
     * Set the cell address via the spreadsheet cell reference format.
     *
     * @param address
     */
    private void setAddress(String address) {
```

```
    int index = 0;
    // Find the first number in the address reference
    for (int cindex = 0; cindex < address.length(); cindex++) {
      if (Character.isLetter(address.charAt(cindex))) {
        index = cindex + 1;
      }
    }

    // create a new array of char to hold the alpha part of the address.
    char[] colAddress = new char[index];
    address.getChars(0, index, colAddress, 0);

    // Convert both to column and row references.
    column = convertToColNum(colAddress);
    row = Integer.parseInt(address.substring(index)) - 1;
  }

  /**
   * Convert a column Alphabet reference to a zero indexed number.
   *
   * @param column array of char to convert to number format.
   * @return
   */
  private int convertToColNum(char[] column) {
    int colAddress = -1;
    String alpha = "_ABCDEFGHIJKLMNOPQRSTUVWXYZ"; // A = 1, Z = 26

    for (int c = 0; c < column.length; c++) {
      colAddress += alpha.indexOf(column[c]) * Math.pow(26, column.length - c - 1);
    }

    return colAddress;
  }
}

/**
 * Class definition of the Vertex Information, with vertices in 2D space.
 */
public class Vertex {

  public int order = 0;
  public ArrayList<Cell> adjList;
  public int inCount = 0;
  public int value = 0;
  public int row;
  public int col;

  /**
   * Default constructor.
   */
  public Vertex(int row, int col) {
    this.row = row;
    this.col = col;
    this.inCount = 0;
    adjList = new ArrayList<Cell>();
  }

  /**
   * Default Constructor.
   *
   * @param row Row of this vertex
   * @param col Column of this vertex
   * @param contents The contents of this cell.
   */
  public Vertex(int row, int col, String contents) {
    this.row = row;
    this.col = col;
    this.inCount = 0;
    adjList = new ArrayList<Cell>();
    setContents(contents);
  }

  /**
```

```
     * Set the value to be contained in this cell
     *
     * @param contents
     */
    public final void setContents(String contents) {
      // Attempt to get a plain number.
      try {
        value = Integer.parseInt(contents);
      } catch (NumberFormatException e) {
        // Must have a formula.
        String f = contents.substring(1); //remove leading "="
        String[] elements = f.split("\\+");
        // Add each element in the formula.
        for (String item : elements) {
          Cell element = new Cell(item);
          // See if our reference already exists?
          if (cells[element.row][element.column] == null) {
            // create a new vertex.
            cells[element.row][element.column] = new Vertex(element.row, element.column);
          }
          // Add that we have a cell that we reference.
          this.inCount++;
          // Let the cell we reference, know that we reference it.
          // (add us to it's adjacency list).
          cells[element.row][element.column].adjList.add(new Cell(this.row, this.col));
        }
      }
    }

    /**
     * Update the value of vertex
     */
    public void calculate() {
      // Update the value of the cells, on our adjacent list with our value.
      for (Cell element : adjList) {
        cells[element.row][element.column].value += this.value;
      }
    }
  }
}
```

Listing 50: Solution to Spreadsheet Problem (Java)

When utilising a topological sort within the spreadsheet problem, there are few issues with the problem.

Test cases to consider would be a spreadsheet with a single cell, a cell that is self referencing (that is, in cell A1, contains the formula “=A1”).

In development of a solution, special care should be taken in regards to development of the method that creates a row, column reference from the spread sheet style reference of A1. Input test cases should include both A1, and ZZZ999, ensuring that the row count is calculated correctly. Utilisation of regular expressions could help in determine the start of the number in the cell reference, however consideration for the penalities on using the regular expression engine, verses perfoming a linear search for at most 3 characters should also be considered.

Another test case to be considered is a cell’s formula that may have multiple references to the same cell, for example “=A1+A1+...+A1”, and that the number of cells referenced in the formula in essentially unlimited in length.

The solution as presented as a slight modification to the main algorithm, in that rather than adding the current vertex being operated on to the sorted path, we simple process the dependency calculation in place. This was done to save cycling through the path list later on, when it can be done in place in the middle of the algorithm.

6.4 Graph Theory (Advanced)

6.4.1 Maximal Cardinality Bipartite Matching

A problem that arise in a number of important applications is the maximum bipartite matching problem. In this problem, we are given a connected undirected graph with the following properties:

- The vertices of G are partitioned into two sets, X and Y.
- Every edge of G has one endpoint in X and the endpoint in Y.

Such a graph is called a bipartite graph. A matching in G is a set of edges that have no endpoints in common - such a set of "pairs" of vertices in X with vertices in Y so that each vertex has at most one "partner" in the other set. The maximum bipartite matching problem is to find a matching with the greatest number of edges (over all matches). Figure 6.23 shows a valid bipartite graph.

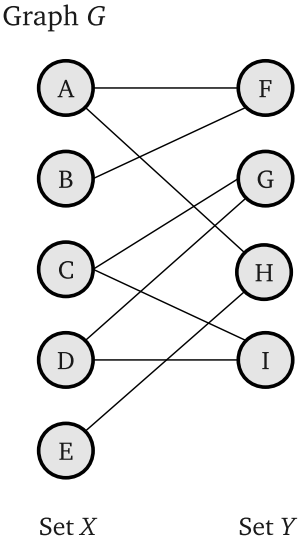


Figure 6.23: Bipartite Graph

Sample problems in which bipartite graphs and related algorithms are useful include:

- Singles Agency - From a group of n women and m single men who wish to be in a relationship, with edges denoting acceptability of partnership, attempt to maximise the total number of partnerships that can be made.
- Task Scheduling - From a group of employees, matched to skills, attempt to maximise the number of skills that can be utilised at one time.
- Classroom assignment - From a group of study units, and a group of rooms attempt to maximise the number of rooms utilised at one time which enables the maximum number of concurrent study units.

The matching between vertices from each set, can be considered maximal is we cannot add any edge to the existing set M without violating the properties of matching.

Algorithm 26 Maximum Bipartite Matching Algorithm

Input A bipartite graph G, consisting of X vertices on the left hand side and Y vertices on the right hand side.
Output The cardinality of the graph G

```
procedure MAXIMUMBIPARTITEMATCHING(S)
    C ← 0                                          ▷ Set cardinately to 0
    for all src in X do                            ▷ For each source in left side vertices
        for all ver in Y do                        ▷ For each vertex in right side
            ver.visited ← false
        end for
        if FINDAUGMENTINGPATH(src) then
            C = C + 1
        end if
    end for
    return C
end procedure
```

Algorithm 27 Maximum Bipartite Matching Find Augmenting Path Algorithm

Input A source vertex v
Output True if a path exists, otherwise false

```
procedure FINDAUGMENTINGPATH(v)
    for all d in v adjency list do                 ▷ For all destinations in v adjacency list
        if d not visited then
            d.visited ← true
            if d is unmatched ∨ FINDAUGMENTINGPATH(d current match) then
                d and v are matched                ▷ Augment function
                return true
            end if
        end if
    end for
    return false
end procedure
```

6.4.1.1 Description of Working The Maximum Bipartite Matching Algorithm as shown requires the following data structures to operate:

- Graph
  - List of all vertices in G.
  - N vertices in left side $(0 \dots N-1)$
  - M vertices in right side $(0 \dots M-1)$.
- Vertices
  - Identifier
  - Visited
  - Matched
  - Matched with
  - Adjacency List

The algorithm works on the concept of alternating paths and determining augmenting paths, in that we continue to develop augmenting paths until no paths exist. (Each time we have an augmented path, we increase the cardinality of the graph).

The Algorithm itself is based on two parts, the main algorithm and the algorithm to find augmented paths. The main algorithm simple cycles through all vertices in the left side X, and attempts to find augmented paths, until such none exist. Each time a path is found, the cardinality is incremented.

The key algorithm is the FindAugmentingPath, which utilises a DFS (Depth First Search) to find vertices that are unmatched or have themselves augmented paths.

Figure 6.24 demonstrates this algorithm in detail.

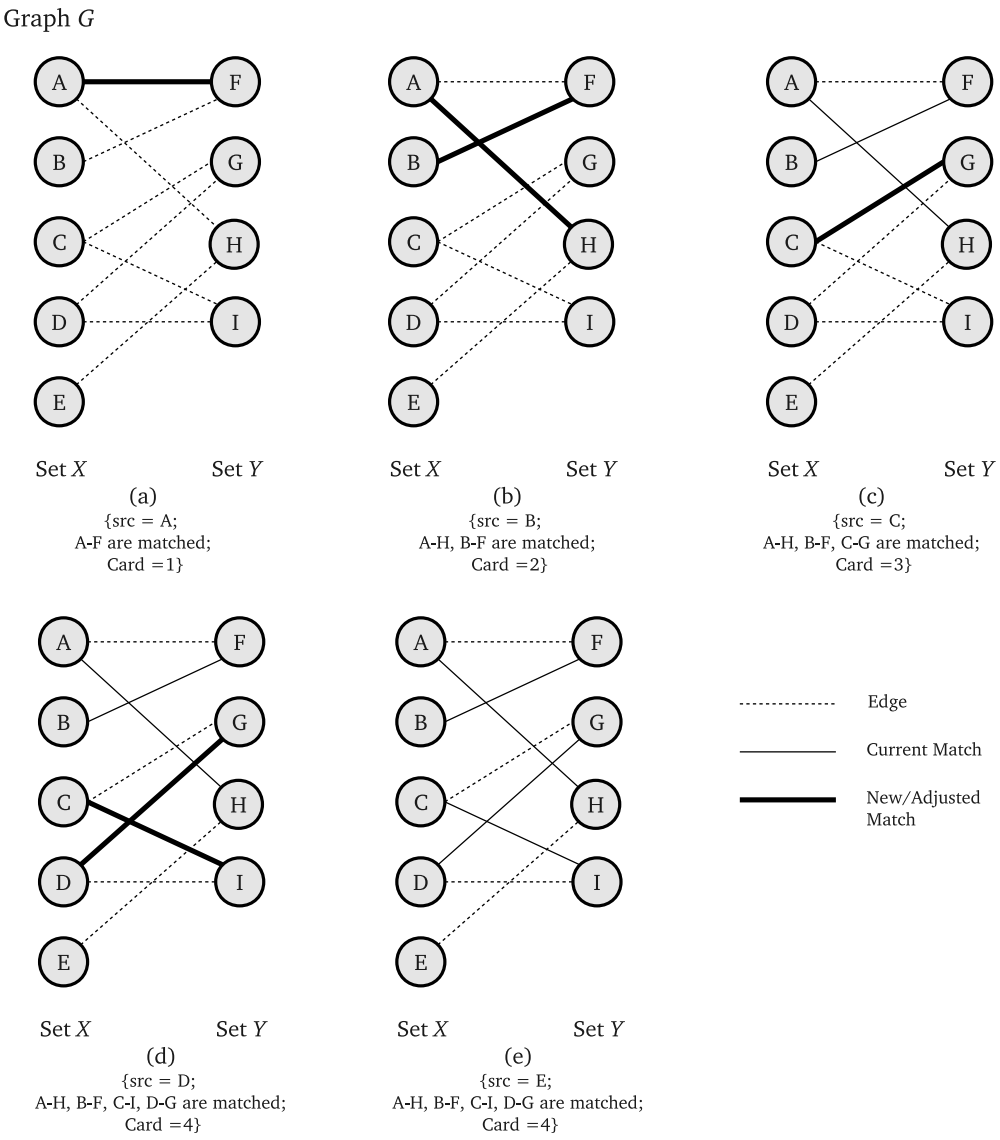


Figure 6.24: Maximum Bipartite Matching Algorithm

In Figure 6.24;

(a), the first iteration of the main loop utilised vertex A as the source, and simply takes the first vertex as a match (being vertex F).

(b), the second iteration of the main loop utilised vertex B as the source, and itself takes vertex F as it's match, as vertex A has an alternative path (being to vertex H). As an augmented path exists, the cardinality is now 2.

(c), the third iteration of the main loop utilises vertex C as the source, and simply takes G as it's match. As G is not matched with anything else, searching for augmented paths doesn't take place.

(d), the fourth iteration of the main loop utilises vertex D as the source, and itself takes vertex G as it's match, as vertex C has an alternative path (being to vertex I). Whilst it is possible for vertex D to match

to I and C to G, the algorithm itself works in a linear path through the adjacency list, and attempts to take the first vertex (if possible) for itself. (In this case the Adjacency List for vertex D is vertex G, vertex I in order).

(e), the final iteration of the main loop utilised vertex E as the source. It does not take vertex H as it's match, as vertex A which is currently matched to vertex H, has no alternative paths. (So it's not possible to steal that match for it's as it could possible decrement the current cardinality value).

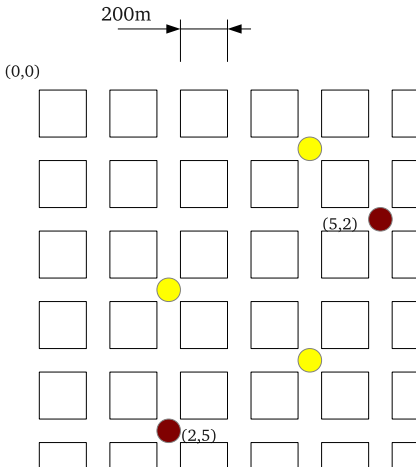
The described Maximum Bipartite Matching Algorithm is found in $\mathbf{O(|V| \times |E|)}$ time.

6.4.1.2 Implementation An implementation of the Maximum Bipartite Matching Algorithm can be found in Listing 51. (A full list of supporting classes can be found in the Problem Solution Listing 52).

```
import java.util.ArrayList;

public class MaximumBipartiteMatching {

  /**
   * Find the cardinality of Graph G, with set X and Y being left and right sides.
   * @param graphX Left hand side graph
   * @param graphY Right hand side graph
   * @return The cardinality of Graph G.
   */
  public int MaximumBipartiteMatching(ArrayList<Vertex> graphX, ArrayList<Vertex> graphY) {
    int cardinality = 0;
    for (Vertex vertexX : graphX) {
      for (Vertex vertexY : graphY) {
        vertexY.visited = false;  // Set all right side to not visited.
      }
      if (FindAugmentingPath(vertexX)) {  // If a path exists...
        cardinality++;
      }
    }
    return cardinality;
  }

  /**
   * Determine is an augmented path exist from vertex.
   * @param vertex The source vertex.
   * @return True, If a path exists
   */
  public boolean FindAugmentingPath(Vertex vertex) {
    // Process all vertices in the adjacency list.
    for (Vertex vertexD : vertex.adjList) {
      if (!vertexD.visited) { // Only query vertex if not already visited.
        vertexD.visited = true;
        if (vertexD.matched == null || FindAugmentingPath(vertexD.matched)) {
          // d and v are now matched and we have a new path.
          vertex.matched = vertexD;
          vertexD.matched = vertex;
          return true;
        }
      }
    }
    return false; // no path...
  }
}
```

Listing 51: Maximum Bipartite Matching Algorithm (Java)

6.4.1.3 Sample Problem - Taxi Party town is a well planned town. Each town block is 200m x 200m and all of the streets are built as straight lines that intersect at right angles (see diagram).



Although the town is well designed and very pretty, the best thing about Party Town is that the inhabitants love to party! Every month they hold a "mystery venue" party. The venue for the party is only announced a short time before the party starts. The "mystery venue" parties tend to fill up very quickly, so everyone wants to get there as soon as possible. To give everyone a chance, the venue is announced on the radio just before the party starts, so everyone tries to get there as quickly as possible.

This is all great news for the Party Town Taxi Service. As soon as the party venue is announced on the radio, people all over town ring up for a taxi to take them to the party. The problem for the taxi service is that a lot of people ask for a ride at the same time and the taxis have to pick up the people quickly.

You have been asked to help the Party Town Taxi Service figure out how many passengers they can take to the party. You must take into account the following constraints:

- Each taxi can only take one passenger
- Passengers always wait at intersections of roads
- At the time of the radio announcement, all taxis are also waiting at intersections
- The taxi has to reach the passenger within a given time limit (or they will be too late to get into the party)

INPUT

The first line contains the number of test cases K ($K \leq 250$). The first line of each test case contains the number of persons $P(1 \leq P \leq 400)$, the number of taxis $T(1 \leq T \leq 200)$ the speed $S(1 \leq S \leq 2000)$ of the taxis in meters per second and the time limit C to collect a person in seconds $(1 \leq C \leq 1000000)$. The next P lines contain the positions of the persons. The next T lines contain the position of the taxis in the city.

SAMPLE INPUT

```
1
2 3 10 40
2 5
5 2
2 3
4 1
4 4
```

OUTPUT

For each test case, output the test case number followed by a colon and a space and then the maximum number of persons who can get to the party.

SAMPLE OUTPUT

```
1:  2
```

6.4.1.4 Sample Solution

A sample solution to the Taxi problem can be found in Listing 52.

```
import java.util.ArrayList;
import java.util.Scanner;

public class Taxi {

  final int BLOCK_SIZE = 200; // meters
  ArrayList<Vertex> taxis; // Graph X
  ArrayList<Vertex> people; // Grpah Y
  int numPeople;
  int numTaxis;
  int taxiSpeed;
  int timeToCollect;

  /**
   * Main
   */
  public static void main(String[] args) {
    new Taxi().run();
  }

  /**
   * Main runner.
   */
  public void run() {
    Scanner sc = new Scanner(System.in);

    int numCases = sc.nextInt(); // Number of test cases
    for (int ncase = 1; ncase <= numCases; ncase++) {
      numPeople = sc.nextInt(); // number of persons
      numTaxis = sc.nextInt(); // number of taxis
      taxiSpeed = sc.nextInt(); // speed of taxis (m/s)
      timeToCollect = sc.nextInt(); // time limit to collect (seconds)

      taxis = new ArrayList<Vertex>();
      people = new ArrayList<Vertex>();

      // read in our people locations
      while (numPeople-- > 0) {
        int x = sc.nextInt();
        int y = sc.nextInt();
        people.add(new Vertex(x, y));
      }

      // read in our taxi locations
      while (numTaxis-- > 0) {
        int x = sc.nextInt();
        int y = sc.nextInt();
        taxis.add(new Vertex(x, y));
      }
      // Create edges between people and taxis, if the taxi is able to service the person.
      createEdges();

      // Output the cardinality
      System.out.printf("%d:␣%d%n", ncase, MaximumBipartiteMatching(taxis, people));
    }
  }

  /**
   * Create edges between taxis and people, if the taxi is able to service a
   * person.
   */
  void createEdges() {
    for (Vertex taxi : taxis) {
      for (Vertex person : people) {
        // If the travel time from the taxi to person is below the required tim
        // then the taxi can service the person.
        if (travelTime(taxi, person) <= timeToCollect) {
          // Taxi is able to service this person so add an edge between them
          taxi.adjList.add(person);
          person.adjList.add(taxi);
        }
      }
    }
  }

  /**
   * Determine the time needed to move from source to destination.
   *
   * @param source The source location.
   * @param destination The destination location.
   * @return the time needed
   */
  int travelTime(Vertex source, Vertex destination) {
    return manhattanDist(source, destination) * BLOCK_SIZE / taxiSpeed;
  }
```

```
  /**
   * Calculate the number of blocks to traverse in Manhattan Distance.
   *
   * @param source The source location.
   * @param destination The destination location.
   * @return The number of blocks to traverse.
   */
  int manhattanDist(Vertex source, Vertex destination) {
    return Math.abs(source.x - destination.x) + Math.abs(source.y - destination.y);
  }

  /**
   * Find the cardinality of Graph G, with set X and Y being left and right
   * sides.
   *
   * @param graphX Left hand side graph
   * @param graphY Right hand side graph
   * @return The cardinality of Graph G.
   */
  public int MaximumBipartiteMatching(ArrayList<Vertex> graphX, ArrayList<Vertex> graphY) {
    int cardinality = 0;
    for (Vertex vertexX : graphX) {
      for (Vertex vertexY : graphY) {
        vertexY.visited = false;  // Set all right side to not visited.
      }
      if (FindAugmentingPath(vertexX)) {  // If a path exists...
        cardinality++;
      }
    }
    return cardinality;
  }

  /**
   * Determine is an augmented path exist from vertex.
   *
   * @param vertex The source vertex.
   * @return True, If a path exists
   */
  public boolean FindAugmentingPath(Vertex vertex) {
    // Process all vertices in the adjacency list.
    for (Vertex vertexD : vertex.adjList) {
      if (!vertexD.visited) { // Only query vertex if not already visited.
        vertexD.visited = true;
        if (vertexD.matched == null || FindAugmentingPath(vertexD.matched)) {
          // d and v are now matched and we have a new path.
          vertex.matched = vertexD;
          vertexD.matched = vertex;
          return true;
        }
      }
    }
    return false; // no path...
  }

  /**
   * Class definition of the Vertex Information, with vertices in 2D space.
   */
  public class Vertex {

    public int x;
    public int y;
    public String id;
    public boolean visited = false;
    public Vertex matched = null;
    public ArrayList<Vertex> adjList;

    /**
     * Default constructor.
     */
    public Vertex(int x, int y) {
      this.x = x;
      this.y = y;
      adjList = new ArrayList<Vertex>();
    }

    /**
     * Default constructor.
     */
    public Vertex(String id) {
      this.id = id;
      adjList = new ArrayList<Vertex>();
    }
  }
}
```

Listing 52: Solution to Taxi Problem (Java)

The solution to this problem is rather simple as it primarily relies on the implementation of the algorithm itself to produce a correct result. Some sample cases that must be considered are:

- No taxi is able to pick up any person, due to either time or distance constraints. (This results in zero egdes between each side).
- There are multiple people at a single intersection.
- There is a large difference between the number of taxis and people.
- A taxi and person are at the same location.

However there is no requirement for explicit testing of these conditions within the algorithm, as the algorithm and implementation naturally handles these conditions.

6.4.2 Network Flow (minimum cost, maximum flow)

An important problem involving weighted graphs is the **maximum flow** problem. In this problem we are given a weighted graph G, with each edge representing a "pipe" that can transport some commodity, with the weight of the dge representing the maximum amount it can transport. The maximum-flow problem problem is to find a way of transporting the maximum amount of the given commodity from some **source** to the destination (called the **sink**). (The source has an in degree of 0, and the sink has an out degree of 0).

A classic example is a computer network, in which links between different hosts/nodes can vary in line speed. In order to determine the maximum bandwidth between two hosts on opposite sides of the network we can use an algorithm to determine this. Figure 6.25 demonstrates such a network, with the source being Vertex D, and the Sink being Vertex G.

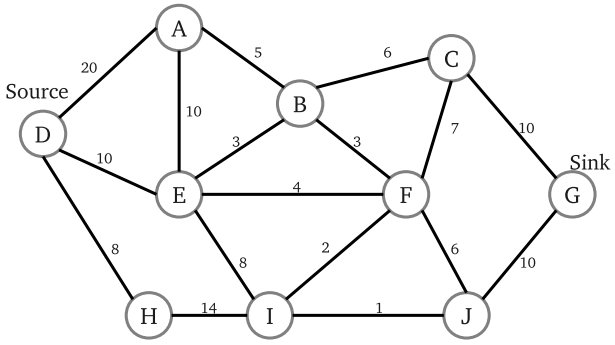


Figure 6.25: Maximum Flow Basic Graph

In Figure 6.25, whilst Vertex D has a pipes outward worth 38, and Vetrex G has pipes feeding in worth 20, the maximum flow is only 15!

It turns out that flows are closely related to another concept, known as cuts. Intuitively, a cut is a division of the vertices of a flow network (such as one in Figure 6.25) into two sets, with source on one side and the sink on the other. Formally, a **cut** of N is a partition $\chi = (V_s, V_t)$ of vertices of N such that $s \in V_s$ and $s \in V_t$. If we can determine a minimum cut for a network, that a cut is found that the cost of the edges through the cut is minimum, then we can determine the maximum flow. Figure 6.26 shows the minimum cut, that also defines the maximum flow of the network. (The sum of all pipe capacities along the cut will give the minimum cut, which is equal to the maximum flow, which in this case is 15).

In order to prove that a certain flow f is maximum (other than visually as done by looking at Figure 6.26), we need some way of showing that there is absolutely no more flow that can be possibly be squeezed into f. This can be done by using two concepts of residual capacity and augmented paths.

Residual capacity is basically any left over capacity not being current utilised along an edge. An augmented path for flow f is a path from the source to the sink with non-zero residual capacity, and

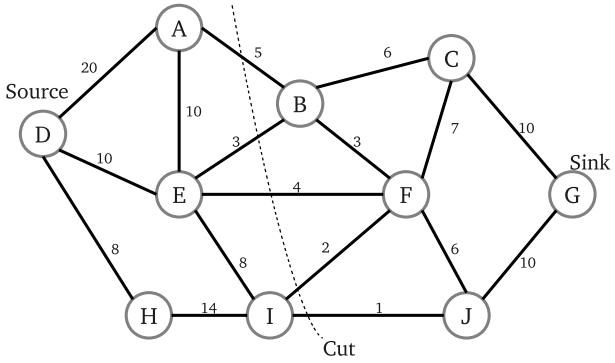


Figure 6.26: Maximum Flow with Cut

when no augmented paths exist, we can determine the maximum flow. (This is a very similar concept to that used to solve the Maximum Bipartite Matching problem[29]). It should also be noted that many graphs may actually have different maximum flows f of the same maximum capacity, but each involve a different set of edges to reach the maximum flow capacity.

The **Ford Fulkerson** algorithm is used to solve the maximum flow problem. The algorithm relies on finding augmenting paths in the underlying residual network. The residual network can be thought of as having forward edges, the original edges in the network, and backward edges once a flow is placed on a particular edge. Augmenting paths from the source to the sink can be found by finding a path in the residual network by using the forward and backward edges. Once a path has been found, the edge with the minimum capacity defines the amount that the flow can be augmented. Forward edges in the underlying residual network then have their capacity reduced and backward edges have their capacity increased.

The augmenting path for determining the maximum flow can be found using any graph traversal algorithm – depth first search, breadth first search or priority first search, where priority is given to the maximum capacity of the adjacent egdes. Breadth first search is quite often used as it is simple to implement and produces the shortest path based on the number of edges in the path.

While it is very useful to solve the Maximum Flow problem, many real life examples also associate a cost to utilising some resource (or edge as modelled in a graph). This may be the cost of using a WAN service, the cost of a toll road, the cost of shipping via a certain company or the cost of using one employee over another to perform a task [30].

The Minimum Cost Maximum Flow algorithm is designed to determine the minimum cost needed for a given set of maximum flows f for graph N. That is, what is the minimum cost needed to achieve the maximum flow through a network.

6.4.2.1 Description of Working

6.4.2.2 Implementation An implementation of the Minimum Cost Maximum Flow Algorithm can be found in Listing

6.4.2.3 Sample Problem - Minimum Cost Maximum Flow Given a network which has a single source vertex with no in edges and a single destination (sink) vertex with no out edges, compute the minimum cost for the maximum flow of the network.

INPUT

[29] We can also use one of the algorithms to solve the Maximum Flow problem to also solve the Maximum Bipartite Matching problem. To utilise this method we introduced a new source vertex to the left side graph (which is connected to all vertices on the left hand side, and a sink vertex to the right side graph (which is connected to all vertices on the right hand side), and then treat the maximum bipartite matching problem as a maximum flow problem.

[30] When using an algorithm for Minimum Cost, Maximum Flow to solve a Maximum Bipartite Matching problem

Algorithm 28 Minimum Cost Maximum Flow Algorithm

Input A graph G, source vertex s, sink vertex t.
Output The minimum cost.

```
procedure MINCOSTFLOWEDMONDSKARP(G, s, t)
    costs ← ∞                                          ▷ Set all costs to infinity
    Set all diagonal costs ← 0                         ▷ set all costs[i][i] = 0
    repeat
        pathCapacity ←FINDAUGMENTINGPATH(G, s, t)
    until pathCapacity > 0
    return CALCULATEMINIMUMCOST(flow, costs)
end procedure
```

Algorithm 29 Minimum Cost Maximum Flow Find Augmenting Path Algorithm

Input A graph G, source vertex s, sink vertex t.
Output The minimum cost.

```
procedure FINDAUGMENTINGPATH(G, s, t)
    return 0
end procedure
```

Algorithm 30 Minimum Cost Maximum Flow Calculate Minimum Cost Algorithm

Input A $flow$ matrix and a $cost$ matrix.
Output The minimum cost.

```
procedure CALCULATEMINIMUMCOST(flow, cost)
    c ← 0                                              ▷ Set total cost to 0
    for all positive values in flow do
        c ← c + flow[i][j] × cost[i][j]
    end for
    return c
end procedure
```

The first line contains the number of test cases $T (T \leq 250)$.

The first line of each test case contains 4 integers – N, E, S and D. N is the number of vertices in the network, $2 \leq N \leq 1000$. E is the number of directed edges in the network, $1 \leq E \leq 10,000$. S is the source vertex of the network, $0 \leq S < N$. D is the sink vertex of the network , $0 \leq D < N$. There is a further constraint that $S \neq D$.

The next E lines contain four integers – V_i, V_j, Cap, $Cost$ – where V_i is the source vertex of the directed edge, V_j is the destination vertex of the directed edge, Cap is the capacity of the directed edge and $Cost$ is the cost for the edge. $1 \leq Cap \leq 10,000$. $1 \leq Cost \leq 1,000$.

SAMPLE INPUT

```
2
4 4 0 3
0 1 10 5
1 3 10 5
0 2 10 2
2 3 10 2
7 8 0 6
0 1 3 10
0 2 1 12
1 3 3 5
2 3 5 1
2 4 4 4
3 6 2 10
4 5 2 6
5 6 3 2
```

OUTPUT

For each test case, output the message "`Test x:  Minimum Cost = y`", where x is the test case number and y is the minimum cost of the maximum flow for the given network. Test case numbers start at 1.

SAMPLE OUTPUT

```
Test 1:  Minimum Cost = 140
Test 2:  Minimum Cost = 74
```

6.4.2.4 Sample Solution A sample solution to the Minimum Cost Maximum Flow problem can be found in Listing

6.5 Computational Geometery

6.5.1 Line Segment Intersection

Determination if two line segments intersect is a key component for many math related problems found in computational geometery. The fields of application include:

- 2D and 3D graphics, determining if two lines intersect, primarily for game logic, entity mapping, path boundaries, etc.
- Locations where roads or paths cross when determining wildlife paths, which may require fencing or some form of safe travel path for wildlife.

To determine if two line segments intersection requires the use of some mathematics and a quick glossary of terms.

A line segment, when plotted on a Cartesian graph has a start location $P_1 = (x_1, y_1)$ and an end location $P_2 = (x_2, y_2)$, and the line (or vector) itself is defined as $L_1 = (P_1, P_2)$. To determine if two line segments as shown in Figure 6.27, requires some transformation of the problem.

As can be seen in Figure 6.27, there are 2 line segments, $L_1 = (P_1, P_2)$ and $L_2 = (P_3, P_4)$. In order to determine the point in which they intersect, we define the line-segments by their end points.

$$P_a = P_1 + U_a(P_2 - P_1)$$
$$P_b = P_3 + U_b(P_4 - P_3)$$

Notice that if you put in zero for U, you'll get the start point, if you put in one, you'll get the end point. Also the point of intersection would be where they are equal. That is:

$$P_a = P_b$$

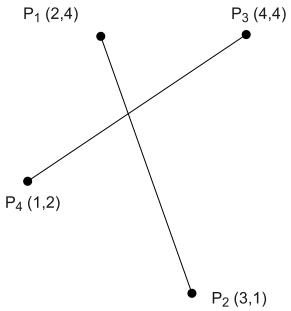


Figure 6.27: Line Intersection

$$P_1 + U_a(P_2 - P_1) = P_3 + U_b(P_4 - P_3)$$

However since we need this in terms of x and y, we can rewrite that last equation as the following two:

$$x_1 + U_a(x_2 - x_1) = x_3 + U_b(x_4 - x_3)$$

$$y_1 + U_a(y_2 - y_1) = y_3 + U_b(y_4 - y_3)$$

We can use those to now solve for U.

$$U_a = \frac{(x_4 - x_3)(y_1 - y_3) - (y_4 - y_3)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)}$$

$$U_b = \frac{(x_2 - x_1)(y_1 - y_3) - (y_2 - y_1)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)}$$

Notice that the denominator on both of those equations is the same. Solve for that first, if it is zero, then the lines are parallel and we're done. (If both numerators are also zero, then the two line segments are coincident.)

Since these equations treat the lines as infinitely long lines instead of line segments we want, there is guaranteed to be an intersection point if the lines aren't parallel. To determine if it happens in the segments we've specified, we need to see if U is between zero and one. Verify that both of these are true:

$$0 \leq U_a \leq 1$$

$$0 \leq U_b \leq 1$$

If we've gotten this far, then our line segments intersect, and we just need to find the point at which they do and then we're done.

$$x = x_1 + U_a(x_2 - x_1)$$

$$y = y_1 + U_a(y_2 - y_1)$$

A source code implementation of the above math can be found in Listing 53.

```
public class LineSegmentIntersection {

  /**
   * Generic class to hold a point.
   */
  public class Point {

    public double x = 0.0;
    public double y = 0.0;
  }

  /**
   * Determine if 2 lines intersect, and provide intersection point.
   *
   * @param Line1 The vector of the first line
   * @param Line2 The vector of the second line
   * @return Point if there is an intersection, otherwise null.
   */
  public Point GetLineIntersection(Point[] Line1, Point[] Line2) {
    Point result = new Point();
    // Precalculate some points;
    double x4_x3 = (Line2[1].x - Line2[0].x);
    double x2_x1 = (Line1[1].x - Line1[0].x);
    double y4_y3 = (Line2[1].y - Line2[0].y);
    double y2_y1 = (Line1[1].y - Line1[0].y);
    double x1_x3 = (Line1[0].x - Line2[0].x);
    double y1_y3 = (Line1[0].y - Line2[0].y);

    // Get the demonitator
    double dem = (y4_y3 * x2_x1) - (x4_x3 * y2_y1);

    // Determine U values.
    double Ua = ((x4_x3 * y1_y3) - (y4_y3 * x1_x3)) / dem;
    double Ub = ((x2_x1 * y1_y3) - (y2_y1 * x1_x3)) / dem;

    // Ensure our U values are in scope
    if (Ua < 0 || Ua > 1 || Ub < 0 || Ub > 1) {
      return null;
    }

    // Calculate the intersection point.
    result.x = Line1[0].x + Ua * (x2_x1);
    result.y = Line1[0].y + Ua * (y2_y1);

    if (Double.isNaN(result.x) || Double.isNaN(result.y)) {
      return null;
    }

    return result;
  }
}
```

Listing 53: Line Intersection (Java)

With this method, we can determine if 2 lines intersect. However if we choose to determine in n-line segments intersect there are a number of approaches, with a naive approach would simply compare all lines with all other lines. This unfortunately would yield $\mathbf{O(n^2)}$ performance. A more efficient method would be to use a Sweep Line Algorithm.

6.5.1.1 Description of Working

The Bentley-Ottmann Line Sweep Algorithm utilises an event queue of points to determine if two line segments should be tested if they have an intersection or not, thereby reducing the number of intersection tests required.

This algorithm involves simulating the sweeping of a vertical line l, over the segments moving from left to right, starting at a location to the left of all input segments. Initially a sorted queue of all endpoints is maintainted, which forms the event queue of points to test.

During the sweep, the set of segments currently intersected by the sweep line is maintained by means of insertions into and removals from a binary search tree. When a new segment end point is encountered, the segment relating to the end point is either added or removed from the binary search tree, thus maintaining the list of segments intersecting with the sweep line. During either the insertion/removal operation a test of all segments that intersect the sweep line are tested to determine if they intersection, and if they do the intersection point is added to the event queue. If an intersection is encounterd during the sweep, the intersection is added to the known Intersection List, and the event is removed from the queue.

Algorithm 31 Bentley-Ottmann Line Sweep Algorithm

Input A set of line Segments S
Output A list of intersection points.

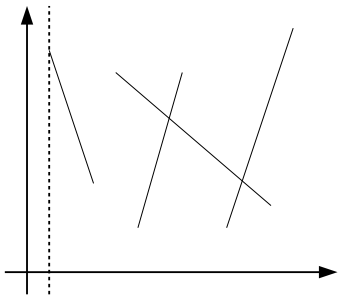
```
procedure BENTLEYOTTMANNLINESWEEP(S)
    I ← ∅                                          ▷ Let I be an empty List of Intersections
    SL ← ∅                                         ▷ Let SL be an empty set of the sweep line
    T ← S                                          ▷ Initialise Event Queue of all segment endpoints
    Sort T by increasing x and y values of points
    while T ≠ ∅ do
        E ← next event from T
        if E is a left endpoint then
            E_seg ← E segment
            SL ← E_seg                             ▷ Add E_seg to SL
            A_seg ← segment above E_seg in SL
            B_seg ← segment below E_seg in SL
            if Intersection between A_seg and E_seg exists then
                T ← Intersection (A_seg, E_seg)    ▷ Insert the intersection into T
            end if
            if Intersection between B_seg and E_seg exists then
                T ← Intersection (B_seg, E_seg)    ▷ Insert the intersection into T
            end if
        else if E is a right endpoint then
            E_seg ← E segment
            A_seg ← segment above E_seg in SL
            B_seg ← segment below E_seg in SL
            Remove E_seg from SL
            if Intersection between A_seg and B_seg exists then
                if Intersection(B_seg, E_seg) not in T then
                    T ← Intersection (A_seg, B_seg)    ▷ Insert the intersection into T
                end if
            end if
        else                                       ▷ E is an Intersection Event
            I ← E
            Let E_seg1 above E_seg2 be E's intersection segments in SL
            Swap E_seg1 and E_seg2, so E_seg2 is above E_seg1
            A_seg ← segment above E_seg2 in SL
            B_seg ← segment below E_seg1 in SL
            if Intersection between E_seg2 and A_seg exists then
                if Intersection(E_seg2, A_seg) not in T then
                    T ← Intersection (E_seg2, A_seg)   ▷ Insert the intersection into T
                end if
            end if
            if Intersection between E_seg1 and B_seg exists then
                if Intersection(E_seg1, B_seg) not in T then
                    T ← Intersection (E_seg1, B_seg)   ▷ Insert the intersection into T
                end if
            end if
        end if
        Remove E from T
    end while
    return I
end procedure
```

Figure 6.28 demonstrates the line sweep in action.

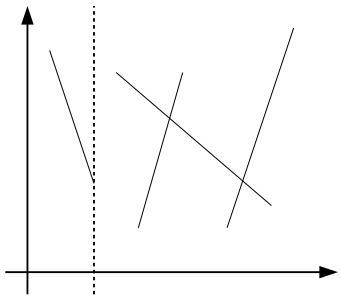
The Bentley-Ottmann Algorithm when used with a balanced binary search tree data structure such as an AVL tree or Red-Black Tree, offers **O(n log n)** performance, as insertions and deletions from the tree data

structure are **O(log n)** operations.

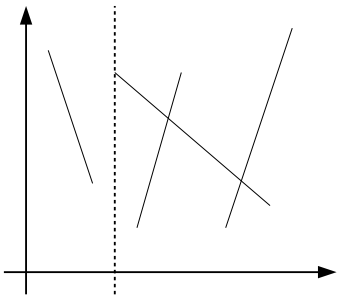
6.5.1.2 Implementation An implementation of a Line Sweep Algorithm can be found in Listing 54. (A full list of supporting classes can be found in the Problem Solution Listing 55).



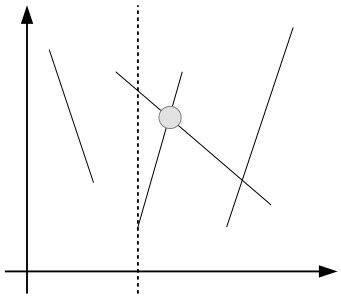
(a) Point 1 is encountered, intersections tested, (none found).



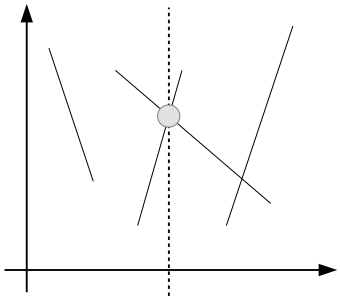
(b) Point 2 is encountered, intersections tested (none found)



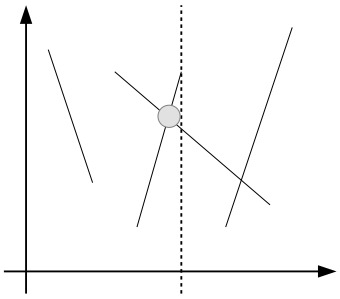
(c) Point 3 is encountered, intersections tested (none found)



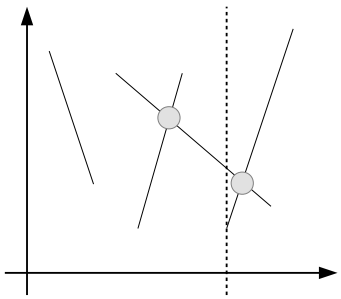
(d) Point 4 is encountered, intersections tested (intersection found, so added to event queue)



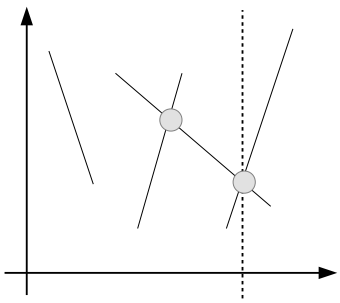
(e) Intersection is encountered, intersections added to known list.



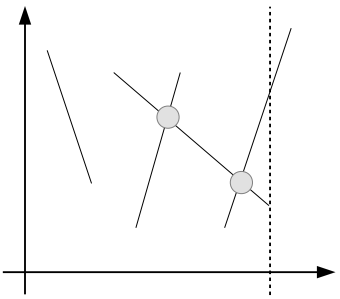
(f) Point 5 encountered, line segment removed from SL.



(g) Point 6 encountered, intersections tested (intersection found, so added to event queue).



(h) Intersection is encountered, intersections added to known list.



(i) Point 7 encountered, line segment removed from SL.

Figure 6.28: Line Sweep Algorithm

```
public class LineSegmentIntersection {
  /**
   * Perform a line sweep on the enclosed segments.
   *
   * @param segments A list of segments to consider.
   * @return A list of points of intersection.
   */
  public List<Point> BentleyOttmann(List<Segment> segments) {
    ArrayList<Point> intersections = new ArrayList<Point>();
    PriorityQueue<Event> eventQueue = new PriorityQueue<Event>();
    Sweepline sweepline = new Sweepline();

    // Add all our endpoints to the event queue.
    for (Segment segment : segments) {
      eventQueue.add(new Event(segment, segment.start));
      eventQueue.add(new Event(segment, segment.end));
    }
    // Continue while we have item in queue.
    while (!eventQueue.isEmpty()) {
      Event event = eventQueue.poll();  // Get our next event.
      if (event.segment == null) {
        // Intersection.
        // Add the point to the known intersections.
        intersections.add(event.point);
        sweepline.resort();
        // Get the segment above and below the current event point.
        Segment segE1 = sweepline.above(event.point);
        Segment segE2 = sweepline.below(event.point);
        Segment segA = sweepline.prev(segE1);
        Segment segB = sweepline.next(segE2);
        if (segA != null && segE2 != null) {
          addIntersection(eventQueue, segA, segE2);
        }
        if (segB != null && segE1 != null) {
          addIntersection(eventQueue, segB, segE1);
        }
      } else if (event.point == event.segment.start) {
        // event is a line start.
        eventTrigger = event.point.x;
        sweepline.add(event.segment);
        Segment segE = event.segment;
        Segment segA = sweepline.prev(segE);
        Segment segB = sweepline.next(segE);
        if (segA != null) {
          addIntersection(eventQueue, segA, segE);
        }
        if (segB != null) {
          addIntersection(eventQueue, segB, segE);
        }
      } else {
        // event is a line end
        eventTrigger = event.point.x;
        Segment segE = event.segment;
        Segment segA = sweepline.prev(segE);
        Segment segB = sweepline.next(segE);
        sweepline.remove(event.segment);
        if (segA != null && segB != null) {
          addIntersection(eventQueue, segA, segB);
        }
      }
    }
    return intersections;
  }

  /**
   * Add in intersection to the event queue, if an intersection exists between 2
   * segments.
   *
   * @param eventQueue The event queue.
   * @param segA The first segment
   * @param segB The second segment
   */
  public void addIntersection(PriorityQueue<Event> eventQueue, Segment segA, Segment segB) {
    Point segBIntersection = GetLineIntersection(segA, segB);
    if (segBIntersection != null) {
      Event segBEvent = new Event(null, segBIntersection);
      if (!eventQueue.contains(segBEvent)) {
        eventQueue.add(segBEvent);
      }
    }
  }
}
```

Listing 54: Line Sweep Algorithm (Java)

6.5.1.3 Sample Problem - n-Line Segment Intersection A metal cutter cuts polygons into a sheet of metal by cutting line segments. Each line segment corresponds to an edge of the polygon. Find the co-ordinates of the corners of the resulting polygon.

INPUT

The input consists of several test cases. Each line segment is listed on a line by itself as 4 positive integers: the x and y coordinates of one endpoint followed by the x and y coordinates of the other endpoint. After the last line segment in the test case, there is the character # on a line by itself. After the last test case there is another character # on a line by itself. You are guaranteed that, in each test case, each line segment intersects exactly two other line segments.

SAMPLE INPUT

```
2 4 4 5
1 6 4 2
8 3 3 5
6 1 7 4
7 2 3 3
#
#
```

OUTPUT

For each test case, output the test case number on a line, the number of corners, and then for each corner a single line with the x and y coordinates, rounded to two decimal places. The corner points should be sorted in increasing order of x coordinates; if multiple points have the same x coordinate, list those points in increasing order of y coordinates.

SAMPLE OUTPUT

```
Test Case 1:
5 corners
2.36 4.18
3.31 2.92
3.56 4.78
6.38 2.15
6.82 3.47
```

6.5.1.4 Sample Solution A sample solution to the n-Line Intersection problem can be found in Listing 55.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.ListIterator;
import java.util.PriorityQueue;
import java.util.Scanner;

public class LineSegmentIntersection {

  /**
   * Main
   */
  public static void main(String[] args) {
    new LineSegmentIntersection().run();
  }

  /**
   * Main runner.
   */
  public void run() {
    int testcase = 1;
    ArrayList<Segment> segments;

    Scanner scn = new Scanner(System.in);
    String line = scn.nextLine();

    // Keep looping until we get a # by itself on a line.
    while (line.compareTo("#") != 0) {
      segments = new ArrayList<Segment>();
      // Read in our segments.
```

```
    while (line.compareTo("#") != 0) {
      String[] values = line.split("␣");
      Point start = new Point();
      Point end = new Point();
      start.x = Double.parseDouble(values[0]);
      start.y = Double.parseDouble(values[1]);
      end.x = Double.parseDouble(values[2]);
      end.y = Double.parseDouble(values[3]);
      segments.add(new Segment(start, end));
      line = scn.nextLine();
    }
    // Process the segments
    //List<Point> intersections = Naive(segments);
    List<Point> intersections = BentleyOttmann(segments);
    Collections.sort(intersections); // Sort the points.

    // Output the corner information.
    System.out.printf("Test␣Case␣%d:\n%d␣corners\n", testcase++, intersections.size());
    for (Point p : intersections) {
      System.out.println(p.toString());
    }
    line = scn.nextLine();
  }
}

/**
 * Naive approach to finding intersections from n-segments.
 *
 * @param segments A list of segments to consider.
 * @return A list of points of intersection.
 */
public List<Point> Naive(List<Segment> segments) {
  ArrayList<Point> intersections = new ArrayList<Point>();
  for (int i = 0; i < segments.size(); i++) {
    for (int j = 0; j < segments.size(); j++) {
      Point intersection = GetLineIntersection(segments.get(i), segments.get(j));
      if (intersection != null && !intersections.contains(intersection)) {
        intersections.add(intersection);
      }
    }
  }
  return intersections;
}

/**
 * Perform a line sweep on the enclosed segments.
 *
 * @param segments A list of segments to consider.
 * @return A list of points of intersection.
 */
public List<Point> BentleyOttmann(List<Segment> segments) {
  ArrayList<Point> intersections = new ArrayList<Point>();
  PriorityQueue<Event> eventQueue = new PriorityQueue<Event>();
  Sweepline sweepline = new Sweepline();

  // Add all our endpoints to the event queue.
  for (Segment segment : segments) {
    eventQueue.add(new Event(segment, segment.start));
    eventQueue.add(new Event(segment, segment.end));
  }

  // Continue while we have item in queue.
  while (!eventQueue.isEmpty()) {
    Event event = eventQueue.poll();  // Get our next event.

    if (event.segment == null) {
      // Intersection.
      // Add the point to the known intersections.
      intersections.add(event.point);
      sweepline.resort();
      // Get the segment above and below the current event point.
      Segment segE1 = sweepline.above(event.point);
      Segment segE2 = sweepline.below(event.point);
      Segment segA = sweepline.prev(segE1);
      Segment segB = sweepline.next(segE2);
      if (segA != null && segE2 != null) {
        addIntersection(eventQueue, segA, segE2);
      }
      if (segB != null && segE1 != null) {
        addIntersection(eventQueue, segB, segE1);
      }

    } else if (event.point == event.segment.start) {
      // event is a line start.
      eventTrigger = event.point.x;
      sweepline.add(event.segment);
      Segment segE = event.segment;
```

```
      Segment segA = sweepline.prev(segE);
      Segment segB = sweepline.next(segE);

      if (segA != null) {
        addIntersection(eventQueue, segA, segE);
      }
      if (segB != null) {
        addIntersection(eventQueue, segB, segE);
      }

    } else {
      // event is a line end
      eventTrigger = event.point.x;
      Segment segE = event.segment;
      Segment segA = sweepline.prev(segE);
      Segment segB = sweepline.next(segE);
      sweepline.remove(event.segment);
      if (segA != null && segB != null) {
        addIntersection(eventQueue, segA, segB);
      }
    }
  }
  return intersections;
}

/**
 * Add in intersection to the event queue, if an intersection exists between 2
 * segments.
 *
 * @param eventQueue The event queue.
 * @param segA The first segment
 * @param segB The second segment
 */
public void addIntersection(PriorityQueue<Event> eventQueue, Segment segA, Segment segB) {
  Point segBIntersection = GetLineIntersection(segA, segB);
  if (segBIntersection != null) {
    Event segBEvent = new Event(null, segBIntersection);
    if (!eventQueue.contains(segBEvent)) {
      eventQueue.add(segBEvent);
    }
  }
}

/**
 * Generic class to hold an event to be used by BentleyOttmann algorithm.
 */
public class Event implements Comparable<Event> {

  public Point point;
  public Segment segment;

  /**
   * Constructor
   *
   * @param seg The Segment related to the event.
   * @param pt The point which will trigger the event.
   */
  public Event(Segment seg, Point pt) {
    point = pt;
    segment = seg;
  }

  /**
   * Comparable method, that compares the event point.
   */
  @Override
  public int compareTo(Event o) {
    return point.compareTo(o.point);
  }
}

/**
 * Generic class to hold a line segment.
 */
public class Segment implements Comparable<Segment> {

  public Point start;
  public Point end;

  /**
   * Main Constructor. Note: Points are sorted based on x location, so that
   * the start point is always to the left of the end point.
   *
   * @param start The start point
   * @param end The end point
   */
  public Segment(Point start, Point end) {
```

```
    if (end.compareTo(start) < 0) {
      // swap them.
      this.start = end;
      this.end = start;
      return;
    }
    this.start = start;
    this.end = end;
  }

  /**
   * Comparable method, that compares the event point y location at
   * intersection with the sweep line
   */
  @Override
  public int compareTo(Segment o2) {
    // Compare y positions of the segment at the x co-ordinate of the
    // event trigger.
    double seg1 = getY(this);
    double seg2 = getY(o2);

    if (seg1 > seg2) {
      return 1;
    } else if (seg1 < seg2) {
      return -1;
    }
    // Same location, so lets determine by slope.
    seg1 = getSlope(this);
    seg2 = getSlope(o2);
    if (seg1 > seg2) {
      return 1;
    } else if (seg1 < seg2) {
      return -1;
    }
    return 0; // Same y and same slope!
  }

  @Override
  public String toString() {
    return String.format("\"S:␣%.0f,␣%.0f,␣E:␣%.0f,␣%.0f\"", start.x, start.y, end.x, end.y);
  }
}

/**
 * Get the y intersection of a segment to the sweep line.
 *
 * @param segment The segment to test against.
 * @return The y location.
 */
public double getY(Segment segment) {
  // Determine slope of segment.
  double slope02 = getSlope(segment);
  // Determine y of segment
  return slope02 * eventTrigger + (segment.start.y - slope02 * segment.start.x);
}

/**
 * Get the slope of the segment.
 *
 * @param segment The segment whose slope is needed.
 * @return The slope of the segment.
 */
public double getSlope(Segment segment) {
  return (segment.end.y - segment.start.y) / (segment.end.x - segment.start.x);
}
/**
 * X location of the event trigger (the location of the sweep line).
 */
private static double eventTrigger;

/**
 * Generic class to hold the segments that the sweep line intersects. This
 * should use a AVL/Red-Black Tree (like the Java TreeMap class), but the Java
 * TreeMap class won't resort the entire tree on insertion / delete operation.
 */
public class Sweepline {

  private ArrayList<Segment> sweepline;

  public Sweepline() {
    sweepline = new ArrayList<Segment>();
  }

  /**
   * Add segment to the list
   */
  public void add(Segment segment) {
```

```
    sweepline.add(segment);
    Collections.sort(sweepline);
  }

  /**
   * Remove a segment from the list
   */
  public void remove(Segment segment) {
    sweepline.remove(segment);
    Collections.sort(sweepline);
  }

  /**
   * Get the entire list of segments in the sweepline.
   *
   * @return
   */
  public List<Segment> getLine() {
    return sweepline;
  }

  /**
   * Get the segment next to the current segment, or null if none exist.
   */
  public Segment next(Segment segment) {
    if (segment == null) {
      return null;
    }
    int index = sweepline.indexOf(segment);
    if (index == -1 || index == sweepline.size() - 1) {
      return null;
    }
    return sweepline.get(index + 1);
  }

  /**
   * Get the segment previous to the current segment, or null is none exist.
   */
  public Segment prev(Segment segment) {
    if (segment == null) {
      return null;
    }
    int index = sweepline.indexOf(segment);
    if (index == -1 || index == 0) {
      return null;
    }
    return sweepline.get(index - 1);
  }

  /**
   * Find the segment above the current point.
   *
   * @param point
   * @return
   */
  public Segment above(Point point) {
    if (point == null) {
      return null;
    }
    for (Segment seg : sweepline) {
      double y = getY(seg);
      if (y == point.y) {
        return seg;
      }
    }
    return null;
  }

  /**
   * Find the segment below the current point.
   *
   * @param point
   * @return
   */
  public Segment below(Point point) {
    if (point == null) {
      return null;
    }
    for (ListIterator iterator = sweepline.listIterator(sweepline.size()); iterator.hasPrevious();) {
      final Segment seg = (Segment) iterator.previous();
      double y = getY(seg);
      if (y == point.y) {
        return seg;
      }
    }
    return null;
  }
```

```
    /**
     * Resort the segments in the sweepline.
     */
    public void resort() {
      Collections.sort(sweepline);
    }
  }

  /**
   * Generic class to hold a point
   */
  public class Point implements Comparable<Point> {

    public double x = 0.0;
    public double y = 0.0;

    @Override
    public int compareTo(Point o) {
      if (o.x == this.x) {
        double temp = o.y - this.y;
        if (temp < 0) {
          return 1;
        } else if (temp == 0) {
          return 0;
        }
        return -1;
      }
      double temp = o.x - this.x;
      if (temp < 0) {
        return 1;
      } else if (temp == 0) {
        return 0;
      }
      return -1;
    }

    @Override
    public String toString() {
      return String.format("%.2f␣%.2f", x, y);
    }
  }

  /**
   * Determine if 2 lines intersect, and provide intersection point.
   *
   * @param Line1 The vector of the first line
   * @param Line2 The vector of the second line
   * @return Point if there is an intersection, otherwise null.
   */
  public Point GetLineIntersection(Segment Line1, Segment Line2) {
    Point result = new Point();
    // Precalculate some points;
    double x4_x3 = (Line2.end.x - Line2.start.x);
    double x2_x1 = (Line1.end.x - Line1.start.x);
    double y4_y3 = (Line2.end.y - Line2.start.y);
    double y2_y1 = (Line1.end.y - Line1.start.y);
    double x1_x3 = (Line1.start.x - Line2.start.x);
    double y1_y3 = (Line1.start.y - Line2.start.y);

    // Get the demonitator
    double dem = (y4_y3 * x2_x1) - (x4_x3 * y2_y1);

    // Determine U values.
    double Ua = ((x4_x3 * y1_y3) - (y4_y3 * x1_x3)) / dem;
    double Ub = ((x2_x1 * y1_y3) - (y2_y1 * x1_x3)) / dem;

    // Ensure our U values are in scope
    if (Ua < 0 || Ua > 1 || Ub < 0 || Ub > 1) {
      return null;
    }

    // Calculate the intersection point.
    result.x = Line1.start.x + Ua * (x2_x1);
    result.y = Line1.start.y + Ua * (y2_y1);

    if (Double.isNaN(result.x) || Double.isNaN(result.y)) {
      return null;
    }

    return result;
  }
}
```

Listing 55: Solution to n-Line Segment Intersection (Java)

The solution itself does not offer a true implementation of the Bentley-Ottmann Algorithm, but a slightly modified version in that the datastructure that holds the segments that currently intersect with the sweep line is a Sorted List (based on the y-intersection of a segment with the current sweepline position), and not a Balanced Binary Search Tree. This does result in some performance penalty as insertions, and deletions are **O(n log n)** and get operations are **O(n)**, as opposed to **O(log n)** if a balanced binary search tree is used. The reason for this modification is that any balanced binary search tree implementation to be effective with this algorithm must be able to work with mutable key values and resort the entire tree based on the mutable keys[31]. (The keys in the tree are the y-intersection position of the segment against the current x-position of the sweep line). This was done as a trade off against implementation complexity and performance complexity. The solution offered does however perform faster than or equal to **O(n^2)** naive approach in most circumstances despite this modification. Table 6 demonstrates some benchmarks as conducted on the authors system, (note: intersection output was enabled during these tests). This highlights that when there are few segments that intersecti with the sweepline, the modified version runs significantly faster, only matching the naive approach when all segments intersect the sweepline at a single time.

	Line Segments	**1,000**	**10,000**	**100,000**
Single segments intersect with sweepline	Naive	210 msec	2107 msec	~
	Bentley-Ottmann	99 msec	210 msec	2009 msec
3 Segments intersect with sweepline	Naive	210 msec	2107 msec	~
	Bentley-Ottmann	164 msec	710 msec	11725 msec
All Segments intersect with sweepline	Naive	400 msec	5012 msec	~
	Bentley-Ottmann	420 msec	5500 msec	~

(Notes: The times are only recorded if the test completes in under 5 minutes, the first test has no intersections, the second test has $\frac{n}{2}$-intersections, and the third test has $2 \times n$-intersections, and output to console is maintained).

Table 6: Bentley-Ottmann vs Naive

As mentioned, the key variable with any implementation of this algorithm is the data structure used to hold the segments that currently intersect with the sweep line, as the rest of the algorithm is rather simple in structure. Any data strucutre must be either sortable on a mutable key value, or offer rapid lookups for next and prev segments based on a mutable key.

The solution to this problem is rather simple as it primarily relies on the implementation of the algorithm itself to produce a correct result. The primary case to handle is that there is no or only a single segment exists in the list of segments to be processed.

However there is no requirement for explicit testing of these conditions within the algorithm, as the algorithm naturally handles these conditions.

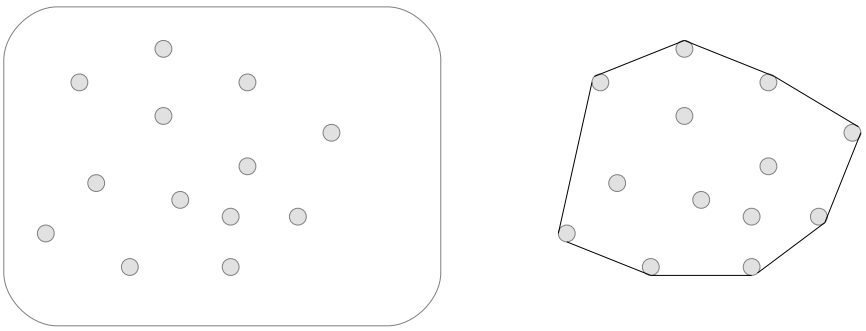
6.5.2 Convex Hull

The convex null corresponds to the intuitive notion of a boundary of a set of points and can be used to approximate the shape of a complex object. Informally speaking, the convex hull of a set of points in the plane is a shape taken by a rubber band that is placed “around the points” and allowed to shrink to a state of equilibrium. (See Figure 6.29).

Indeed, computing the convex hull of a set of points is a fundamental operation in computational geometery.

Applications of the related algorithms include area calculations, the length of the boundary of the object (which is useful for “gift wrapping”, or calculating the material required to cover an object), object culling in 3D/2D graphics for non-retangular shapes.

[31] As far as the author is aware, implementations of balanced binary search trees that come with system standard libraries assume the key utilised is immutable.



(a) (b)

The convex hull of a set of points in the plane: (a) an example "rubber band" placed around a set of points; (b) the covex hull of the points.

Figure 6.29: Convex Hull

Algorithm 32 Graham Scan Algorithm

Input A list S of points in the plane beginning with point a, such that a is one the convex hull of S and the remaining points of S are sorted counterclockwise around a

Output List S with only convex hull vertices remaining.

```
procedure GRAHAMSCAN(S, a)
    S.insertLast(a)                          ▷ Add a copy of A at the end of S
    prev ← S.first()                         ▷ So that prev = a initially
    curr ← S.after(prev)                     ▷ the next point is on the current vertex chain
    repeat
        next ← S.after(curr)                 ▷ advance
        if points (point(prev), point(curr), point(next)) make a left turn then
            prev ← curr
        else
            S.remove(curr)                   ▷ point curr is not in the convex hull
            curr ← S.before(prev)
        end if
        curr ← S.after(prev)
    until curr = S.last()
    S.remove(S.last())                       ▷ Remove the copy of a
end procedure
```

6.5.2.1 Description of Working The Graham Scan Algorithm for computing the convex hull H of a set P of n points in the plane consists of the following four phases:

1. We find a point a of P and call this the anchor point. Typically this will be the point with the minium y value (and minimum x value if a tie).

2. We sort the remaining points in P using a radial comparison, so that all points are sorted in counterclockwise direction from a. (See Figure 6.30).

3. We call the Scan function, which scans through all points in S in radial order, maintaining at each step a list of H storing a convex chain "surrounding" the points scanned so far. Each time we consider a new point p:

   (a) If p form a left turn with the last two points in H, or if H contains fewer than 2 points, then add p to the end of H.

   (b) Otherwise, remove the last point in H and repeat the test for p.

4. We stop when when we reach the anchor point a, at which point H sotres the verives of the convex hull in counterclockwise order.

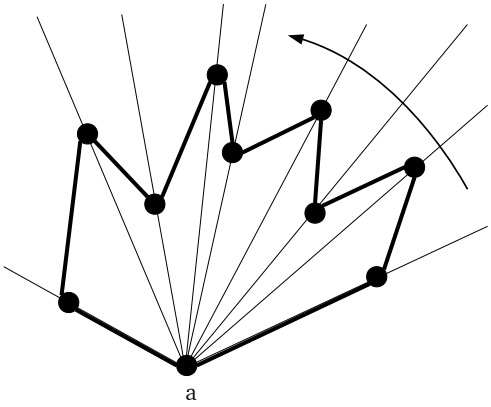


Figure 6.30: Graham Scan - Sorting around the anchor point

Figure 6.31 demonstrates the third phase of the Graham Scan Algorithm.

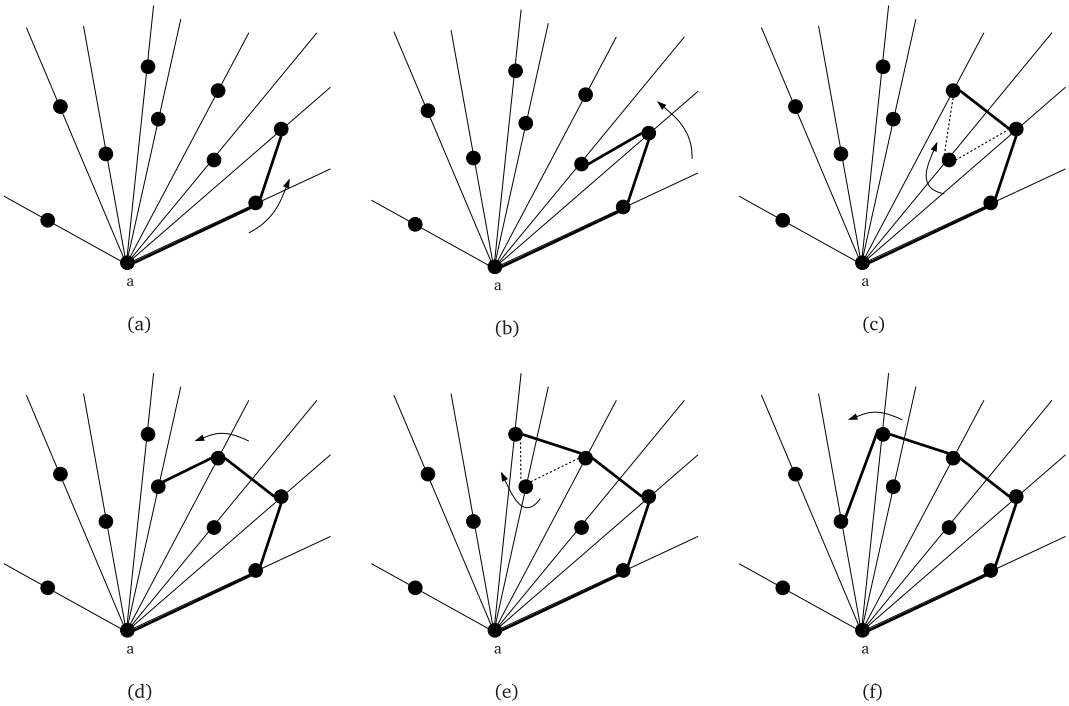


Figure 6.31: Graham Scan - Demonstration of Scan Operation

The Graham Scan Algorithm has been shown to operate in **O(n log n)** time.

6.5.2.2 Implementation An implementation of the Graham Scan Algorithm can be found in Listing 56. (A full list of supporting classes can be found in the Problem Solution Listing 57).

```
import java.util.ArrayList;
import java.util.PriorityQueue;
import java.util.Stack;

public class ConvexHull {
```

```
  ArrayList<Point> points;  // general list of points in the set.
  PriorityQueue<Point> pointsQueue; // The points sorted by radial angle from the anchor.
  Stack<Point> convexHull; // The resultant convex hull
  Point anchor; // The anchor point.

  /**
   * Perform a scan of the points queue, building the convex hull.
   */
  private void grahamScan() {
    convexHull.push(pointsQueue.poll()); // Add our anchor point to the convex hull
    convexHull.push(pointsQueue.poll()); // Add our next point to the convex hull
    while (!pointsQueue.isEmpty()) {
      Point pt2 = convexHull.pop(); // Get our prev point
      Point pt1 = convexHull.peek();  // Get our curr point
      convexHull.push(pt2);
      Point pt3 = pointsQueue.poll(); // Get out next point
      if (turnsLeft(pt1, pt2, pt3)) {
        convexHull.push(pt3); // Add our last point since we turned left
      } else {
        pointsQueue.add(pt3); // Add our just tested point back into the queue
        convexHull.pop(); // Get rid of the last point as we turned right.
      }
    }
  }

  /**
   * Determines if there is a left turn between 2 points.
   * This is done by using the difference in the cross product of points.
   */
  private boolean turnsLeft(Point p1, Point p2, Point p3) {
    int result = (p3.x - p2.x) * (p1.y - p2.y) - (p3.y - p2.y) * (p1.x - p2.x);
    return (result > 0);
  }
}
```

Listing 56: Graham Scan Algorithm (Java)

The implementation described uses a Priority Queue to hold points to be processed (based on radial angle from the anchor) and a Stack of points that form the convex hull. This makes it easy add and remove points as needed from both data structures, especially doe to the ordered nature of the priority queue.

The "`turnsLeft()`" method only needs to determine if we turned left or not, and not the angle, which allows us to determine this based on the cross product of the points in question.

6.5.2.3 Sample Problem - Convex Hull Tomato plants are planted in the ground at various points in a garden. We wish to build a single continuous fence around the tomato plants at minimal cost. Each linear metre of fence costs $5; posts cost $1. You may assume that you do not need to leave any buffer zone around a tomato plant. You may also assume that the area enclosed by the fence will always be non-zero.

INPUT

The first line of input will be a number on a line by itself which is the number of test cases to run. For each test case, the first line will be a number $N \geq 3$ on a line by itself which is the number of tomato plants. The next N lines consists of a pair of non-negative integers, which are the x and y coordinates of the end points of the line segments.

SAMPLE INPUT

```
2
7
5 4
4 2
3 3
6 1
7 4
2 1
1 3
4
6 2
6 3
6 4
5 1
```

OUTPUT

For each test case, output a single line containing the cost of building the fence, rounded to two decimal places.

SAMPLE OUTPUT

```
$82.61
$35.88
```

6.5.2.4 Sample Solution A sample solution to the Convex Hull problem can be found in Listing 57.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Scanner;
import java.util.Stack;

public class ConvexHull {

  private double costPost = 1.0;
  private double costFencePerM = 5.0;
  ArrayList<Point> points;  // general list of points in the set.
  PriorityQueue<Point> pointsQueue; // The points sorted by radial angle from the anchor.
  Stack<Point> convexHull; // The resultant convex hull
  Point anchor; // The anchor point.

  /**
   * Main
   */
  public static void main(String[] args) {
    new ConvexHull().run();
  }

  /**
   * Main runner.
   */
  public void run() {
    Scanner scn = new Scanner(System.in);
    int numberOfTestCases = scn.nextInt();

    while (numberOfTestCases-- > 0) {

      int numberOfPoint = scn.nextInt();
      points = new ArrayList<Point>(numberOfPoint);
      pointsQueue = new PriorityQueue<Point>(numberOfPoint);
      convexHull = new Stack<Point>();

      // read in all our points.
      while (numberOfPoint-- > 0) {
        points.add(new Point(scn.nextInt(), scn.nextInt()));
      }

      // Sort our list of points, based on y and x values;
      Collections.sort(points, new PointSort());
      anchor = points.get(0); // Get the start of the list, this is our anchor.

      // Add the points to the priority queue.
      for (Point point : points) {
        pointsQueue.add(point); // These are sorted based on angle between anchor and point.
      }

      // Don't bother building a convex hull with 3 or less points.
      if (points.size() > 3) {
        grahamScan();
      } else {
        // Just add the points to the convex hull...
        while (!pointsQueue.isEmpty()) {
          convexHull.push(pointsQueue.poll());
        }
      }
      System.out.printf("$%.2f\n", calculateFenceCost());
    }
  }

  /**
   * Perform a scan of the points queue, building the convex hull.
   */
  private void grahamScan() {
    convexHull.push(pointsQueue.poll()); // Add our anchor point to the convex hull
    convexHull.push(pointsQueue.poll()); // Add our next point to the convex hull
    while (!pointsQueue.isEmpty()) {
```

```
    Point pt2 = convexHull.pop(); // Get our prev point
    Point pt1 = convexHull.peek();  // Get our curr point
    convexHull.push(pt2);
    Point pt3 = pointsQueue.poll(); // Get out next point
    //System.out.printf("Test Points: (%s), (%s), (%s)\n", pt1, pt2, pt3);
    if (turnsLeft(pt1, pt2, pt3)) {
      convexHull.push(pt3); // Add our last point since we turned left
    } else {
      pointsQueue.add(pt3); // Add our just tested point back into the queue
      convexHull.pop(); // Get rid of the last point as we turned right.
    }
  }
}

/**
 * Generic class to hold a point
 */
public class Point implements Comparable<Point> {

  public int x = 0;
  public int y = 0;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  /**
   * Comparable function based on radial angle from the anchor.
   */
  @Override
  public int compareTo(Point point) {
    // Determine the angle from the anchor to this vs point.
    double anglePoint = getAngle(point);
    double angleThis = getAngle(this);
    if (anglePoint > angleThis) {
      return -1;
    } else if (anglePoint < angleThis) {
      return 1;
    }
    // Angles are the SAME! so determine by distance from anchor.
    anglePoint = vectorLength(findVector(anchor, point));
    angleThis = vectorLength(findVector(anchor, this));
    if (anglePoint > angleThis) {
      return -1;
    } else if (anglePoint < angleThis) {
      return 1;
    }
    return 0;
  }

  @Override
  public String toString() {
    return String.format("%d␣%d", x, y);
  }

  /**
   * Return the angle between the anchor (as origin) and point.
   */
  private double getAngle(Point point) {
    return Math.atan2(point.y - anchor.y, point.x - anchor.x);
  }
}

/**
 * Generic Comparator to sort a list by y, and x values.
 */
private class PointSort implements Comparator<Point> {

  @Override
  public int compare(Point point1, Point point2) {
    if (point1.y < point2.y) {
      return -1;
    } else if (point1.y > point2.y) {
      return 1;
    } else if (point1.x < point2.x) {
      return -1;
    } else if (point1.x > point2.x) {
      return 1;
    }
    return 0;
  }
}

/**
 * Determine the vector from point 1 to point 2.
 */
```

```
  private Point findVector(Point point1, Point point2) {
    return new Point(point2.x - point1.x, point2.y - point1.y);
  }

  /**
   * Determine the vector length from origin.
   */
  private double vectorLength(Point point) {
    return Math.sqrt(Math.pow(point.x, 2) + Math.pow(point.y, 2));
  }

  /**
   * Determines if there is a left turn between 2 points. This is done by using
   * the difference in the cross product of all points.
   */
  private boolean turnsLeft(Point p1, Point p2, Point p3) {
    int result = (p3.x - p2.x) * (p1.y - p2.y) - (p3.y - p2.y) * (p1.x - p2.x);
    return (result > 0);
  }

  /**
   * Calculate the cost of the fence around the convex hull.
   *
   * @return
   */
  double calculateFenceCost() {
    double numPosts = convexHull.size();
    double metersOfFence = 0;

    Point first = convexHull.peek();
    Point previous = convexHull.pop();

    // Work around the convex hull and determine the length of the fence.
    while (!convexHull.isEmpty()) {
      Point next = (Point) convexHull.pop();
      metersOfFence += vectorLength(findVector(previous, next));
      previous = next;
    }

    // Add in the final section of fence (from the last point back to anchor).
    metersOfFence += vectorLength(findVector(previous, first)); // join fence back to start

    // Return the cost.
    return (costPost * numPosts) + (costFencePerM * metersOfFence);
  }
}
```

Listing 57: Solution to Convex Hull Problem (Java)

The solution to this problem is rather simple as it primarily relies on the implementation of the algorithm itself to produce a correct result.

Some sample cases that must be considered are:

- Less than 3 points within a series of points to determine the convex hull. (This is explicity tested for).
- A series of points that have the same radial angle from the anchor. This can be handled by ordering them based on distance from the anchor in such case, or simply removing the closest point to the anchor from the queue of points to be processed.

Other than the first point, there is no requirement for explicit testing of the other conditions within the algorithm, as the algorithm naturally handles these conditions.

6.6 Dynamic Programming

Dynamic Programming is a problem solving approach and not a specific algorithm, and thus maybe used to solve a multitude of different problems.

The primary aspect of Dynamic Programming is the ability to break larger complex problems into simple problems, however to use Dynamic Programming, a problem should exhibit properties of overlapping subproblems and/or optimal substructure.

Overlapping subproblem refers to if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather

than always generating new subproblem. For example, the problem of computing the Fibonacci sequence exhibits overlapping subproblems. The problem of computing the nth Fibonacci number F(n), can be broken down into the subproblems of computing F(n − 1) and F(n − 2), and then adding the two. The subproblem of computing F(n − 1) can itself be broken down into a subproblem that involves computing F(n − 2). Therefore the computation of F(n − 2) is reused, and the Fibonacci sequence thus exhibits overlapping subproblems.

A problem is defined to have optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its subproblems. Optimal Substructure This property is used to determine the usefulness of dynamic programming and greedy algorithms in a problem. Typically, a greedy algorithm is used to solve a problem with optimal substructure if it can be proved by induction that this is optimal at each step. Otherwise, providing the problem exhibits overlapping subproblems as well, dynamic programming is used. If there are no appropriate greedy algorithms and the problem fails to exhibit overlapping subproblems, often a lengthy but straightforward search of the solution space is the best alternative.

6.6.1 Knapsack Problem

The Knapsack Problem is a problem of combinational optimisation, where given a set of items of differing weights, attempt to find the optimal list of items to be included based on value within set total weight constriants. The problem derives its name from the challenge faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

The problem often arises in areas where resource allocation is to occur where there are financial or physical restraints associated with activities and is useful for applications such as producing the least wasteful method of cutting raw materials, selection of investments, as well as generating keys for the Merkle–Hellman knapsack cryptosystem.

The knapsack problem formally is attempting to achieve:

$$\text{maximise } \sum_{i \in S} b_i \text{ subject to } \sum_{i \in S} w_i \leq W.$$

Where S is a set of items, each item i has a weight w_i and a benefit value b_i, whic is the utility value assigned to item i.

Algorithm 33 Knapsack Algorithm

Input A set S of n items, such that item i has positive benefit b_i, and positive integer weight w_i; positive integer maximum total weigth W.

Output For $w = 0, \ldots, W$, maximum benefit $B[w]$ of a subet of S with total weight as most w.

```
procedure 01KNAPSACK(S, W)
    for w ← 0 to W do
        B[w] ← 0
    end for
    for k ← 1 to n do
        for w ← W down to w_k do
            if B[w − w_k] + b_k > B[w] then
                B[w] ← B[w − w_k] + b_k
            end if
        end for
    end for
end procedure
```

6.6.1.1 Description of Working The Knapsack Problem can be solved by utilising an exhaustive search of the combination space, however requires that all subset combinations be calculated which often leads to **O(2^n)** type performance.

However the Knapsack Algorithm shown in Algorithm 33, simplifies the solution by breaking down the larger problem into smaller problems. This is formulated by defining the benefit for each item to be based on the weight of the item, and the number of items available, that is $B[k, w]$, and by examining the algorithm we see that $B[k, w]$ is built from $B[k-1, w]$ and possibly $B[k-1, w-w_k]$. Thus we have a subproblem definition that is simple (it involves just two parameters) and satisfies the subproblem optimisation condition. Moreover, it has subproblem overlap, for an optimal subset of total weight at most w may be used by many future problems.

The running time of the Knapsack algorithm is dominated by the two nested for-loops, where the one iterates n times and the inner one iterates at most W times. After it completes we can find the optimal value by locating the value $B[w]$ that is greatest among all $w \leq W$. Typically the Knapsack algorithm will be in the order of **O(nW)**, however it is common to refer to the algorithm as running in pseudo-polynominal time, for it's running time depends on the magnitude of a number of given input.

6.6.1.2 Implementation An implementation of the Knapsack Algorithm can be found in Listing 58.

```
public class Knapsack {

  /**
   * Default class to hold an item.
   */
  public class Item {
    public int value;
    public int weight;
  }

  /**
   * Determine the maximum weight to hold.
   * @param items Array of Item to be considered.
   * @param capacity The maximum capacity of the container.
   * @return The maximum weight of items.
   */
  public int Knapsack(Item[] items, int capacity) {
    int[][] solution = new int[capacity+1][items.length + 1];

    for (int k = 1; k < items.length + 1; k++) {
      for (int w = 1; w < capacity+1; w++) {
        if (items[k - 1].weight > w) {
          solution[w][k] = solution[w][k - 1];
        } else {
          solution[w][k] = Math.max(solution[w][k - 1],
                  solution[w - items[k - 1].weight][k - 1] + items[k - 1].value);
        }
      }
    }
    return solution[capacity][items.length];
  }
}
```

Listing 58: Knapsack Algorithm (Java)

6.6.1.3 Sample Problem - Knapsack Given weights and values for a set of items, determine the maximum value of items that can be carried in a vehicle with a given carrying capacity without exceeding that capacity.

INPUT

The input begins with a line containing a single integer, T, the number of test cases which follow, $1 \leq T \leq 100$.

For each test case, 4 lines of integers follow.

- The first line contains the capacity of the vehicle, $W, 100 \leq W \leq 5000$.
- The second line contains the number of items available to choose from, $N, 1 \leq N \leq 1000$.
- The third line contains weights for the individual items $w_1 \ldots w_n, 1 \leq w_i \leq 1000$.

- The fourth line contains values for the individual items, $v_1 \dots v_n, 1 \leq v_i \leq 1000$.

SAMPLE INPUT

```
2
5
4
2 1 3 2
12 10 20 15
10
4
7 3 4 5
42 12 40 25
```

OUTPUT

For each test case output the maximum value of the load that fits in the capacity of the vehicle on a line by itself.

SAMPLE OUTPUT

```
37
65
```

6.6.1.4 Sample Solution A sample solution to the Knapsack problem can be found in Listing 59.

```
import java.util.Scanner;

public class KnapsackProblem {

  /**
   * Main
   */
  public static void main(String[] args) {
    new KnapsackProblem().run();
  }

  public void run() {
    int vehicleCapacity;
    int numItems;

    Scanner in = new Scanner(System.in);

    int testCases = in.nextInt();
    while (testCases-- > 0) {

      // Get our vehicle capacity, and maximum number of items.
      vehicleCapacity = in.nextInt();
      numItems = in.nextInt();

      Item[] items = new Item[numItems];
      // Read in the weights and values for our items.
      for (int i = 0; i < numItems; i++) {
        items[i] = new Item();
        items[i].weight = in.nextInt();
      }
      for (int i = 0; i < numItems; i++) {
        items[i].value = in.nextInt();
      }

      // Calculate the max weight and output.
      System.out.println(Knapsack(items, vehicleCapacity));
    }
  }

  /**
   * Default class to hold an item.
   */
  public class Item {
    public int value;
```

```
    public int weight;
  }

  /**
   * Determine the maximum weight to hold.
   * @param items Array of Item to be considered.
   * @param capacity The maximum capacity of the container.
   * @return The maximum weight of items.
   */
  public int Knapsack(Item[] items, int capacity) {
    int[][] solution = new int[capacity+1][items.length + 1];

    for (int k = 1; k < items.length + 1; k++) {
      for (int w = 1; w < capacity+1; w++) {
        if (items[k - 1].weight > w) {
          solution[w][k] = solution[w][k - 1];
        } else {
          solution[w][k] = Math.max(solution[w][k - 1],
                  solution[w - items[k - 1].weight][k - 1] + items[k - 1].value);
        }
      }
    }
    return solution[capacity][items.length];
  }
}
```

Listing 59: Solution to Knapsack Problem (Java)

The solution to this problem is rather simple as it primarily relies on the implementation of the algorithm itself to produce a correct result.

Some sample cases that must be considered are:

- Only a single item exists in the list of items.
- Only a single item exists in the list of items, and it's weight exceeds the maximum weight.
- All items have weights that exceed the maximum weight.

However there is no requirement for explicit testing of these conditions within the algorithm, as the algorithm naturally handles these conditions.

6.6.2 Edit Distance

The Edit Distance Problem defines the edit distance for a pair of strings, as being the minimum number of edits needed to transform one string to the the other string. A number of allowed transactions are permitted for the transformation, these include:

- insertion of a character.
- deletion of a character.
- substitution of a single character.

For example, the edit distance between "hamlet" and "camelot" is 3.

hamlet $\rightarrow$ camlet $\rightarrow$ camelet $\rightarrow$ camelot.

Various applications for calculating the edit distance include:

- File Revision - The UNIX command `diff` finds the difference between two text files, producing an edit script to convert the first file into the second.

- Spelling Correction - Algorithms related to the edit distance may be used in spelling correctors. If a text contains a word, w, that is not in the dictionary, a 'close' word, i.e. one with a small edit distance to w, may be suggested as a correction. Transposition errors are common in written text. A transposition can be treated as a deletion plus an insertion, but a simple variation on the algorithm can treat a transposition as a single point mutation.
- Plagiarism Detection - The edit distance provides an indication of similarity that might be too close in some situations.
- Molecular Biology - The edit distance gives an indication of how 'close' two strings are. Similar measures are used to compute a distance between DNA sequences (strings over {A,C,G,T}, or protein sequences (over an alphabet of 20 amino acids), for various purposes, e.g.:
  1. to find genes or proteins that may have shared functions or properties
  2. to infer family relationships and evolutionary trees over different organisms
- Speech Recognition - Algorithms similar to those for the edit-distance problem are used in some speech recognition systems: find a close match between a new utterance and one in a library of classified utterances.

Additionally there are different algorithms for Edit Distance, each refined to serve a particular set of requirements or the domain in which they will exist, with most of these focused on the length of the alphabet to be processed and the length of strings to be processed. The algorithm described is also known as the Levenshtein distance.

Mathematically, edit distance between two strings a, b is given by $E_{a,b}(|a|, |b|)$:

$$E_{a,b}(i,j) = \begin{cases} 0 & , i = j = 0 \\ i & , j = 0 \wedge i > 0 \\ j & , i = 0 \wedge j > 0 \\ \min \begin{cases} E_{a,b}(i-1,j) + 1 \\ E_{a,b}(i,j-1) + 1 \\ E_{a,b}(i-1,j-1) + [a_i \neq b_j] \end{cases} & , \text{else} \end{cases}$$

Algorithm 34 Edit Distance Algorithm

Input Two strings a and b of non-zero length.
Output The edit distance from a to b.

```
procedure EDITDISTANCE(a, b)
    len_a ← a.length
    len_b ← b.length
    cost ← 0
    if a[0] ≠ b[0] then
        cost ← 1
    end if
    if len_a = 0 then
        return len_b
    else if len_b = 0 then
        return len_a
    else
        return min(EditDistance(a[1 ... len_a], b) + 1,
                   EditDistance(a, b[1 ... len_b]) + 1,
                   EditDistance(a[1 ... len_a], b[1 ... len_b]) + cost)
    end if
end procedure
```

6.6.2.1 Description of Working The Edit Distance Algorithm as shown in Algorithm 34, creates 4 cases that define the problem. The first cases defines, then when both strings are 0 length, the distance is 0. The second and third cases are simply that is the length of one of the string arguments is 0, then the edit distance is simply the length of the alternate argument string. (These are the base cases for the algorithm). These 3 cases are clearly displayed in the mathematical description given above.

The final and fourth case breaks the main problem into a set of subproblems, in that this may be E($a, b-1$), E($a-1, b$) and E($a-1, b-1$) + $[a_i \neq b_j]$ respectively. As it's not possible to determine the correct course of action, we attempt all 3 subproblems and take the minimum of the three.

Since examination of the relations reveals that E(a, b) depends only on E(a', b') where a' is shorter than a, or b' is shorter than b, or both. This allows the dynamic programming technique to be used, in that a two dimension matrix m can be used to hold the distance values and the matrix can be computed row by row. That is row m[i,] depends only on row $m[i-1,]$.

This has multiple advantages, in that we avoid recursion of the main algorithm, and previously calculated values are not recaculated.

The gives the a performance complexity of **O(|a|.|b|)** or **O(n^2)** if a and b have similar lengths.

6.6.2.2 Implementation An implementation of the Edit Distance Algorithm can be found in Listing 60.

```
public class EditDistance {
  /**
   * Return the the cost of the difference between the two characters.
   *
   * @param a The first character
   * @param b The second character
   */
  public int diff(char a, char b) {
    return (a == b ? 0 : 1);
  }

  /**
   * Calculate the edit distance between the two strings.
   *
   * @param source The source string
   * @param target The resulting string.
   * @return The distance between both strings
   */
  public int editDistance(String source, String target) {
    // create the matrix
    int editMatrix[][] = new int[source.length() + 1][target.length() + 1];

    // initialise first column
    for (int i = 0; i <= source.length(); i++) {
      editMatrix[i][0] = i;
    }

    // initialise first row
    for (int i = 0; i <= target.length(); i++) {
      editMatrix[0][i] = i;
    }

    // Complete the matrix
    for (int i = 1; i <= source.length(); i++) {
      for (int j = 1; j <= target.length(); j++) {
        editMatrix[i][j] = Math.min(Math.min(editMatrix[i - 1][j] + 1,
            editMatrix[i][j - 1] + 1),
            editMatrix[i - 1][j - 1] + diff(source.charAt(i - 1), target.charAt(j - 1)));
      }
    }

    return editMatrix[source.length()][target.length()];
  }
}
```

Listing 60: Edit Distance Algorithm (Java)

6.6.2.3 Sample Problem - Edit Distance Calculate the Levenshtein Distance between two stings x and y. The cost of deletion is 1, the cost of insertion is 1, and the cost of substitution is 1.

INPUT

The input begins with a line containing a single integer, T, the number of test cases which follow. For each test case, two lines of text, each with length between 0 and 1000 folllow.

SAMPLE INPUT

```
2
polynomial
exponential
virtual
ritual
four


four
```

OUTPUT

For each test case output the Levenshtein Distance between the two strings on a line by itself.

SAMPLE OUTPUT

```
6
2
4
4
```

6.6.2.4 Sample Solution A sample solution to the Edit Distance problem can be found in Listing 61.

```
import java.util.Scanner;

public class EditDistanceProblem {

  /**
   * Main
   */
  public static void main(String[] args) {
    new EditDistanceProblem().run();
  }

  public void run() {
    Scanner sc = new Scanner(System.in);
    // number of test cases
    int cases = sc.nextInt();
    sc.nextLine();

    while (cases-- > 0) {
      // calculate the distance and output
      System.out.println(editDistance(sc.nextLine(), sc.nextLine()));
    }
  }

  /**
   * Return the the cost of the difference between the two characters.
   *
   * @param a The first character
   * @param b The second character
   */
  public int diff(char a, char b) {
    return (a == b ? 0 : 1);
  }

  /**
   * Calculate the edit distance between the two strings.
   *
   * @param source The source string
```

```
    * @param target The resulting string.
    * @return The distance between both strings
    */
   public int editDistance(String source, String target) {
     // create the matrix
     int editMatrix[][] = new int[source.length() + 1][target.length() + 1];

     // initialise first column
     for (int i = 0; i <= source.length(); i++) {
       editMatrix[i][0] = i;
     }

     // initialise first row
     for (int i = 0; i <= target.length(); i++) {
       editMatrix[0][i] = i;
     }

     // Complete the matrix
     for (int i = 1; i <= source.length(); i++) {
       for (int j = 1; j <= target.length(); j++) {
         editMatrix[i][j] = Math.min(Math.min(editMatrix[i - 1][j] + 1,
              editMatrix[i][j - 1] + 1),
              editMatrix[i - 1][j - 1] + diff(source.charAt(i - 1), target.charAt(j - 1)));
       }
     }

     return editMatrix[source.length()][target.length()];
   }
}
```

Listing 61: Solution to Edit Distance Problem (Java)

The solution to this problem is rather simple as it primarily relies on the implementation of the algorithm itself to produce a correct result.

Some sample cases that must be considered are:

- Zero Length Strings.

However there is no requirement for explicit testing of these conditions within the algorithm, as the algorithm naturally handles these conditions.

7 Afterword

A References

Throughout this guide multipe sources where used to construct this guide. The main sources of information presented through out this guide were:

- Corney, M. (2011). INB204 Special Topic Advanced Algorithms, Semester 2, 2011, Lecture Slides and Notes.
- Tang, M. (2012). INB371 Data Structures and Algorithms, Semester 1, 2012, Lecture Slides and Notes.
- Goodrich, M., Tamassia, R., (2002) *Algorithm Design: Foundations, Analysis and Internet Examples*, Wiley Press. ISBN: 0-471-38365-1 (QUT Library: 005.1 622)

Additionally, the following sections were derived from the following sources:

- Section 2.5 - Competition Strategy, derived from:
  - Corney, M. (2011). ACM Style Programming Competitions.pdf (part of INB204 Special Topic Advanced Algorithms, Semester 2, 2011, Notes).
- Section 4.2 - Program Structure, derived from:
  - Corney, M. (2010). Program Structure - Java.pdf (part of INB204 Special Topic Advanced Algorithms, Semester 2, 2011, Notes).
- Section 5.3.1 - Array Performance, Timing Values and Cache layouts, derived from:
  - Intel Corporation, (2002) *Intel Pentium 4 and Intel Xeon Processor Optimisation Reference Manual*. Intel Corporation, Order Number: 248966-007 (No longer available in print and very limited circulation, but newer editions in electronic form may be found at: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html).
- Section 6.1.1 - Greatest Common Divisor, Description Text, derived from:
  - Wikipedia (n.d.), Greatest Common Divisor, Retrieved August 16, 2012, from http://en.wikipedia.org/wiki/Greatest_common_divisor
  - Wolfram Mathworld (n.d.), Greatest Common Divisor, Retrieved August 13, 2012, from http://mathworld.wolfram.com/GreatestCommonDivisor.html
- Section 6.1.2 - Sieve of Eratosthenes, Figure 6.1, derived from:
  - Wikipedia (n.d.), Sieve of Eratosthenes, Retrieved August 15, 2012, from http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
- Section 6.3.6.1 - A* Search, Figures, derived from:
  - Lester, P. (2005), A* Pathfinding for Beginners, Retrived September 10, 2012, from http://www.policyalmanac.org/games/aStarTutorial.htm
- Section 6.4.2 - Minimum Cost, Maximum Flow, introduction text, algorithm and description of working derived from:
  - Corney, M. (2011), Maximum Flow and Minimum Cost Maximum Flow document. Emailed from M. Corney on October 9, 2012 and part of INB204 Special Topic Advanced Algorithms, Semester 2, 2011, Notes.
- Section 6.5.1 - Bentley-Ottmann Algorithm, algorithm derived from:
  - Sunday, D (2006), Intersection for a 2D set of Segments, Retrieved September 20, 2012, from http://softsurfer.com/Archive/algorithm_0108/algorithm_0108.htm

- Section 6.6.1 - Knapsack Description, excerpts taken from:
  - Wikipedia (n.d.), Knapsack problem, Retrieved September 17, 2012, from http://en.wikipedia.org/wiki/Knapsack_problem