On the Science Behind Computing

Samir Khuller *

1 Preface

My research lies in the area of Computer Science called Algorithms. The reaction I get from most people is “Oh, I know about that - I learnt about Logarithms in High School”. First of all Algorithms is not Logarithms spelt incorrectly. I am writing this monograph to correct the impression that many of my relatives and friends have about what is it that I do. Most people, upon hearing that I am a “Computer Science Professor” immediately conclude that I am immensely qualified to fix their PC. Nothing could be further from the truth! In fact, I had never even used a PC in my life until 1997, when I was asked to fix one for a neighbor. I was too embarassed to tell her that I will not be able to fix the PC. In the end I was able to fix it, but only by a stroke of luck. I suggested that the cable was faulty, lacking any other useful suggestion. She changed the cable the next day, and called me to tell me that the printer now worked. She now thinks I am a whiz at fixing PC’s. In fact I was so scared of the next call I was going to get from her, when there was a serious problem with her PC, that we moved out of the neighborhood.

Another incident that prompted the writing of this monograph was a discussion with someone who was taking a course in computer applications. I explained my work to her by using a simple example of “sorting”, since she seemed to be familiar with that problem. Given a collection of numbers such as \{63,24,1,49,3\} how do we put them in sorted order? The sorted order (in increasing value) is 1,3,24,49,63. I started to explain that there were many methods that could be used to obtain this order. Different methods (algorithms) can take a different number of steps based on what method they use. Some algorithms use more steps, and some use fewer steps. After this short discussion on sorting algorithms she said “But all sorting algorithms take the same amount of time. You just hit return and the answer comes back in sorted order.” Sigh.

Finally, I decided that I must write a monograph that I can hand to my relatives or friends, or their high school children, so that they can get an understanding of my field of research if they do so care. People are knowledgeable about elementary Chemistry, Physics, Mathematics, Biology, and other subjects that they learn about in middle and high school. Sadly, when people talk about teaching computer science at the high school level, its mostly about programming in one language, or programming in another language, or both. I learnt about the field of Algorithms somewhat late in life unfortunately. To me, it seems like the most fascinating subject within the entire field of Computer Science. It is simply to convey some of the beauty and elegance of this field did I decide to write this monograph. I see no reason as to why students interested in Mathematics, or abstract thought cannot learn about this field at earlier stages of their schooling. In addition, some of these algorithms play a role in the design of widely used software. We all

* Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail : samir@cs.umd.edu.
use these software packages to do things for us, without realizing how these algorithms are used, or are even being aware of their existence.

I will introduce the reader to many problems, and discuss the solutions to several of these problems explaining how the algorithms work mostly by example. For some algorithms, we will also be able to discuss the correctness proofs.

An algorithmic approach represents a way of thinking about problems in a fairly abstract manner. Such an approach is useful, perhaps even before learning programming. Learning programming teaches one how to use a set of tools that can eventually be used to perform interesting and useful computations. But what should these computations be? Which problems are we trying to solve, and how?

2 Introduction

The basic idea behind this writeup is to present the science behind computing. First of all, this is already a daunting challenge – to try and condense decades of research on computing into one small semester long course. There are too many aspects of computing ranging from the design of the actual hardware (building computers), to design of systems software (software that manages your computer), to designing applications, not to mention the Internet and a host of other issues such as defining computation, understanding the limits of computation etc. The attempt here will be to introduce specific nuggets of information that will convey a sense for what the field of computing is about and this will hopefully provide a glimpse into the mind of a computer scientist and our way of looking at the world.

Many people use several software packages, without really having an understanding as to how these work, or without a complete understanding of what exactly happened, let alone how it happened. There is no sound of the engine revving up, or the squeal of the brakes, or the smell of burning rubber as in a powerful racecar. Thus we cannot “feel” computation. Since the underlying software is not something people can “see”, there is little appreciation for what is involved in building software tools that work efficiently and correctly, unlike appreciating an impressive building’s architecture, or looking under the hood of a sports car! For example people use online maps, get directions, use word processing software, search for files on their computer containing keywords, use search engines etc. They simply click a button and the answer comes back promptly without giving a clue as to what really happened in that fraction of a second. Computing has become like “electricity”, you flick a switch and something happens. Except that now you don’t just get one thing, but access to a world of information.

Moreover, there is even less understanding about the underlying algorithms that drive software packages. All software packages that come on a CD drive, look identical. Why is it that it takes years of effort, with hundreds of programmers all working day and night to produce software that goes into a single CD? What are all the thousands of engineers at Google, Microsoft and IBM doing? They are busy solving problems, designing systems and efficient methods that takes years of effort to engineer and build.

One of the goals of this course is to essentially show how a lot of the tools we use, those that have become part of our everyday life, really work. More surprisingly, how they can be mapped to rather easy to state mathematical objects and problems. These mathematical problems appear to be toy like sometimes, but the algorithms that have been developed to solve them, are incredibly powerful, and can be used to build rather complex and useful tools.
3 Introduction to Algorithms

Algorithms can be thought of as a set of rules that specify how to compute the desired output, from an input to a problem. A simple algorithm might describe how to bake a cake using a set of ingredients. The set of rules that specify how to convert a specified input into the desired output is the algorithm itself. Algorithms can be described at a variety of levels. Our descriptions, will mostly be given in English. A computer on the other hand needs very careful and precise specifications about the algorithm. Even a small mistake, or imprecise specification can create unexpected and strange behavior from a computer.

It is important here to note the distinction between an algorithm and a program. An algorithm is a set of rules that tells us how to perform the computation, as well as what computation to perform. A program on the other hand, is the embodiment of this set of rules in a specific programming language that can be executed on a computer. It is similar to a story, that can be written and translated in several languages. The story corresponds to the algorithm, and the description of the story in a specific language corresponds to a program that implements the algorithm. At the lowest level\(^1\), a programmer could describe her algorithm in assembly language, a very simple language with very simple instructions. Programs written in this language are somewhat long, and tedious and hard to understand. Thus most people use high level languages that make it easy to specify the algorithm. Compilers are then used to translate the descriptions given in high level languages to assembly language.

The field of algorithms is a little like mathematics. Often the problems are easy to describe, the solutions are sometimes very ingenious, and sometimes even easy to verify by a reader who knows little about the problem! Often it is easy to understand new problems, as well as understand their solutions. However, coming up with the solutions can be quite challenging; and being an expert sometimes is of no help at all. The most creative solutions could come from smart graduate students, who have just started research in the area (often they are the ones with wild ideas, and have the energy to show that some of these wild ideas actually work). In the process of explaining the problems and their solutions, we will see that the reader can learn some useful tools which can then be used to attack other problems.

I do not want to assume a high level of mathematical maturity, so the “proofs” will be informal, usually by simple explanation or examples. The reader is urged to read the technical papers or books referenced at the end of each section to identify further readings.

4 Algorithms Background

Algorithms are simply methods or recipes for solving problems. An algorithm is a step by step procedure that takes an input and produces the desired output. Take for example the simple problem of “given an integer \(n\), is it prime?”. Recall that a prime number is one that is only divisible by 1 or itself, and no other number. For example, 2 and 3 are prime numbers, but 4 is not prime as 2 divides 4. Again, 5 and 7 are prime, but 6, 8 and 9 are not since 2 divides 6 and 8, and 3 divides 9. For small numbers, it is easy to obtain a listing of all prime numbers by checking for divisors that are smaller than the number itself. However, given a number such as 1019, how would we check to see if it is prime? (Prime numbers have a variety of uses...) One trivial algorithm would be to check if some number \(i\) divides it, but we will have to perform this test for each value of \(i < 1019\). Clearly 2 does not divide it, and neither does 3. If 2 does not

\(^1\)Strictly speaking this is not the lowest level, since a programmer could also write her program in machine code, a language directly understood by the computer.
divide it, then clearly 4 does not either. In fact, we can generalize this observation to only check for divisibility with known primes. The numbers 5 and 7 do not divide it either. What about 11? Is there no better way? What would we do if we needed to check if a much larger number was prime? While there are fast algorithms for solving this problem, they are out of scope for this monograph.

However, we describe one simple approach. Suppose we wish to check if 887 is prime or not. We can try to divide 889 by 2, 3, 4, 5 etc and after failing to find a number that divides 889, we ask ourselves if we have to keep going all the way to 889 itself? A moment of thought suggests that we can actually stop when we reach 30! Suppose there was a number that divided 889. Note that then there is always a number \( j \leq 30 \) that would divide 889.

Suppose we have a street map of Washington DC. We need to get to the airport quickly, in order to not miss our flight. How do we choose a route? People have different ways for doing this. Some rely on local knowledge of the area, and knowledge of traffic patterns etc. Clearly if we have been living in an area for a long time, we can do a good job doing route planning. Suppose we are just visiting a city and renting a car and we have no idea about the traffic patterns? Suppose we just want to simply take the shortest route? Or a route that will get us there in the least amount of time. Often we look at a map and do “visual navigation” to compute a route. One algorithm would be “go from your starting point via some short route to the nearest highway, and then use the highway to approximately where we wish to go, and then get off at the appropriately marked exit, and then look up the map for a short route to the destination”. Have you ever thought about how a computer would solve this problem? There are services such as mapquest\(^2\) that gives directions and routes between chosen locations. Of course there is an issue of the “representation” of the map in the computer. One can scan a map in, and this is fine for simply doing a visual display of the map, or for zooming and viewing parts of the map. How about actually doing route planning? For this a different representation is required so that we can manipulate the information in an easier manner, perform computations and solve interesting problems.

How do we know if the algorithm is good or bad? Typically most algorithms are evaluated by the number of “elementary steps” that they take to solve a given problem (also referred to as running time of the algorithm). However, what an elementary step is can be the subject of a lot of discussion. Informally, we will assume that this is an “instruction” that can be carried out in some fixed amount of time. Often, the actual number of steps taken by the algorithm depend on several factors. The size of the input to the algorithm is itself one of the biggest factors determining the running time of the algorithm. Moreover, the running time of the algorithm may vary when given two different inputs of the same size. We usually parameterize the size of the input to the algorithm by a parameter \( n \), and measure the maximum running time of the algorithm for any input of size \( n \). The issue is that for different inputs of the same size, the algorithm may have different running times. How do we measure the running time in this case as a function of \( n \)? In fact, there may be an infinite number of possible inputs of size \( n \). One option is to try and derive some bound on the average running time for all possible inputs of size \( n \). However, this is non-trivial and has only been done for a relatively small number of problems. Another traditional method is to derive an upper bound on the running time over all possible inputs of size \( n \). This is referred to as worst case analysis of the algorithm, and is probably the most common method used for analyzing algorithms. However sometimes this can

\(^2\)http://www.mapquest.com
be too pessimistic. There are some classical algorithms, such as the famous simplex algorithm for solving linear programs, which have a very poor worst case bound, but on average they do extremely well and are the algorithms that should be used in practice.

Let us focus briefly on the need to design efficient algorithms. Consider for example the problem faced by a police sheriff in a small town. The sheriff would like to visit every street intersection to check on the traffic lights after a big storm. How does the sheriff plan a route that minimizes the travel time? One approach is to enumerate all possible ways of visiting the intersections, computing the cost of each one. However, this can be a very large number. In other words, even for reasonably small cities, the total number of possible routes is huge and cannot be enumerated efficiently by even a computer. For example, if we are looking to visit a set of $n$ locations, and consider all possible orders of visiting the locations, the number of orders to be examined is $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$. Even for $n = 10$ this is as large as 3628800. This problem is also popularly referred to as the Traveling Salesman Problem and several heuristics are known for it. We will discuss this problem in more detail later.

5 Choice of Problems

There are many sub-areas in the field of algorithms - this can be seen by the proliferation of conferences and workshops in this field such as online algorithms, approximation algorithms, dynamic algorithms, efficient algorithms, geometric algorithms, graph algorithms, string matching algorithms, algebraic algorithms, biological algorithms etc.

To illustrate the beauty of algorithms I have chosen topics that are more readily understandable to a non-specialist, rather than choosing problems that are more important. Sometimes, the background needed to explain a particular result requires more mathematical knowledge than I wish to assume, or simply requires knowledge about other results. I have tried to pick examples that are related to real-life problems, or fundamental problems that are useful in developing methods that are really useful in practice, and at the same time easy to explain. While these are not precisely the problems I work on myself, but they do represent a good sample of the thought processes involved in the design of algorithms.

Algorithms and Computing were initially confined to performing mathematical calculations. A quote widely attributed to Thomas Watson Jr. (president of IBM), is “I think there is a world market for maybe five computers”. According to Wikipedia, there is no proof that he ever said this. However, it is based on the fact that computers were rather expensive devices, that were in use for performing primarily mathematical calculations. In time, however, it became clear that computers could process information of any type and not just perform mathematical calculations.

6 Some Examples from Everyday Life

In this section we study some compelling examples from everyday life. We will discuss some compelling examples from everyday life, and show how mathematical modeling and algorithms can be used to approach these problems. However, it should be noted that many of the specific algorithms used are closely guarded company secrets and we can only observe properties about these methods based on their behaviour without knowing the specific algorithms being used. For this course its not even important to understand the exact methods in use. What we are trying to show is simply the connection between research in algorithms to specific applications. A lot more information about these topics is likely to be available online.
6.1 Online News Let us consider the popular news aggregator Google News. What is it about Google News that makes it different from other online news agencies such as the Washington Post, or New York Times? The main innovative feature of Google News is that it provides a grouping of articles by topic. Consider the reporting of the Iranian elections. All articles from different sources are grouped into one section, and you can read articles from the New York Times, BBC News, or the Iranian government news about the elections and see the different perspectives and reporting.

This begs the question – how did these articles get grouped together? If you look at the bottom of the page it says “The selection and placement of stories on this page were determined automatically by a computer program.” There are literally hundreds of news channels (if not thousands), all reporting hundreds of articles everyday. Moreover, these news articles are scanned soon after they appear, analysed and indexed as well as categorized immediately. This is too large a problem to be done by hand. Can we have a set of humans who can read and discuss each article and decide which articles are related to each other, and what categories should they be listed under? I suppose one could employ an army of humans to do this, but this task can be done much faster and automatically using software.

However this begs the question – how can this be done?

6.2 Internet Search In 1995 when the world wide web was fairly new, I tried to search for information on the Grand Canyon. Many of the web sites that came up were of the type “Here are the pictures of my trip to the Grand Canyon”. However, these were not the web pages I was looking for! I was looking for the web page of the National Park Service. This page was also there but was buried a bit lower down. However, try doing this search today on a modern search engine. The National Park Service web page now comes up as the very first hit on Google. Clearly something changed! A new method is being used to rank relevant pages when you search for information, this new method has its origins in two papers that were based on analysing the link structure of the web (the links are the hypertext references that web pages have to one another).

6.3 eHarmony Eharmony is a company that matches people using a sophisticated algorithm! Typically, most individuals looking for a mate fill out a detailed questionnaire about themselves, and abstractly we can view the process as the following graph problem. Let $G = (M, F, E)$ be a bipartite graph defined by the set of males $M$ and the set of females $F$. The profile information is presumably used to obtain a compatibility score $w(m_i, f_j)$ between male $m_i$ and female $f_j$. Suppose we assign scores between 0 and 1, with 1 being the highest possible compatibility and 0 being the lowest possible compatibility score. One option may be to suggest matches between all pairs of individuals whose score is above a certain threshold $\delta$. However, this has the undesirable feature that some nodes will have lot of matches suggested, while other nodes have very few suggested matches. How do we fix this problem? Clearly, we would like the suggested matches to form a subgraph where each node has degree at most $\Delta$. Namely that a small number of matches are suggested to each individual every week. Moreover, the graph is continuously evolving as more and more people join everyday. This leads to a massive computational problem that involves finding say a maximum weight subgraph where every node has degree at most $\Delta$, and we also do not have many low weight edges.
6.4 **Kidney Exchange** This is another amazing application of matchings in graphs. Suppose Tom needs a kidney, and his wife Jane is a willing donor. However, due to compatibility issues, her kidney cannot be donated to Tom.

Suppose that in the same city, there is another couple - Mary and Dan. Mary has a failing kidney and Dan is a willing donor. However, again due to a mismatch in kidney types, Dan’s kidney cannot be donated to Mary. However, it may happen that Dan’s kidney can be donated to Tom and Jane’s kidney to Mary. Each donors desires that their partner simply get a kidney. Its not important who specifically gets their kidney. As a result we can set up a kidney exchange graph as follows. Each donor-recipient pair is a node in this graph (assuming their kidney does not match). Edges connect pairs of nodes that have cross compatibility. In this graph, we are looking for a pairing of nodes (also called a matching).

6.5 **Re-assigning Starbucks employees**

6.6 **Online Advertising**

6.7 **Image Search**

6.8 **Recommendation Systems**
7 Computing Square-roots

We start off with a very ancient problem. Suppose Ramses needs to give a poor farmer a square piece of land of a specified area, then how does he go about accomplishing this? Let’s say that he need to give $N$ units of a square plot of land to the farmer. What should each side of the square be? If the side of the square is $X$ then we know that $X \cdot X = N$. In other words $X = \sqrt{N}$. How do we compute squareroot? These days we can hit one button on a calculator and get the answer pretty accurately – but how would we have done this five hundred years back? Or moreover, how does the little calculator compute square-roots so quickly?

For this we need an algorithm to help us find the answer. Here is one possible algorithm. We start off with a guess for $X$. It is often easy to find a good starting guess (more on this later). For example if we wish to compute the square root of 40 then 6 or 7 are both reasonable guesses to start with since $6^2 = 36$ and $7^2 = 49$ since we know that the answer will be between 6 and 7. However, the algorithm works even if we do not pick a good guess!

The algorithm is as follows. Assume that $\epsilon$ is the error we are willing to tolerate.

If $|X - \frac{N}{X}| \leq \epsilon$ then stop; otherwise set $X \leftarrow \frac{1}{2}(X + \frac{N}{X})$ and repeat.

Let us apply this algorithm to a simple example with $N = 40$ (we only report it to 3 decimal places). As you can see this converges extremely quickly.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>$X$</th>
<th>$\frac{N}{X}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>6.666</td>
</tr>
<tr>
<td>2</td>
<td>6.333</td>
<td>6.315</td>
</tr>
<tr>
<td>3</td>
<td>6.324</td>
<td>6.324</td>
</tr>
</tbody>
</table>

Here is another example.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>$X$</th>
<th>$\frac{N}{X}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>5.714</td>
</tr>
<tr>
<td>3</td>
<td>6.357</td>
<td>6.292</td>
</tr>
<tr>
<td>4</td>
<td>6.324</td>
<td>6.324</td>
</tr>
</tbody>
</table>

We come back to the problem of getting an accurate starting estimate for $X$. If $N$ is a perfect square, we are done.

Suppose we have $Y^2 < N < (Y + 1)^2$. (Example: If $N = 40$ then $Y = 6$, since 40 is between $6^2$ and $7^2$. Imagine a square of side $Y$. This clearly has area $Y^2$. We would like to increase the area of the square to $N$. Suppose we increase the length of the side by adding $\delta$ to each side. The new area now is $(Y + \delta)^2 = Y^2 + 2Y\delta + \delta^2$. We would like this area to be $N$. Since $\delta < 1$ we can ignore $\delta^2$ since that is very small. So we would like

$$Y^2 + 2Y\delta \approx N$$

$$\delta = \frac{N - Y^2}{2Y}$$

If we set $N = 40$ and $Y = 6$ this gives $\delta = \frac{40 - 36}{2 \cdot 6} = \frac{4}{12} = \frac{1}{3}$. Thus $\sqrt{N} \approx 6\frac{1}{3}$. This is a pretty good estimate!
8 Stable Marriage Problem

This problem finds applications in a variety of situations where we are trying to find an “optimal” pairing of people. One situation arises in the assignment of medical interns to hospitals, where an algorithm based on this approach is used, once the interns and the hospitals submit their “preference” lists. The situation is not identical to the one described below; however, the basic approach that is used is very similar to the one described below.

In the most basic version of the problem we are given a set of $n$ boys and $n$ girls. Each girl provides a preference list of all the boys such that the first element in the list is the boy that she likes the most and the last element in the list is the least preferred boy. She is required to rank order all the boys. In a similar manner, each boy rank orders all the girls. The preference lists of the girls and the boys contain all the individuals of the opposite sex. A pairing of each boy with a distinct girl is called a marriage, and the matched pairs are called couples. So for example, let us assume that there are three boys and three girls. Suppose that the boys are called John, Rajeev and Pierre and the girls are called Janet, Saira and Marie. Each boy submits a rank ordered preference list of the girls - so for example, the list submitted by the boys could be:

John: Saira, Marie, Janet.
Rajeev: Marie, Saira, Janet.
Pierre: Marie, Janet, Saira.

The lists submitted by the girls could be:

Marie: John, Pierre, Rajeev.

A marriage is a pairing of each boy with a distinct girl. A marriage is called unstable if the following is true: There is a pair of individuals, who are not married to each other, but both prefer each other to their current mates. So for example, the marriage \((John, Saira), (Rajeev, Janet), (Pierre, Marie)\) is unstable since Rajeev clearly prefers Saira to his current partner Janet. At the same time, Saira prefers Rajeev to her current mate John. This is an unstable marriage since both have an incentive to leave their partners for each other. The marriage is called unstable, regardless of how the jilted partners John and Janet like each other.

The most interesting fact is that a stable marriage always exists! Moreover, there is a simple algorithm that can find it. (If only Computer Scientists were allowed to run society, and people’s preference lists did not change!)

The algorithm proceeds in parallel steps, with the boys making proposals. Even though we describe the method as a synchronized process, actually it does not really matter for the algorithm in what order the boys make proposals. Initially everyone is unengaged.

**Algorithm for finding a stable marriage**

**Algorithm 8.1.**

Each currently unengaged boy proposes to the most preferred girl that he has not proposed to until now.

Every girl who receives a proposal checks her preference list. If she is unengaged, she accepts the best proposal, turning down the other proposals that she receives. If she is currently engaged,
she breaks the engagement if she receives a better proposal and accepts the new proposal, and
gets engaged to the most preferred boy among the ones who proposed to her.

In the next round, the boys who lost their fiancés as well as the ones who are unengaged, will
propose. If ever we reach a situation where everyone is engaged then the algorithm halts and
outputs this solution as a stable marriage.

We will illustrate this algorithm by an running it on the above example to make it clear. In
the first round, all three boys propose to their top choice. Thus John proposes to Saira, Rajeev to
Marie and Pierre to Marie. Note that Saira receives one proposal and accepts it, getting engaged
to John. At the same time Marie receives two proposals, and prefers Pierre to Rajeev. She gets
engaged to Pierre and turns down Rajeev’s proposal. In the next round, Rajeev proposes to the
next girl on his list, namely Saira. Saira breaks her engagement with John, since she receives
a preferred proposal and gets engaged to Rajeev. In the next round, John proposes to Marie,
the next girl on his list. Marie prefers the new proposal from John, and breaks her engagement
with Pierre and gets engaged to John. In the next round, Pierre proposes to Janet. This is
the first proposal that Janet receives and she accepts it. All parties are now engaged and we
perform the wedding ceremonies. We end up with the marriage: (Pierre, Janet), (Rajeev, Saira)
and (John, Marie).

The above algorithm will always terminate: because, when a boy proposes to the last girl in
his list all the other girls are engaged (to the other \( n - 1 \) boys), so the remaining girl (who is not
engaged) has to accept his proposal.

**Proposition 8.1.** The marriage found by the above algorithm is **stable**.

**Proof.** We can informally argue this as follows. Suppose the algorithm ends up with a marriage.
Assume that there is a boy \( A \) who prefers \( Y \) to his partner \( B \). At the same time \( Y \) prefers \( A \)
to her partner \( X \). Since \( A \) prefers \( Y \), \( A \) must have proposed to \( Y \) before proposing to \( B \) and
was turned down (either immediately, or was engaged to her for a while before \( Y \) broke the
engagement). In either case, \( Y \) gets a partner who is preferable to \( A \). Hence the partner that \( Y \)
ends up with, must be a preferred choice over \( A \). This is a contradiction to the assumption that
\( Y \) prefers \( A \) to her partner \( X \).

To see this point, observe that in the stable marriage produced by the algorithm there is no
unstable pair. For example, note that Rajeev prefers Marie to Saira, his wife. However, he did
propose to Marie, who turned him down for Pierre, and ended up with an even better partner
John (from her point of view). In fact, when a boy ends up with a girl who is not his first choice
(for example John ends up with Marie), there is no stable marriage at all with someone he prefers
to the partner he gets. In particular, there is no stable marriage at all in which John is paired
with his first choice Saira.

Interestingly, the algorithm can also be run with the girls proposing to the boys. This may
produce a different stable marriage. In fact, the solution is far from unique. There are situations
when there is a large number of possible stable marriages. In fact, the sex that proposes has a
distinct advantage in this approach. With the way in which we have described the algorithm,
the boys end up with the best choices they possibly can in any stable solution. The girls end up
with the worst possible solution from their point of view! In fact in the example given above,
note that if the girls are the ones proposing, after the first round itself the algorithm terminates
and each girl gets her first choice.
<table>
<thead>
<tr>
<th></th>
<th>Round 1</th>
<th>Round 2</th>
<th>Round 3</th>
<th>Round 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Saira ✓</td>
<td>Saira</td>
<td>Marie ✓ → Marie</td>
<td></td>
</tr>
<tr>
<td>Rajeev</td>
<td>Marie ¥</td>
<td>Saira ✓ → Saira</td>
<td>→ Saira</td>
<td></td>
</tr>
<tr>
<td>Pierre</td>
<td>Marie ✓ → Marie</td>
<td>→ Marie</td>
<td></td>
<td>Janet ¥</td>
</tr>
</tbody>
</table>

**KEY**

- ¥ = rejected immediately
- ✓ = accepted
- ¥¥ = rejected (after acceptance)
- → = relationship continued to next round

Figure 1: Running the stable marriage algorithm.
9 Bipartite Matchings

Let us consider another type of assignment problem, in a sense similar to the stable marriage problem. Suppose we have a set of workers $W$ and a set of jobs $J$. For example $W = \{Bill, Jill, Tom\}$ and $J = \{Mulching, Planting, Trimming\}$. Moreover every worker is not qualified to perform each task. For example Bill can only do mulching. Jill can do all three jobs, and Tom can either do planting or trimming. How can we assign each worker to one job that they are capable of performing? A matching is simply an assignment of workers to jobs, so that each worker is assigned to at most one job, and each job is assigned to at most one worker. A perfect matching is a matching in which every worker is assigned to a unique (and distinct) job, and each job has a worker assigned to it. For convenience we will assume that the number of jobs and workers is the same. There are situations when there is no perfect matching. For example if both Bill and Jill can only do mulching, then there is no perfect matching. We can model this situation as a graph. There is a set of nodes $W$, one node for each worker, and a set of nodes $J$, one node for each job. We add a link between a worker and a job if the worker can perform that job. Given an arbitrary graph structure of this type we would like to quickly compute a perfect matching (note that the perfect matching is not unique, and several different solutions may exist). For example, we can have a solution $\{(Bill, Mulching), (Jill, Planting), (Tom, Trimming)\}$ or $\{(Bill, Mulching), (Jill, Trimming), (Tom, Planting)\}$. While it is easy to find all the solutions for such a small graph, how do we solve the problem quickly when we have hundreds of workers and jobs? Or even thousands of workers and jobs?

Of course one can try all possible ways of assigning workers to jobs, and then check which ones are feasible assignments. Such an algorithm while easy to implement on a computer, can actually be extremely slow. The running time of such an algorithm is $N!$, where $N$ is the number of jobs (or workers). There are many situations where we need to find a maximum (or perfect) matching in a graph, so it will be incredibly useful to develop such an algorithm. Below we describe precisely such an algorithm.

We will illustrate this algorithm by showing how to apply it to the “prom problem”.

Let suppose there are four girls and four boys. Each person specifies a subset of people they are willing to go to the prom with. For example Margaret, Jane, Amy and Katherine would all like to go to the prom with someone they like. Suppose Margaret is willing to go with Tom or Bill. Jane will only consider going with Bill. Amy on the other hand is willing to go with either Bill, Tom, or Don. Finally, Katherine is willing to go either with Don, or Chris. Each girl lets their preferences be known to their mothers. The mothers all meet at Starbucks one morning and are trying to figure out if there is a solution in which each girl can ask someone to go to the prom with her, so that their offer is accepted. Lets assume that the boys will accept any proposal that is made, if they do not get asked by another girl within a day of getting asked by some girl. The mothers on the other hand, do not want their daughters to have their offer rejected as it could lead to a tense situation!

The mothers come up with the following matching $(Margaret, Tom), (Jane, Bill), (Amy, Don), (Katherine, Chris)$. Each mother tells their daughter to make the appropriate proposal, and is pleased to find out that her daughters proposal was accepted a day later.

How did they come up with this solution? To describe the algorithm, lets play the following mental game: suppose Margaret asked Bill and he accepted. Amy asked Tom and he accepted. Katherine asks Don and he accepts. Finally, Jane asks Bill. (Even though no-one has asked
Figure 2: The graph construction.
Chris as yet, Jane does not want to go with him and does not ask him.) Bill says that he will accept Jane’s offer, provided he can get a friend to go with Margaret (and it has to be someone Margaret likes). Bill then asks Tom to go with Margaret. Tom has already said yes to Amy, but again is willing to go if he can find someone who Amy likes. Amy likes Don, but Don has already said yes to Katherine. Now we can “shift” Katherine to Chris, freeing Don. Amy can now go with Don, freeing Tom, Margaret can go with Tom, and Jane finally gets to go with Bill. However, if the mothers knew all this before, they could confer and compute this solution before any actual proposals are made. Each person then only makes one offer, which is accepted. The main idea is that we can start with any initial pairing, that matches $k$ compatible pairs together. The algorithm can then iteratively “improve” this pairing to add one more valid pair. However, this involves changing the current solution. This process continues until no further improvement is possible. At that stage we can prove that if the algorithm fails to increase the number of compatible pairs, then no better solution exists.

Suppose we change the preference lists in the following way: suppose Amy hears that Don is not such a nice person after all, and does not want to go with him. The current solution might be (Margaret,Bill) (Amy,Tom) and (Katherine,Chris). Jane can only go with Bill and if we shift Margaret to her remaining choice Tom, then there is no other valid choice for Amy. In this case, the algorithm fails to find an improvement, because none exists. In fact, note here that the three girls, Margaret, Jane and Amy, are only willing to go with either Tom or Bill. Clearly there can be no solution which finds a valid pairing for all three girls since they are interested in only two boys. In fact, Hall proved that there is always a perfect matching unless there is a collection of $p$ girls, such that there are less than $p$ boys they are willing to go with. (In this situation, $p = 3$.)
10 Internet Search

In 1995 when the world wide web was fairly new, I tried to search for information on the Grand Canyon. Many of the web sites that came up were of the type “Here are the pictures of my trip to the Grand Canyon”. However, these were not the web pages I was looking for! I was looking for the web page of the National Park Service. This page was also there but was buried a bit lower down. However, try doing this search today on a modern search engine. The National Park Service web page now comes up as the very first hit on Google. Clearly something changed. A new method is being used to rank relevant pages when you search for information, this new method has its origins in two papers that were based on analysing the link structure of the web (the links are the hypertext references that web pages have to one another).

In two remarkable papers that appeared around the same time – Kleinberg’s HITS algorithm [?] and the Pagerank algorithm due to Brin and Page [?]), were proposed as ways of using the “web” graph structure. This may appear at first glance to be somewhat surprising, but in fact this has emerged as one of the powerful tools in the variety of tools employed by search engines. (The details are closely guarded company secrets.)

With the amazingly rapid increase in the popularity of the web in 1994, literally thousands of new web pages were being created every day. It would not be possible to create a master “index” of this collection of web pages.

Why is the search problem hard? First of all is the question of interpreting the users query – the same word may mean completely different things to different people. If I am searching for “Fit” am I looking for information on the Honda Fit? Perhaps I was looking for the Fashion Institute of Technology and was too lazy to type “FIT”, or perhaps the Florida Institute of Technology. Or perhaps I was looking for information to simply stay fit.

The main point to show is how graph theory can be used to help figure out which web page is more important than some other web page.

Moreover, the problem is compounded by the fact that spammers are involved. Suppose I sell water filters as a side business to help supplement my income as a professor. I set up a web page that gives information about my filters. I observe after a few weeks that very few people are looking for water filters (say by monitoring the level of traffic to my web site) or I find out that very few people in the world are looking for water filters. Suppose most people are searching for suppliers of bottled water – by simply embedding the word “bottled water” within my web page, I could try to ensure that my web page is one of the ones displayed by a search engine when people search for “bottled water”. Maybe a lot of people are searching for “Melanie Oudin”, perhaps by putting her name in my document I can get people to look at my web page for water filters!

In addition, we may be searching for “a good search engine” and would like a list of relevant URL’s. However, good search engines are unlikely to actually contain this as a keyword....similarly if we are searching for “auto makers”, it is unlikely that Honda would have this keyword on their web page. In addition, sometimes people make a mistake in spelling the name of a person they are looking for, or do not know the exact spelling. Search engines do a remarkable job, and also sometimes indicate that perhaps we intended to type a different search query. How did they figure this out? At least by now it should be sufficiently clear that this is a tricky problem. The goal of a search engine should be to find the most relevant information for a query.

The first generation of search engines were based on the following simple idea. Suppose we
are searching for “Madonna”. Abstractly any document could be considered as a set of words (in unordered form). The search engine would first create a “master index” for the entire World Wide Web (WWW). This means that for every possible keyword, a list of all the web pages that contain that keyword will be stored. For example, all the web pages that have the word “Madonna” will be stored, so that when a user types this query, all the relevant documents can be returned. However, this creates the following problem – if a keyword is very popular, the set of web pages that contain this keyword is very large. What if 10,000 documents contain the name “Madonna”? How do we find the most relevant web pages?

To understand some of the ideas behind this we need to study a small amount of Graph Theory.

Let us start by introducing the concept of undirected graphs. Undirected graphs are composed of two basic objects – vertices or nodes and edges. We usually denote the set of nodes by $V$ and the set of edges by $E$. An edge is simply an unordered pair of vertices. Suppose we create a graph on four vertices $V = \{1, 2, 3, 4\}$ and $E = \{(1, 2), (1, 3), (2, 3), (3, 4)\}$ (see Fig. 5). An edge $(a, b)$ is said to connect $a$ and $b$, and we also say that the edge is incident on/to vertices $a$ and $b$. The edge can be written as $(a, b)$ or $(b, a)$, both refer to the same edge. The degree of a vertex is the number of edges incident to it. For example, the degree of node 1 is 2, the degree of node 2 is 2, and the degree of node 3 is 3, and the degree of node 4 is 1. Note that the degree of a node can range from 0 (no edges incident on the node) to $N - 1$ where $N$ is the number of vertices in the graph.

How many edges can an undirected graph have? One can argue that an undirected graph can have at most $\frac{N(N-1)}{2}$ edges. If every node has degree $N-1$ (edges to every other node), and there are $N$ nodes in all, then it might seem at first glance that the total number of edges should be $N(N-1)$. However, every edge was counted exactly twice so we have to divide by 2, to get the exact answer. (This is a different argument from the one we did in class.)

On the other hand, in directed graphs edges (often called arcs) are associated with an orientation. An edge $(a, b)$ is an edge that points from $a$ to $b$. The in-degree of a vertex $v$ is the number of edges of the form $(u, v)$, namely a count of the edges that point from some node to $v$. The out-degree of a vertex $v$ is the number of edges of the form $(v, c)$, namely a count of the edges that point from $v$ to another node. For example the in-degree of node 1 is 0, and the out-degree of node 1 is 2. The in-degree of node 2 is 2 and the out-degree of node 2 is 1. In a directed graph we may have an edge $(a, b)$ as well as an edge $(b, a)$. Note that a directed edge may have up to $N(N-1)$ directed edges.

We will learn a fair amount of graph theory as we go along, and we will introduce new ideas and concepts whenever we need them.

We now discuss the central idea developed by Kleinberg in the HITS algorithm. Let $D_1, \ldots, D_n$ be a list of $n$ documents (web-pages). The main insight is to view the web graph as
a directed network, where there is a vertex $i$ corresponding to each document $D_i$. If document $D_i$ has a hyper-link to another document $D_j$ then we add an arc from node $i$ to node $j$. This defines the “web graph”.

Suppose we type in a search query for keyword $\sigma$. We can extract from the graph the root set of documents $R_\sigma$ - this is the collection of documents that are relevant for the query $\sigma$. (These could be precisely the set of documents containing the keyword $\sigma$, or a set of documents that are considered to be relevant for the query, say using a first generation search engine.) The key point is that if there are a lot of documents in the root set then we somehow need to choose the most relevant documents. This root set $R_\sigma$ corresponds to a set of vertices $V_\sigma$ in the web graph that we construct and process. Note that this graph is constructed after the query is made, so it is sensitive to the query $\sigma$.

Next we expand the the root set of nodes $V_\sigma$, by adding all nodes corresponding to documents that are linked to by the documents in the root set $R_\sigma$ as well as documents that link to documents in the root set $R_\sigma$.

The graph that we work with is this expanded node set $S_\sigma$ and we include all arcs between pairs of nodes in $S_\sigma$.

A Hub node is a node that provides information about good Authorities, and an authority is a node pointed to by good hubs. The main idea is that web pages are either hubs or authorities, or a bit of both. In the end the algorithm will assign a hub score and an authority score to each node. These scores will be between 0 and 1. The higher the score, the better the hub/authority the node is. The method works by slowly modifying the initial scores in rounds. Each node initially has a hub and authority score of 1. Let us denote a node’s hub score (in round $t$) by $h^t[i]$ and its authority score by $a^t[i]$. In round one, we have $h^1[i] = 1$ and $a^1[i] = 1$.

For $t = 2 \ldots k$:
We define $a^t[i] = \sum_{j : (j,i) \in E} h^{t-1}[j]$. Similarly we define $h^t[i] = \sum_{j : (i,j) \in E} a^t[j]$.

After each iteration, the algorithm “normalizes” the scores so that $\sum_i (h^t[i])^2 = 1$ and $\sum_i (a^t[i])^2 = 1$. The algorithm does this for a certain number of iterations until these scores converge to stable values. Its not clear that this will ever happen, or quickly, but for that you really have to read Kleinberg’s paper. Its not that crucial for understanding the basic algorithm however.

---

3 This is an interesting departure point from the other method we will study - pagerank, where a fixed pagerank value is computed for every web page in the web graph. After the query $\sigma$ is issued, then we return the set of pages $R_\sigma$, but in the order of decreasing page rank value.

4 We can safely assume that a node’s out-degree is not too large (at most a few hundred), a node’s in-degree cannot be bounded similarly. There could be certain popular web pages that a lot of people link to. For example, perhaps millions of people link to Michael Jackson’s web page. Hence Kleinberg selects a small subset of such in-links. Kleinberg also suggests other methods to identify links that intuitively do not confer authority, such as removing navigational links.
11 Greatest Common Divisor Algorithm

Suppose we would like to simplify the fraction $\frac{30}{36}$. We can divide the numerator and denominator by 2, and obtain $\frac{15}{18}$. Can we simplify this further? We can divide the numerator and denominator again by 3, and get $\frac{5}{6}$. However, we may be doing this for a while. A simpler method is to compute the Greatest Common Divisor (GCD), and then divide the numerator and denominator by the GCD. In this case, the GCD of the two numbers is actually 6. Note that dividing the numerator and denominator by 6 directly gives us the simplest form of the fraction $\frac{5}{6}$. However, we still need to figure out how to compute the GCD.

Here is another interesting reason why computing the GCD may be useful. Suppose a country prints postage stamps in two possible values - a postage stamp worth $x$ patacas and another postage stamp worth $y$ patacas. Suppose that we have to stick $n$ patacas worth of postage, then we would like the ability to stick some stamps worth $x$ patacas and some stamps worth $y$ patacas, and put the exact postage. Note that with some choices this is not possible. Suppose we have a stamps worth 2 patacas, and stamps worth 6 patacas, we cannot generate any odd value. We can stick postage worth 2, 4, 6, 8, 10 etc. Suppose we have stamps worth 3 and 6 patacas, then we can stick postage worth 3, 6, 9, 12, etc. It turns out if we have the property that GCD of $x$ and $y$ is 1, then we can always generate any value $n$, as long as $n \geq n'$ for some value $n'$. Taking $x = 2$ and $y = 5$ is a good example. First note that the GCD of $x$ and $y$ is 1. We can generate 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, etc. in postage; in other words, for all $n \geq 4$ we can generate $n$ units of postage. For example, we can get 7 units of postage by sticking one stamp of 2 and one of 5. We can get 8 units of postage by sticking 4 stamps of 2, we can get 9 units of postage, by sticking 2 stamps of 2 each, and one stamp of 5.

The GCD is a fairly basic function to compute, and in fact even the ancient Greeks were trying to compute the GCD of a pair of numbers. Suppose we wish to compute the GCD of two numbers $a$ and $b$. A simple algorithm to compute the GCD is as follows. We can check every possible value $i$ (starting from 1) to see if $i$ divides both $a$ and $b$. Let us try an example – suppose $a = 30$ and $b = 36$. Clearly 1 divides both numbers, 2 also divides both numbers, 3 also divides both numbers. However 4 divides 36, but not 30, and 5 divides 30 but not 36, but 6 divides both numbers, 7 does not divide either number, and neither does 8. 9 divides 36, but not 30; 10 divides 30 but not 36, 11 divides neither number. 12 divides 36, but not 30, 13 and 14 divide neither. 15 divides 30 but not 36, 16 and 17 divide neither. 18 divides 36, but not 30...you get the idea, this is a bit of a pain. 6 is the largest number that seems to divide both, so 6 is the answer.

Euclid’s algorithm for computing the Greatest Common Divisor of two numbers $a$ and $b$ is one of the oldest known algorithms. The GCD of a pair of numbers is a number that divides both numbers perfectly, and at the same time is the largest number with this property. For example, the greatest common divisor of 30 and 36 is 6. The common divisors of these two numbers are 2,3 and 6 with 6 being the largest common divisor. The algorithm is simple, but not at all immediately obvious.

Assume that $a > b \geq 0$. In the algorithm, we first check if $b$ divides $a$. If so, we are done. $b$ is the GCD. Otherwise, we remove $b$ from $a$ to obtain the pair $b$ and $a - b$. We apply the same algorithm to this pair of (smaller) numbers.

For example, take the numbers 12 and 3. Since 3 divides 12, the GCD is 3. Now consider 12 and 9. 9 does not divide 12. So we now consider the pair 9 and 12-9 = 3. We apply the algorithm

---

*A pataca is the currency of Macao.*
to this pair of numbers. Since 3 divides 9, the algorithm terminates with 3 as the answer. Let us apply this algorithm to the pair 30 and 36. 30 does not divide 36, so we now look for the GCD of 36 - 30 = 6 and 30. Since 6 divides 30, we have our answer, its 6. We are done. This was a lot simpler!

Why does this method work? This argument may be skipped if you are really not interested. However, before you skip it, I would suggest trying to read it at least once. We next argue that \( \text{GCD}(a, b) = \text{GCD}(a - b, b) \) when \( a > b \) and \( b \) does not divide \( a \). (When \( b \) divides \( a \), it is trivial to see that the GCD is \( b \) itself.) Suppose the GCD of \( a \) and \( b \) is \( X \). We would like to show that \( X \) is also the GCD of the pair \( a - b \) and \( b \). To prove this we need to show that a) \( X \) is a divisor of \( a - b \) and \( b \); and b) \( X \) is the largest such number. If \( a = X \times Y \) and \( b = X \times Z \) then \( a - b = X \times (Y - Z) \) and hence \( X \) is a divisor of the pair. We now argue that there is no divisor of \( a - b \) and \( b \) that is larger than \( X \). Suppose such a divisor \( X' \) exists, then we have \( a - b = X' \times Y' \) and \( b = X' \times Z' \). Hence \( a = X' \times Y' + X' \times Z' = X' \times (Y' + Z') \). This is a contradiction to the assumption that \( X \) is the largest divisor of the pair \( a, b \).

The key insight is to show that either \( b \) divides \( a \) or \( \text{GCD}(a, b) = \text{GCD}(b, a - b) \). Let us apply this algorithm to the numbers 99 and 81. \( \text{GCD}(99, 81) = \text{GCD}(81, 18) = \text{GCD}(18, 63) = \text{GCD}(18, 45) = \text{GCD}(18, 27) = \text{GCD}(18, 9) \). The algorithm now terminates with 9 as the answer. Note that at the stage when the algorithm has the pair 81, 18 we can actually jump directly to 18,9 after subtracting 18 from 81 four times. This can substantially reduce the number of steps that the algorithm takes to compute the GCD. In fact, we argue next that this is indeed a very fast algorithm. Another (equivalent) way to describe the algorithm us by using the mod operation. \( a \mod b = r \) where \( r \) is the remainder when we repeatedly subtract \( b \) from \( a \) as long as we are left with a non-negative number. Hence \( 5 \mod 3 = 2 \). We subtract 3 from 5 once, and cannot subtract 3 again, otherwise we will get a negative number. Similarly \( 11 \mod 5 = 1 \). We subtract 5 from 11 twice and the remainder is 1.

Hence \( \text{GCD}(a, b) = a \) if \( b = 0 \) otherwise \( \text{GCD}(a, b) = \text{GCD}(b, a \mod b) \).

Number of steps taken by Euclid’s Algorithm: There is a rather interesting connection between the running time of Euclid’s algorithm, and Fibonacci numbers \(^6\).

If Euclid’s algorithm takes \( k \) steps of re-writing the pair \( (a, b) \) then we can prove that \( a \geq f_{k+2} \) and \( b \geq f_{k+1} \). From this it follows that if \( b < f_{k+1} \) for some value \( k \), then the algorithm takes at most \( k \) steps. Each “step” involves computing \( a \mod b \) which is a non-trivial operation for large numbers. However, we can assume that this can be performed in a polynomial number of steps in the size of the numbers.

For example, if we have a pair of numbers \( (a, b) \) for which we are trying to compute the GCD, then suppose \( b < f_{k'+1} \) for some value \( k' \). If \( b = 12 \) then \( k' = 6 \) since \( f_7 = 13 \). In this case, there are at most 6 computations of the mod function.

The proof of the claim is by a method called induction. We first prove that the claim is true for \( k = 1 \). (If we are claiming that the claim is true for all integers \( k > 0 \) then it must be true for \( k = 1 \).) If we perform one mod computation then we know that \( b > 0 \). Since \( a > b > 0 \) we have that \( b \geq 1 \) and \( a \geq 2 \). Thus \( a \geq f_3 = 2 \) and \( b \geq f_2 = 1 \). We now show that if the claim is

\(^6\)Fibonacci numbers are easy to define. Let \( f_0 = 0, f_1 = 1 \). Define \( f_k = f_{k-1} + f_{k-2} \). Thus \( f_2 = 1 + 0 = 1 \). Similarly, \( f_3 = f_2 + f_1 = 1 + 1 = 2 \). \( f_4 = f_3 + f_2 = 2 + 1 = 3 \), and \( f_5 = f_4 + f_3 = 3 + 2 = 5 \). Note that \( f_6 = f_5 + f_4 = 5 + 3 = 8 \). \( f_7 = f_6 + f_5 = 8 + 5 = 13 \). These number actually increase very rapidly, and have what is called “exponential growth”. These numbers were originally studied in the context of rabbit populations. If we start with a pair of rabbits, and each female rabbit delivers a pair of offsprings, every month starting at the age of 2 months, then the number of pairs of rabbits after \( n \) months is \( f_n \). This assumes that the rabbits do not die (if they have a life span of a few years, then this assumption is true for the few years that we decide to breed rabbits). This also assumes that half the rabbits born are female.
true for $k = m - 1$ (induction hypothesis) then it must also be true for $k = m$. Thus the claim is true for $k = 2$, since it is true for $k = 1$. If its is true for $k = 2$ then it must be true for $k = 3$ etc. Suppose we have $m$ computations of the mod function. In the first mod computation we replace $GCD(a, b)$ by $GCD(b, a \mod b)$. After this if we only have $m - 1$ mod computations then we can claim that $b \geq f_{m-1+2}$ and $a \mod b \geq f_{m-1+1}$. This must be true by the induction hypothesis. Note that $b$ is the first parameter and $a \mod b$ is the second parameter. Since $b \geq f_{m+1}$ the claim about $b$ follows. Note that $a \geq b + a \mod b$. This is because $a > b$. By the induction hypothesis the right hand side (RHS) is at least $f_{m+1} + f_{m} = f_{m+2}$. Thus $a \geq f_{m+2}$. This completes the proof.
12 Intro to Ruby

You can find many online resources for Ruby – this is a brief introduction to the basics about programming in Ruby. We discuss three rather simple programs.

This program defines two variables, a and b and then sets their values to 100 and 21 respectively. Variables are just names for memory locations, and we can refer to the value inside a memory location by using the name of the variable.

The program then prints the values of the variables a and b and then computes various functions of these two values and prints them out.

```ruby
# scalar variables
a = 100
b = 21

print "a is #{a}, b is #{b}
"

# arithmetic
s = a + b
D = a - b
p = a * b
q = a / b
m = a % b
e = a ** b

print "sum (a+b) is #{s}\n"
print "diff (a-b) is #{d}\n"
print "product (a*b) is #{p}\n"
print "quotient (a/b) is #{q}\n"
print "mod (a%b) is #{m}\n"
print "exponent (a**b) is #{e}\n"
```

This program prints out the squares of all numbers up to 24. The variable i has no meaning outside the loop. The command inside the { } is executed 25 times, starting from i = 0.

Max=25

```ruby
Max.times{|i| print('The square of #{i} is #{i*i}\n') }
```

The third program is the most interesting one since it makes decisions based on the input data. We have an array A, and we would like to find the index (position) of the largest element in the array. In other words, we would like the function `max` return the index `bigindex` such that `A[bigindex] ≥ A[i]` for all i.

Notice that in the very first iteration, nothing happens since we compare `A[0]` and `A[0]`. In the second iteration when `i=1` then we compare `A[0]` with `A[1]`. In general for any value of i
what is true is that $A[\text{bigindex}]$ represents the position of the largest element chosen from the array until position $A[i-1]$. We then compare this largest element with $A[i]$ and update the value of bigindex if we found a larger element.

A=Array[3,6,1,17,-22,11,2,5,6,7]

def max
    bigindex=0
    A.length.times{|i|
        if (A[i]>A[bigindex])
            bigindex=i
        end
    }
    return bigindex
end

print(‘‘The max index is #{max}\n’’)
13 Graphs and Facebook

Facebook is a social networking website. From my point of view it is simply a graph – people correspond to nodes in this large graph, and edges correspond to friendship links (neighbors). Let us consider a very basic computational problem faced in Facebook. One of the things that Facebook does is to make “friend suggestions” - these may be based on several factors, however once you join the network, one possible rule that Facebook might use is to consider pairs of individuals that have common friends, and then examine their data and interests and make recommendations for a subset of these pairs.

Let's say I join Facebook, and invite 20 other people to become my friends in Facebook. If 18 accept my invitation, then I have 18 friends - my degree in this graph is 18. However, Facebook might observe that I am friends with Kristin, and Jessica is also Kristin’s friend, and might make a “friend suggestion” to me about Jessica since we have a common friend (Kristin). (In practice, Facebook may use other parameters as well such as only suggesting people who have multiple friends in common, or might look at other data as recommending friends who attended the same college and have a friend in common.)

How can we determine which pairs of individuals who are currently not friends, have a friend in common?

This is a fairly simple computational problem, but as we will see it also leads to an interesting discussion about representing the Facebook graph as well as the issues that are relevant while designing algorithms. After we produce a list of such individuals, we might want to further analyse this list to decide which ones to actually recommend.

We will assume that the graph has \( N \) nodes in all. Current estimates put \( N \) in the range of several million users. Let’s assume for simplicity that we have ten million users\(^7\); this suffices to make our main point. Most users will have a much smaller list of friends - let’s assume that we bound the maximum number of friends a user can have by \( \Delta \) (this in effect is an upper bound on the degree of any node in the graph). Let say that we set this bound to 1000; this seems like a reasonable number, (except for those few insanely popular people).

Let \( d_i \) be the degree of node \( i \). Let \( V \) be the set of nodes and \( E \) be the set of edges.

We first make some simple observations about the graph itself.

\[
2|E| = \sum_{i \in V} d_i
\]

This essentially says that if you add the degrees of all the nodes in the graph you get exactly twice the number of edges. (Please draw a few graphs to convince yourself of this.)

For an arbitrary graph since the degree of any node is at most \( N - 1 \), we can easily show that \( 2|E| \leq N(N - 1) \) by simply saying that \( d_i \leq N - 1 \) and adding the \( N \) terms gives us a bound of \( N(N - 1) \).

Also note that \( d_i \leq \Delta \) by our assumption on the Facebook graph. Thus for this graph we can replace the upper bound of \( N - 1 \) by \( \Delta \) and show that \( 2|E| \leq N \times \Delta \).

Let us now focus on our original problem of finding all pairs of nodes with the property that they are not friends, but have at least one common friend. Let’s try the following algorithm, we call it the “Generate Pairs” algorithm.

First make a list of all possible pairs of nodes. From this list we shall eliminate the pairs of nodes that are already friends. Suppose our list of all possible pairs (see Fig. 4) is \{ (Hailey,
Figure 4: Figure showing a small social network. Pairs of nodes that have a friend in common are also shown.

For each pair on this list we need to check if they have a common neighbor. In other words, if we consider the pair (Samir, Jessica) we need to examine all the neighbors of Samir and all the neighbors of Jessica and determine if the two lists have a common element. So if the graph does have the edge (Jessica, Kristin) and the edge (Kristin, Samir) then we will know that the two do indeed have a common friend (there may be others as well). Note that Hailey and Jessica do not have any common friends. If there is no common friend we can remove the pair from the list. This check is not trivial and may take time proportional to the degree of a node $d_i$. To derive a crude upper bound on the running time of the algorithm let us assume that we need to do the following. For each pair of nodes $(X, Y)$ we have to go through all the neighbors of $X$ and for each such neighbor check to see if any of them are neighbors of $Y$. This may take upto $\Delta$ steps (since we have to check go over all the neighbors of a node).

After removing the existing friends from the list, the list of pairs can still be quite large. In general we will have $\frac{N(N-1)}{2} - |E| \geq \frac{N}{2}(N - 1 - \Delta)$ such pairs on our list. (We discussed this point earlier – this was the number of handshakes when $N$ people were at a party and everyone shook hands with everyone else. We subtracted $|E|$, the number of edges since we only want to make a list of the possible pairs that do not have an edge in common and we know that $2|E| \leq N \times \Delta$, so by subtracting an even larger quantity ($N \times \Delta$) we get a smaller number.) However for each such pair we are spending “roughly” (this is a slight over count clearly) $\Delta$ steps to check if they have a common neighbor. The total number of steps is therefore roughly $\Delta \frac{N}{2}(N - 1 - \Delta)$. If $\Delta = 10^3$ and $N = 10^7$ then we get a rough estimate of $5 \times 10^{16}$. This is a HUGE number! Even if a computer performs $10^9$ operations per second, this will take $5 \times 10^7$ seconds. A day has 86,400 seconds. This is almost 600 days! (In class we got an answer that was around 10 days, but that was for a million node graph.)

Let us consider a slightly different method next. We call this the “Node Centric” method. Our focus will not be to generate a list of pairs of non-adjacent nodes but simply to focus on an individual $X$. This person has at most $\Delta$ friends. Suppose we consider all the friends of friends - how large can that set be? Observe that this set has size at most $\Delta^2$ (try a simple example to check this). By definition, any such node $Y$ in this set already has a common neighbor with $X$. All we need to check for each such $Y$ is that if they already have an edge to $X$ or not. As long as we can do this quickly, we are in business and can generate all the potential pairs involving $X$ in $\Delta^2$ computational steps. However, we still have to repeat this for every possible choice $X$ and this increases the computation by a factor of $N$. In any case this method takes no more than $N\Delta^2$ steps, which is $10^7 \times (10^3)^2$ or $10^{13}$. If we can perform $10^9$ computations per second, then it takes $10^4$ seconds compared to 600 days by the previous method. This is less than 3 hours of
computation time. Note that we were able to achieve this by a simple change of the algorithm. Can we do any better?

We will next discuss graphs and their representation. Clearly we cannot represent a million node graph by drawing all the links.
14 Arrays

Often we need to store a large collection of values, and not just a single value. Say for example we wish to compute the average height of the students in the class. To do this, we first need to add up the heights of all the students in the class. We use something called an “Array” to store the heights. Let A be the name of the array. The array essentially allows us to create a large set of variables that we can access by referring to A[0], A[1] etc. Each A[i] is actually an integer variable storing the height of the \(i^{th}\) student.

Suppose we have a set of students with heights 62, 65, 64, 68, 63, 62 (in inches). The following Ruby program will compute the average height.

```ruby
sum=sum+A[0]
sum=sum+A[1]
sum=sum+A[2]
sum=sum+A[3]
sum=sum+A[5]
A=Array[62,65,64,68,63,62]
sum=0
A.length.times{|i| sum=sum+A[i] }
print(''#{sum/A.length}''
```

The main idea is to create a variable called “sum”. In this variable we store the sum of all the elements seen so far as we scan the array. We initialize the value of sum to 0, and then first add A[0] to sum, then we add A[1] to sum, then A[2] to sum etc.

A.length tells us the value of the length of the array. In this example, this will be 6. We will iterate over the array with \(i = 0, 1, 2, 3, 4, 5\) and essentially do the following commands.

At the end, sum contains the sum of the six elements in the Array A. To compute the average height, we divide by the number of entries in the array, which is given by A.length.

15 Looking up information

There are many many applications where we need to look up information in extremely large tables. Consider the following simple problem faced by a cellular provider that has millions of customers. Cell phone towers have a small coverage range – as we travel by car or by train, we move from connecting to one cell tower to another. How do calls then get routed to a user who is moving around from one region to another? If I call the person’s number – how does the phone network “know” which cell tower to route the call to? One simple solution would be to create two large “tables”. The first table (an array) A stores all the phone numbers in order. In other words, A[0] is the first phone number (smallest phone number say 2022132637), A[1] may be the next one, 2022132641 etc. The second table stores the cell phone towers that users are currently close to. For example if A[j] = 3012373100 then in B[j] we store the cell tower number that this phone number is currently connected to. If this phone moves from cell tower 317 to cell tower 419, then we change B[j] from 317 to 419. It is partly due to these rapid changes that cellular companies disallow the use of cell phones on airplanes, while they are in flight.
How do we look up the location of a particular phone? Given the phone number \( Y \), we want to “find” \( i \) such that \( A[i] == Y \). Once we have \( i \), \( B[i] \) gives us the location of phone \( Y \). As the customer moves from region to region, the phone sends messages whenever it registers with a new tower to update the table so that calls can be routed to the correct cell tower.

This motivates the need to look up information in large tables. The most amazing thing about the searching method is that even in a table with one million entries, by considering only 20 entries, we can find the correct location where a given phone number is stored. This is simply amazing! Later on we will discuss why we only need to look up no more than 20 entries. We accomplish this using a simple but powerful idea called “binary search”.

The key idea behind binary search is to order the entries and then consider the “middle” entry. If the value we are searching for is exactly the middle entry, then we are done - we have found the value. If it is smaller than the middle entry then we need to search for the value in an array of half the size (the left half) and if the value we are searching for is larger than the middle entry then we need to search in the right half of the array.

We define two indexes into the array – called \( \text{low} \) and \( \text{high} \). The portion of the array we are searching in is the portion going from \( A[\text{low}] \) to \( A[\text{high}] \). Initially \( \text{low} =0 \) and \( \text{high} = A\text{.length} \). At each step, by making a single comparison we are able to narrow down the search by either increasing \( \text{low} \) to \( \text{mid}+1 \), or by decreasing \( \text{high} \) to \( \text{mid}-1 \). The binary search repeatedly compares \( y \) to \( A[\text{mid}] \) until we find it, or \( \text{low} \) becomes equal to \( \text{high} \). In this case, our array has only one element – if this element is not \( y \) then it does not exist in the array. (As an exercise you should try to run binary search to look for entries not in the array.)

To do a binary search in an array containing \( N \) elements, it takes \( \log_2 N \) comparisons. Thus if we do binary search in an array containing 1000 elements it only takes 10 comparisons. For a million elements it takes 20 comparisons. You may want to at least think about why this is the case.

```ruby
# Binary Search
A=Array[3,7,9,11,15,19,27,29,31,45,56,78,99]

def find(y)
  low=0
  high=A.length
  
  while (low < high)
    mid = (low+high)/2 #define the middle index
    print("#{mid}\n") #this is just a check to see what comparison was done
    if (A[mid]==y) #this is when you find the value
      return mid
    elsif (A[mid]<y)
      low=mid+1
    elsif (A[mid]>y)
      high=mid-1
    end
  end

  if (A[low]==y) then
```

```
return low
else
  return -1  # return a special code if y was not found
end
end

# the following lets us provide the input argument from the command line
# ruby find.rb 9 will search for element 9 in the input array, and return 2
if (ARGV.length == 1)
  p = Integer(ARGV[0])
end

print("The index of #{p} is #{find(p)}\n");
We had several discussions about graphs and even informally discussed some simple algorithms that processed graphs to achieve certain objectives. However we only drew pictures of graphs – computer algorithms however need other ways of representing graphs, so that they can be created and manipulated efficiently. The precise representation used will depend on the exact objectives – representations often have different properties and some would enable certain graph operations efficiently, while others would enable other operations efficiently. The two main graph representations we will talk about are the **Adjacency matrix** representation and the **Adjacency list** representation.

An adjacency matrix is essentially a two-dimensional array. Earlier we learnt how to create a one dimensional array in Ruby – two dimensional arrays can be created as well. To create a two dimensional array, we first create a one dimensional array, and define each array entry as a new one-dimensional array.

The following piece of code defines a two dimensional array \( a \).

```ruby
def mda(width,height)
a = Array.new(width)
a.length.times{|j| a[j]= Array.new(height) }
return a
end
```

We can refer to elements in the array by writing \( a[3][4] \). We are now ready to define the adjacency matrix. We set \( a[i][j] \) to 1 if there is an edge in the graph connecting nodes \( i \) and \( j \) and 0 otherwise. Just by looking up the entry \( a[i][j] \) we can know if there is an edge in the graph connecting nodes \( i \) and \( j \).

The only problem with this representation is that it uses a lot of storage space – for example for a graph with 10 million nodes \((10^7)\), we need storage space of \(10^{14}\) since there is one entry for every pair of nodes. One Kilobit (Kb) is \(10^3\) bits. One Megabit (Mb) is \(10^6\) bits, and one Gigabit (Gb) is \(10^9\) bits.

While this representation makes it easy to look up whether or not a pair of nodes \((i,j)\) are adjacent or not, the main problem is that the space used is very large. In addition, most of the entries in the array \( a[i] \) are actually 0, so we cannot easily find out the neighbors of node \( i \) – we have to scan the entire array \( a[i] \) to find the neighbors of \( i \). This is unacceptable, since node \( i \) may only have a small number of neighbors compared to the number of nodes in the graph.

A better representation is to store in \( a[i] \) all the neighbors of node \( i \). Since this is an array, it is easy to store a large set of neighboring nodes (upto say our maximum degree \( \Delta \)).

An array is associated with each node in the graph, and contain all the neighbors of the node. An array is a one dimensional structure.

We can create a graph with \( N \) nodes by defining a collection of \( N \) arrays \( G[i] \).

\[
N=4
G=Array.new
# want G.length to be the number of nodes in the graph
#G[i] is the array corresponding to the neighbors of node i.
N.times{|i| G[i]=Array.new}
```

How do we write into a specific box?
G[0][2] will index into the 3rd box of G[0]. So we can set G[0][2]=3 to indicate that node 0 has neighbor node 3.

Suppose we wish to represent the following graph (see Fig. 5) in the adjacency list form.

The following code in ruby will construct the appropriate adjacency list representation. Note that node 0 has two neighbors – namely nodes 1 and 2 and we append to G[0] nodes 1 and 2 etc.

```ruby
N=7
G=Array.new
N.times{|j| G[j]=Array.new }
G[0]<<1<<2
G[1]<<0<<2<<6
G[2]<<0<<1<<3<<4
G[3]<<2<<4
G[4]<<2<<3<<5
G[5]<<4
G[6]<<1
```

Now that we have constructed this representation we can do a collection of interesting things. **List the neighbors of a node**

We now describe a function that will list all the neighbors of a given node. This function is extremely simple.

```ruby
def listnbrs(i)
  G[i].length.times{|j| print G[i][j]}
  print "\n"
end
```

**Count the total number of edges in a graph**
The simple idea is that if we add up the degrees of all the nodes, then we get a quantity that is exactly twice the total number of edges.

```ruby
edgecount = 0
#In each iteration we add up the degree of node i to edgecount
G.length.times{|i|edgecount = edgecount+G[i].length}
print edgecount/2
```

**Generate Friend suggestions**

This is a bit more complicated than the two examples we just discussed. This ruby program reads as input a node number p, and then generates all the friend suggestions for this node. In other words, we would like to list all nodes that are neighbors of p’s neighbors, but not already adjacent to p.

In addition we create a function `notadj(a,b)` that returns true if the given pairs of nodes are not adjacent. We scan the adjacency list of the first node looking for the other node. If we find it, we return `false`, otherwise we return `true`.

The main function is called `recpairs(j)` that examines the adjacency list of node j and for each node x on that adjacency list, goes through the adjacency list of node x and checks to see if the neighbors of x are adjacent to j or not. If we find a node (other than j) that is not adjacent to j then we need to output it as a pair along with j.

```ruby
# Recommend Pairs of Non-adjacent Nodes with common neighbors

def notadj(a,b)
    #returns true if a and b are not adjacent
    #search for b in a's adjacency list
    G[a].length.times{|i|
        if (G[a][i] == b)
            return false
        end
    }
    return true
end

def recpairs(j)
    #generate all recommendations for node j
    #these are nodes at distance exactly 2
    #x is defined as a neighbor of j in each iteration
    G[j].length.times{|i|
        x=G[j][i]
        #now look at x’s adj list to see which nodes are to be chosen
        #if the node is j, we skip it and if its adj to j we skip it
        #ok to print duplicates
        G[x].length.times{|f|
```
if ((not(G[x][f] == j)) and notadj(G[x][f], j))
    print("(#{j},#{G[x][f]})")
end
}
}
end

N.times{|p|
    recpairs(p)
    print("\n")
}
Searching Graphs using DFS

The title might seem a bit strange - precisely what could we mean by “searching” a graph? We will discuss a particular method for exploring graphs called depth first search. We recently learned how graphs could be compactly represented on a computer using arrays. What kinds of questions might we want to ask about graphs now?

One of the most elementary computations is the following – suppose the graph represents a map, and we would like to go from node A to node B – how can we accomplish that? We are also interested in actually finding the shortest route to go from node A to node B. We could assume that in some way the length of edges in the graph have been encoded, so for example we could have a map of cities on the east coast with distances marked, and try to compute a shortest route to go from one place to another. In fact, when you use Google maps, this happens in some way. We first focus on the simpler problem first of simply finding one route from A to B.

Our algorithm is itself rather simple - but we do need some mechanism for keeping track of which nodes have been visited already. Our search algorithm will start from one specific vertex \( p \) and output a list of all the nodes that are reachable from vertex \( p \).

Along with the graph representation, this time we also define an array called \( V \). The entry \( V[i] \) will be either true or false. The entry is false initially, and when we visit a node \( i \), we change \( V[i] \) to true. (You could think of \( V \) for “visited”.)

When we first reach a node that is not yet visited, we begin to explore all of its unvisited neighbors in turn. The important thing to note is that the moment we find an unvisited neighbor we mark it visited and then immediately move to that node to explore its unvisited neighbors. Eventually when all edges coming out of a node \( v \) have been explored, we return to the node from where we got to \( v \) in the first place (unless \( v \) is the starting node, in which case the procedure terminates).

Consider the graph in Fig. 6. Suppose we start at node A; we first mark it visited. If node A has neighbors B, C, D and E. We move to B, mark it visited, and then from there to C, and mark C visited. Suppose now that C has no unvisited neighbors, then we simply return to where we came from – thus we would find ourselves back at node B, and then since there is an edge from B to D, we will move to D, mark it visited and since it has no unvisited neighbors we move back to where we came from, namely B. Now B has no unvisited neighbors so we go back to where we came from – namely A. Since A has an unvisited neighbor E, we move to E, mark it visited and since E has an unvisited neighbor F, we mark F as visited and move to F. However F has no unvisited neighbors, so we return to E, and since E has no more unvisited neighbors we return to A and the search terminates.

The actual ruby code is shown below and prints the set of nodes that are visited when we first call dfs on some node of the graph.

A graph is said to be connected if there is a route that connects every pair of nodes. So if we call dfs from any node in the graph, all the remaining N-1 nodes are visited. If the graph is not
connected then a dfs search from any vertex cannot visit all the nodes of the graph.

While it looks as if this was perhaps a trivial problem and we did not really do too much besides list the set of nodes that are visited while running the search from an initial starting node, in fact, it turns out that depth first search (dfs) is actually a rather powerful tool in the analysis of graphs and extremely widely used for testing graphs for specific properties, along with computing certain pieces of information that cannot be computed locally.

```ruby
# DFS
N=9
G=Array.new
N.times{|j| G[j]=Array.new  }
G[0]<<1<<2
G[1]<<0<<2<<6
G[2]<<0<<1<<3<<4
G[3]<<2<<4
G[4]<<2<<3<<5
G[5]<<4
G[6]<<1
G[7]<<8
G[8]<<7

#V is the visited array of size N and G is the graph
#You will have to create the representation for a graph G
V=Array.new
N.times{|j|V[j]=false}

def dfs(i)
  V[i]=true
  G[i].length.times{|k|
    if not (V[G[i][k]])
      print G[i][k]
      dfs(G[i][k])
    end
  }
end

if (ARGV.length == 1)
  p = Integer(ARGV[0])
end

print("The nodes reachable from #{p} are ")
dfs(p)
print "\n"

Application: One interesting application is the problem of **indexing** the entire Web. How does Google store all of the web pages in the World Wide Web (WWW)? One way they can do this is by doing what is called a “web crawl”. This web crawl might start from one web page (think
of this as a node in a giant graph), say \texttt{www.umd.edu}, from there they could follow links to other web pages and discover them? Again some mechanism is needed for recording which web pages we have already visited. If you think about this the process is the really the same as that of “searching/exploring” a graph. Except that they have to do this incrementally and if I create a new web page (say one for CMSC 198K) and then add a link to it from my web page, and from \texttt{www.cs.umd.edu}, somehow Google “discovers” this new web page and links to it. That is because they probably repeatedly scan (perhaps once a day, or once a week) the web page \texttt{www.cs.umd.edu} to see what new links have been added.

How would we design an experiment to see how frequently Google scans the web to learn about new web pages?

18 Data Structures for managing data

A stack can be thought of as a pile of books. The last book added is on top of the stack. At every step we may add a new book, or remove a book from the stack. The main point is that this is a LIFO (last in first out) structure. If we add object X, and then object Y. Now if we wish to remove an object, we will first remove Y and then remove X.

Let \texttt{S} denote a stack. The functions below can be used to push a new element \texttt{i} onto the stack, or to pop an element from the stack. The pop function returns the popped element. Notice that we set the top element to nil as well.

\begin{verbatim}
S=Array.new

def push(S,i)
    S << i
end

def pop(S)
    #Assume S is not empty
    i=S[S.length-1]
    S[S.length-1]=nil
    return i
end
\end{verbatim}

For computing shortest paths we use a structure that will allow us to do some specific operations. We call this construct a queue. This is similar to a stack. In a stack we just add and remove objects from one end (the top); here we add things at one end, but we remove them from the other end. A stack is what is called a LIFO structure (last in, first out). A queue in contrast, is a FIFO structure (first in, first out).

The three most useful operations are \texttt{emptyQ}, \texttt{AddQ(i)} and \texttt{RemoveQ}. We will discuss how to implement a queue later on.

19 Shortest Paths (BFS)

The goal is to compute the shortest path to every other node in the graph. We assume that the input is an unweighted graph and we would like to find the shortest route from a node A to node B. In fact, even though we only wish to compute a shortest route to one particular node in the
graph, the algorithm will actually compute the shortest route to all nodes in the graph that can be reached from the starting node.

The intuition is the following: somehow we would like to discover nodes in increasing order of distance. We first discover all nodes adjacent to the root, and after that we discover nodes that are at distance two, and then nodes at distance three and so on. We start our breadth first search by adding the starting node A to the queue. At each step we remove a node from the queue and then immediately mark all of its unvisited neighbors as visited, and add them to the queue. Consider the graph we ran dfs on earlier. There we would mark A as visited, and then add all the nodes B, C, D and E to the queue after marking them as visited. We would then remove node B from the queue. However since both of B’s neighbors have been visited already, we do not add any more nodes to the Queue when we process node B. We next process C and D, and again no new nodes are discovered. Now the queue contains only node E. However when E is removed and processed we add F to the queue, and then finally F is processed.

#V is the visited array and G is the graph of N nodes
V=Array.new; N.times{|j|V[j]=false}

def bfs(i)
  V[i]=true
  AddQ(i)
  while not (emptyQ) do
    v=RemoveQ
    G[v].length.times{|j|
      if not(V[G[v][j]])
        AddQ(G[v][j]); V[G[v][j]] = true
      end
    }
  end
end

if (ARGV.length == 1)
  p = Integer(ARGV[0])
end

print("The nodes reachable from #{p} are ")
bfs(p)
## 20 Shortest Path Routing in Graphs

Many applications need to compute routes to go between nodes in a graph. This occurs when we use online maps to get directions. When we use online maps, we enter a starting and ending address, and ask for a route to go from the start to the destination. The starting and ending points are really nodes in a very large graph. When we get directions, we really get a path in the graph that connects the two nodes. A path is a sequence of nodes connected by edges. Consider the graph shown in Fig. 7. For example, the following is a path – \( r, a, d, e \), since we have a sequence of edges \((r, a), (a, d), (d, e)\). Note that there are many many possible paths between a pair of nodes in a graph. Another path from \( r \) to \( e \) is \( r, b, f, e \). Most map based software packages try to optimize for some quantity. For example, one could try to find the shortest possible route, or perhaps a route that avoids highways, or a route that minimizes travel time (this may be different from taking the route that is shortest in distance due to a variety of factors – traffic lights, different speed limits etc.).

In data networks such as the Internet, data is transferred in packets – each time your web browser downloads a video, or any image, the data is transferred over the network from the server to your computer. Each computer on the Internet has an address. An IP address looks something like 70.108.35.210. (You can use the URL [www.ip-adress.com](http://www.ip-adress.com) to find out the IP address of your computer.) This data may travel large distances over the Internet and may go through a sequence of nodes (called routers) before it gets to your computer. The packets do not travel in an uncontrolled way, otherwise there would be complete chaos! The routers “know” where to forward each packet simply based on the destination address. In this case nodes maintain “lookup tables”. For example node \( A \) may receive a packet on some link that has the destination address \( X \). Where do we send this packet? In the lookup table \( T_A \) (the table for node \( A \)), we will have an array \( \text{addr} \) such that if \( \text{addr}[j] = X \) then \( \text{forward}[j] = B \) which means that a packet meant for node \( X \) should be forwarded on the link to neighbor \( B \). Thus when we get a packet meant for node \( A \), we first check to see if is our address. If so, we do not forward the packet. Otherwise we look up the table to find the index \( j \) such that \( \text{addr}[j] = X \) (recall binary search!). We now forward the packet to the neighbor specified in \( \text{forward}[j] \).

The following is an example of what may be stored in the table for a node. Of course in reality we would not have letters like A, B etc. but IP addresses of nodes.

<table>
<thead>
<tr>
<th>addr</th>
<th>forward</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>X</td>
<td>B</td>
</tr>
<tr>
<td>Y</td>
<td>B</td>
</tr>
<tr>
<td>Z</td>
<td>C</td>
</tr>
</tbody>
</table>

To simplify the problem a little, let us focus on a special case. Let us focus our attention on a particular node, say \( r \) (called the root). Suppose now that each node \( v \) wishes to compute the address of the neighbor to forward the packet to if the destination address is \( r \). In fact, for this case assume that all packets are going to \( r \) – each node simply needs to know where to forward packets meant for node \( r \). If we can do this computation for every possible node \( r \), then we can
build the desired tables. So when we get a packet for any destination node, our table would tell us what to do with the packet. Moreover, this routing scheme will guarantee that at each step when we forward the packet, we are getting closer to \( r \).

We will study an algorithm proposed by Edsger Dijkstra in 1959. The main idea behind the algorithm is to maintain an array called \textit{Visited}, which is initially set to \textit{false} for every node \( i \). In other words we set \textit{Visited}[i] = \textit{false} for each node \( i \). When the algorithm terminates, then \textit{Visited}[i] is \textit{true} for all nodes.

In addition, the algorithm maintains two other arrays – the first array \textit{forward}[i] tells node \( i \) which neighbor to send packets to, that are meant for the root \( r \). The second array \textit{d}[i] records the length of a known path to the root \( r \).

For nodes \( i \) that have \textit{Visited}[i] = \textit{true}, the value of \textit{d}[i] is \textit{exactly} the length of a shortest path to the root \( r \). For the remaining nodes it is clearly an upper bound on the length of a shortest path to \( r \), since we know of a path of length \textit{d}[i] but are not confident that this is indeed the shortest path to \( r \).

When the algorithm starts, we initially set \textit{d}[r] to be 0. For the remaining nodes, we can set the \textit{d}[i] value as \( \infty \). Among all the nodes with \textit{Visited}[i] being false, clearly the node with the minimum \textit{d}[i] value is \( r \) (see Fig. 7). If we scan the edges adjacent to \( r \), then we can see that we have paths to \( a \) and \( b \) of lengths 1 and 5 respectively. Note that these are currently estimates of the shortest path lengths. In other words, we know that there is a path of this length to the node, but we are not confident that these actually denote the shortest path length. In fact, for node \( b \), the shortest path has length 4 by going through \( a \).

However, the algorithm proposed by Dijkstra observes that if we consider the node \( p \) with minimum \textit{d}[j] among all nodes that have \textit{Visited}[j] = \textit{false}, for these nodes we can confirm that the \textit{d} value is indeed correct, or it is the shortest path distance from root \( r \). If so, we can set \textit{Visited}[p] = \textit{true}. We repeat this until all nodes have \textit{Visited}[i] set to \textit{true}. In the previous example, this would allow us to claim that for node \( a \) the distance value is indeed correct and no shorter path to \( a \) can exist.

In addition, after we are sure that we have the shortest path to \( p \), we can consider the edges adjacent to \( p \) in turn. If we have an edge \((p, q)\) with \textit{Visited}[q] = \textit{false} then we check to see if we have discovered a shorter path from \( q \) to \( r \), through \( p \) by checking if \textit{d}[p] + \textit{w}(p, q) < \textit{d}[q]. If this condition is true then we have indeed found a shorter path to \( r \) and we also update \textit{forward}[q] to be \( p \), and set \textit{d}[q] to \textit{d}[p] + \textit{w}(p, q).

The argument as to why this algorithm works is not immediately obvious. The main property

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure7.png}
\caption{Figure showing a small graph.}
\end{figure}
that this proof relies on that may not be immediately obvious is that the $d[i]$ is actually the shortest path from $r$ to $i$, that is restricted to going through nodes that have their Visited value as true. This is a point worth pondering, since it really explains why the algorithm works. For example, when all nodes have Visited set to tt false, then there is only a valid path to $r$. After we process $r$ and set Visited[$r$] to true, then the $d$ values for both $a$ and $b$ are correct since they are set to 1 and 5 respectively, since we are not allowed to go through any other vertices to get to either $a$ or $b$. After we set Visited[$a$] as true we are allowed to cut through node $a$, and thus we set $d$ values for both $d$ and $e$, and also update the $d$ value of $b$.

We illustrate the algorithm by an example using $r$ as the root. In the first column we show the node chosen in each iteration, and the $d$ values of all nodes that have Visited set to false after processing the chosen node. In ( ) we show the neighbor to which packets should be forwarded if the destination is $r$.

We now discuss why the algorithm works. Among all the nodes that have their Visited value as false let $x$ be the node with the minimum $d$ value. We would like to observe that any path that comes to $x$ from $r$ must have length at least $d[x]$. Lets say $P(r, x)$ is a shortest path to $x$ in the graph. If the path cuts through only nodes that have Visited set to true then we have already “captured” this path since the $d[x]$ value denotes the length of the shortest path cutting through nodes that have Visited set to true. Suppose the path $P(x, r)$ cuts through nodes that have Visited set to false. Let $y$ be the first such node as we traverse the path from $r$ to $x$. Since $d[y] \geq d[x]$, it must be that any path $P(r, x)$ that goes through $y$ is at least as long as $d[x]$. 

<table>
<thead>
<tr>
<th>Node</th>
<th>r</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>£</td>
<td>£</td>
<td>£</td>
<td>£</td>
<td>£</td>
<td>£</td>
</tr>
<tr>
<td>r</td>
<td>-</td>
<td>1(r)</td>
<td>5(r)</td>
<td>£</td>
<td>£</td>
<td>£</td>
<td>£</td>
</tr>
<tr>
<td>a</td>
<td>-</td>
<td>-</td>
<td>4(a)</td>
<td>£</td>
<td>£</td>
<td>£</td>
<td>£</td>
</tr>
<tr>
<td>b</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6(b)</td>
<td>4(a)</td>
<td>7(b)</td>
<td>5(b)</td>
</tr>
<tr>
<td>d</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6(b)</td>
<td>-</td>
<td>6(d)</td>
<td>5(b)</td>
</tr>
<tr>
<td>f</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6(b)</td>
<td>-</td>
<td>6(d)</td>
<td>-</td>
</tr>
<tr>
<td>c</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6(d)</td>
<td>-</td>
</tr>
<tr>
<td>e</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 8: The first column shows the nodes chosen at each iteration. For each node we have a column that displays the $d$ (forward) values, until they stop changing.

```ruby
r=Root
N=Numnodes
G=Array.new
N.times{|i| G[i] = Array.new}
# Create graph here
# We also create a weight array called W, so that W[i][j] is the weight of the
# edge connecting node i to node G[i][j]
W=Array.new
def minDvalue(g,d,v)
```
```
min=nil
minvalue= \mbox{$\infty$}
g.length.times{|i|
  if not (v[i]) then
    if d[i]< minvalue then
      minvalue=d[i]; min=i
    end
  end
}
end

Visited =Array.new
G.length.times{|i| Visited[i] = false}
D=Array.new
# Let M be a large number, greater than any possible shortest path length
G.length.times{|i| D[i]= M}

D[r]=0
numVisited =0
forward[r]=nil

while (numVisited < N)
  p = minDvalue(G, D, Visited)
  Visited[p] = true
  numVisited = numVisited+1
  G[p].length.times{|j|
    q = G[p][j]
    If (Visited[q]==false) and (D[q] > D[p]+W[p,j]) then
      D[q] = D[p]+ W(p,j)
      forward[q]=p
  end
}
end
```
In this section we discuss a generalization of the stable marriage problem. Suppose $n$ girls go on a trip together. Since they are on a budget, they would like to share rooms. Suppose we put two girls in a room. Each girl has a preference list ranking the remaining $n-1$ girls. Our goal is to find a pairing of the girls so that each person is paired with one other girl (we will assume for simplicity that $n$ is an even number). If $n$ is odd, the oldest girl gets a single room, and we pair the others. A pairing is called unstable if the following is true: There is a pair of girls, who are not roommates, but both prefer each other to their current roommates. Consider the following example:

Kelly: Rose, Lisa, Pauline.
Lisa: Rose, Pauline, Kelly.
Rose: Kelly, Lisa, Pauline.
Pauline: Lisa, Rose, Kelly.

Suppose we consider the pairing $\{(\text{Kelly},\text{Pauline}), (\text{Lisa},\text{Rose})\}$. Note that this is not a stable pairing, as Kelly prefers Rose to Pauline, and Rose prefers Kelly to her partner Lisa. For example, the pairing $\{(\text{Kelly},\text{Rose}), (\text{Lisa},\text{Pauline})\}$ is on fact stable. Since Rose and Kelly get their first choices, they are happy. Lisa does not get her first choice (Rose) and in fact prefers Rose, but Rose would rather be Kelly’s roommate so there is no instability. Now consider the following (modified) instance:

Kelly: Rose, Lisa, Pauline.
Lisa: Kelly, Rose, Pauline.
Rose: Lisa, Kelly, Pauline.
Pauline: Lisa, Rose, Kelly.

It turns out that for this preference ordering there is no stable solution! To see this consider whoever Pauline is paired with. If Pauline is paired with Lisa, then Lisa prefers Rose to her roommate Pauline, and at the same time Rose prefers Lisa to her partner Kelly. If Pauline is paired with Kelly, then Kelly prefers Lisa to her roommate Pauline, and Lisa prefers Kelly to her roommate Rose. If Pauline is paired with Rose, then Rose prefers Kelly to her roommate Pauline, and Kelly prefers Rose to her roommate Lisa. If we cannot pair Pauline with anyone in a stable way, there can be no stable pairing.

The algorithm is slightly more complex, and works in two stages. In the first stage we simply eliminate potential pairings once we determine that they are not possible pairings in any stable pairing. The first stage is very much like the algorithm discussed previously for the stable marriage problem. The algorithm will again work with girls making “proposals” to other girls on their list. Once $X$ proposes to $Y$, then $Y$ can delete everyone on her list after $X$. So for example if $Y$ prefers $X$ to $Z$, then the pair $(Y,Z)$ is eliminated from consideration, and we can delete $Z$ from $Y$’s list and $Y$ from $Z$’s list. If $Z$ was currently paired with $Y$, then $Z$ becomes free. (We can also describe the stable marriage algorithm in this way.)

In the modified example, suppose Kelly first proposes to Rose that they be roommates. Rose can then remove Pauline from her list (and Pauline removes Rose). The argument is that Rose can never be paired with Pauline in a stable assignment, as she will prefer Kelly, and if Rose is
with Pauline, Kelly is with someone else. Hence the pair (Kelly, Rose) is unstable. Rose proposes to Lisa, her first choice and thus Lisa removes Pauline from her list (and Pauline removes Lisa). Lisa then proposes to Kelly, and Kelly removes Pauline from her list (and Pauline removes Kelly). Note that Pauline’s list is now empty! This means that there is no stable assignment pairing Pauline with any one else.

Let us now consider the original example. Kelly proposes to Rose, and thus Rose can remove both Lisa and Pauline from her list. We also remove Rose from Pauline and Lisa’s lists. Lisa then proposes to her first choice, Pauline (recall that Rose was removed from the list). Pauline gets a proposal from Lisa and removes both Rose and Kelly from her list. Rose proposes to Kelly, and that causes Kelly to remove both Lisa and Pauline from her list. We also remove Kelly from the lists of Lisa and Pauline. The lists now look as follows:

Kelly: Rose.
Lisa: Pauline.
Rose: Kelly.
Pauline: Lisa.

The unique stable solution now is: \{ (Kelly, Rose), (Lisa, Pauline) \}.

However, this is not the complete algorithm since we can only use this method to prune the lists. After pruning the lists, we might still have the situation where some lists have several entries. If any list is empty, we know that no solution exists.
Prime numbers are easy to define. A number is said to be *prime* if it is divisible only by 1 and itself. For example the numbers 2 and 3 are both prime. The number 4 is not prime as 2 divides 4. The numbers 5 and 7 are prime, but 6 is not prime as both 2 and 3 divide 6. Given an arbitrary number \( n \), how can we tell whether or not it is prime. While this task is easy for small numbers, simply check to see if a number \( j < n \) divides it or not; it is actually quite computationally intensive for large numbers. For example, how can we check if 109990034572439 is prime or not? We could check for each number \( 109990034572439 > j > 1 \) whether or not \( j \) divides it. However, this could turn out to be an extremely slow algorithm. Is there no better way to tell whether or not this number is prime?

**Sieve Method:** We first study an old algorithm due to Eratosthenes\(^8\) called the Sieve Method. The algorithm works as follows. Suppose we want to know whether or not \( p \) is prime. First we put the integers 2, 3, \ldots, \( p \) in a list. Let \( q \) be the first prime number 2 (initially). The algorithm will modify the value of \( q \) later. We go down the list and remove all multiples of the prime \( q \). When \( q = 2 \) we remove all the even numbers from the list, namely 4, 6, 8 etc. These number are not prime since they are multiples of \( q \) (in this case multiples of 2). Since 3 has not been removed, it is prime. We now remove 3, and also remove all multiples of 3 that have not been already removed. For example, 6 is not on the list anymore. We thus remove 9, 15, 21 etc. Note that 12, 18 etc have been removed already. 4 has been removed and is not prime. Since 5 has not been removed it is prime. We now remove all the multiples of 5 that are left on the list. These are not prime.

---

\( ^8 \)Eratosthenes was a famous scholar and the director of the famous library of Alexandria. One of his many achievements was to accurately estimate the diameter of the earth.
23 Finding the $k^{th}$ smallest

Given a collection of $n$ numbers, how should we find the smallest number? The general problem can be posed as follows: given a collection of $n$ numbers, how can we find the $k^{th}$ smallest number? For example, when $k = 1$ this corresponds to the above problem of finding the smallest number. When $k = 2$ this corresponds to the problem of finding the second smallest number etc.

One simple algorithm can be used to find the smallest element – we simply scan all the numbers, keeping track of the minimum element “seen so far”. For example, if the collection of numbers was 34, 51, 21, 11, 13, 17. We would consider the first element 34. This would be the minimum element “seen so far”. We would then compare this to the second element 51. Since 34 < 51 we would not change the minimum element seen so far. When we consider 21, since 21 < 34 we would update the minimum element seen so far to 21. When we scan the fourth element 11, we would update the minimum element seen so far once more to 11, since 11 < 21. After that, all the elements we encounter will be greater than 11, so we will not make any changes. The algorithm will then terminate with 11 as the answer. Note that the running time of this algorithm is $O(n)$. The number of updates to the minimum element may be a lot fewer.

How can we find the second smallest element? One method would involve doing a second scan of the set, to find the smallest element after temporarily removing the smallest element. What if we wanted to find the 3$^{rd}$ smallest element, and so on? Would we have to scan the set three times? If we wanted to find the $k^{th}$ smallest element, this technique would involve $k$ scans of the set. It turns out that there are much better ways that can be used to solve the problem, especially if $k$ is large.

One approach can be described as follows. We first pick a partitioning element, say “17” from the above collection. We then compare all the elements to the chosen element to create two sets: one set is called SMALL and the other is called BIG. For the above example, SMALL would contain elements 11 and 13. BIG would contain the elements 34, 51, and 21. SMALL denotes the elements that are smaller than the chosen element, and BIG denotes the elements that are larger than the chosen elements. Now we compare $k$ with the size of the set SMALL. If $k \leq |\text{SMALL}|$ then we run the algorithm again on the set SMALL, looking for the $k^{th}$ smallest element. If $k = |\text{SMALL}| + 1$ then the chosen element is the element we are looking for. If $k > |\text{SMALL}| + 1$ then we run the algorithm on the set BIG, except that now we are looking for the $(k - (|\text{SMALL} + 1))^{th}$ element in the set BIG. We keep running this algorithm until the set of elements becomes very small and then we can solve the problem easily. Let us illustrate this by an example.

If $k = 2$ and we were looking for the second smallest element. For the previous choice of chosen element, $|\text{SMALL}| = 2$. Note that since $k \leq 2$ we can only consider the set SMALL since it must contain the second smallest element. If $k = |\text{SMALL}| + 1$ then we know that the chosen element, namely 17 is the 3$^{rd}$ smallest element. Suppose $k = 4$; now we know that if look in the set BIG, we are actually looking for the smallest element since we have eliminated 3 elements from consideration. Namely, elements 17 and the set SMALL, containing 11 and 13. Since the 4$^{th}$ smallest element is 21, it is actually the smallest element in the set BIG. We now continue looking for the smallest element in the set BIG. This time we choose an element from the set BIG, say “34”. With this choice of element, the sets SMALL and BIG are 21 and 51 respectively. Since $k = 1$ now, we continue looking for the smallest element in SMALL. Since SMALL only has a single element, the process terminates and we stop and we have found the answer, namely 21 which is the 4$^{th}$ smallest element of the original set.
For this algorithm we can actually prove that the total (expected) running time is $O(n)$ for any value of $k$ provided we choose the partitioning element randomly. In other words, we could pick any element from the given set. The choice is made in an impartial manner, and the choice of each element is equally likely, like in a lottery. This kind of choice is actually extremely useful and often yields simple and natural algorithms.
24 Euler Tours

Let’s start with a little children’s puzzle. Can you draw the shown figure (see Fig. 9) without lifting your pen from the paper, and without repeating the same line again? Such a path is called an Euler path.

A few minutes of trying will illustrate that this is not easy at all. In fact, there is no way of doing this unless one uses a trick! (One has to fold the sheet of paper to obtain two disjoint lines, and then one can complete it.)

Now we will try to argue why this is not possible. We can view this drawing as a graph, with nodes (junction points) and edges (lines). Notice that there are four vertices, each having degree exactly five (an odd number). Since we start a line at some node, and end at some node we can only have two nodes with odd degree. All other nodes must have even degree, as we enter the node on some line and then leave on a different line. If we start the drawing in the middle of a line, then we must return to that location and in this case, all nodes must have even degrees. So again, there is no way to make this drawing. If we delete one of the diagonals, then we will have only two odd degree nodes, and such a drawing should be possible (the reader should verify this).

In this section we study a non-trivial application of the Euler tour problem. What is the Euler tour problem? Given a graph $G = (V, E)$ we wish to find a tour in which every edge is traversed exactly once. This problem may be defined for undirected graphs, or directed graphs. If the graph is undirected and connected then an Euler tour exists if and only if every node in the graph has even degree. Notice that here we have to return to the starting vertex, so no odd degree node is possible. If a directed graph is strongly connected, then an Euler tour exists if and only if every node has the same in-degree and out-degree values. A strongly connected graph is one in which there is a directed path between every pair of nodes in each direction.

Imagine the following problem (we will discuss applications shortly). We would like to create a circular sequence of $2^n$ bits (a 0/1 sequence) so that every sequence of $n$ bits occurs as a substring exactly once. Note that there are exactly $2^n$ strings of $n$ bits. Consider the following example with $n = 3$. We would like a circular sequence of 8 bits, so that every possible string of length 3 occurs exactly once. The possible strings of length 3 are: \{000, 001, 010, 011, 100, 101, 110, 111\}. 

![Figure 9: The puzzle.](image-url)
It turns out that there is such a string! Let us try to model this problem as a graph problem. Create a graph, with $2^n$ vertices where each vertex corresponds to an $n$ bit string. In this example, the graph will have $2^n$ vertices. Each node $v$ corresponds to a string $\ell(v)$. We put an edge from $u$ to $v$ if $\ell(u)$ and $\ell(v)$ are compatible. What this means is that these two strings can appear consecutively in the ordering. For example 001 has an edge to 010 and 011 as the last 2 digits of 001 are the same as the first 2 digits of 010, and so they are compatible. In this graph, we wish to find a tour that visits every vertex exactly once. Such a tour would correspond to a valid string of length $2^n$ that contains all possible strings of length $n$. However, this is a very hard problem also known as the Hamilton cycle problem. All known algorithms take time exponential in the size of the graph. An algorithm for Hamilton cycles would be prohibitively slow even for a graph with 1024 nodes (say for $n = 10$).

We now show how to formulate the problem as an Euler tour problem. Rather than modeling the $2^n$ strings as nodes in a graph, we will model them as edges in a graph. We will then ask for a tour that traverses each edge of the graph exactly once. The construction of the graph is less intuitive however. We construct the graph as follows: corresponding to each string of length $n - 1$ we create a vertex. Hence there are $2^{n-1}$ vertices. Suppose we have a string $\ell(u) = x_1x_2\ldots x_{n-1}$ as the string corresponding to vertex $u$. Suppose $v$ and $v'$ are the vertices corresponding to the following strings: $x_2x_3\ldots x_{n-1}0$ and $x_2x_3\ldots x_{n-1}1$. We add labeled edges from $v$ and $v'$ to $u$. The label of edge $(u, v)$ is 0 and the label of $(u, v')$ is 1 (see Fig. 11). Note that each vertex has in-degree and out-degree exactly two, and hence there are $2^n$ edges in this graph. The intuition
is that when we are at a vertex $u$, the string has the characters $x_1 x_2 \ldots x_{n-1}$. We now have a choice of either a 0 or 1. Each choice corresponds to creating a string of length $n$. At this point we are in a new state defined by $x_2 x_3 \ldots x_{n-1} x_n$ where $x_n$ is the choice that was made. Again we have a choice of the next character. Choosing an edge corresponds to making that choice. On this graph we can now run an Euler tour algorithm, which is extremely efficient. In fact, since each vertex has in-degree two and out-degree two, an Euler tour must exist and can be found efficiently.
25 Cooking a Feast

Suppose Martha gets home from work at 5pm, and has guests coming for dinner at 6:30pm. She would also like to take a shower, lay the table, and relax a bit before dinner.

Martha is planning a simple dinner. She would like to make a salad, chicken noodle soup and rice pudding for dessert. Each dish requires some preparation time (chopping, collecting ingredients etc), and then some cooking time. For example, the salad has 25 minutes of preparation time, and no cooking time. The chicken soup requires 10 mins of preparation time, and 40 mins of cooking time on the stove. The rice pudding requires 15 mins of preparation time and 20 mins of cooking time. Martha has no help today, so she can work on preparing one dish at a time. While the dish is cooking, it requires her presence in the kitchen (for occasional stirring), but she can actually prepare another dish while some dishes are cooking on the stove. The question that we are interested in is: in what order should she work on the dishes in order to minimize the time spend in the kitchen?

Suppose Martha first makes the salad, then the chicken soup and then the dessert. Lets suppose she starts at 5pm. By 5:25pm, she has finished the salad. From 5:25pm to 5:35pm she works on preparing the ingredients for the chicken soup (chicken pieces, garlic, chopped carrots, parsley and onions). At 5:35pm she puts the chicken soup to cook on the stove. This will be ready at 6:15pm. She also starts making a pudding for dessert at 5:35pm, and this is ready to cook at 5:50pm. Since the cooking time is 20 mins, it will be ready at 6:10pm. In all she spends 75 mins in the kitchen, from 5pm until 6:15pm when dinner is ready. She is now in panic mode since her guests are about to arrive, and she does not have sufficient time to lay the table, and
take a shower before they get here.

Suppose Martha had done things in a different order. Consider the following order: first work on the chicken soup. This is ready to be put on the gas at 5:10pm, and will be ready by 5:50pm. At 5:10pm, Martha starts working on the pudding. The pudding is ready to go into the oven at 5:25pm, 15 mins later, and will be ready at 5:45pm after 20 mins of cooking time. At 5:25pm she starts preparing the salad and the salad is ready at 5:50pm at the same time as the chicken noodle soup! She has reduced the time she spent in the kitchen significantly! She now has sufficient time to lay the table, take a shower and relax before the guests arrive.

Perhaps because of a lack of understanding of algorithms, people are unable to juggle several different tasks at the same time and choose to order pizza for dinner instead(?). In this example, we are only cooking a small number of dishes, and so perhaps we can pre-compute all possible orders in which Martha could get her cooking done, and see which method is the fastest. Since there are only three dishes, it is easy to try all possible orders. However such an approach (also called "brute force") is not a very efficient algorithm. Suppose we wish to make $N$ dishes. The number of possible orders Martha will have to consider is $N!$ (also called $N$ factorial). In fact, this number is very large. For example, if we are trying to make only 10 dishes, the number of orders is $3,628,800$. This is an extremely large number, and even a computer will take considerable time in enumerating through all these options. If we would like to make 15 dishes\footnote{This is pretty common in Indian weddings.}, the number is even larger, namely $1307674368000$.

A simple algorithm that solves this problem optimally is as follows. First we introduce some notation. Let $P_i$ and $C_i$ refer to the processing times and cooking times for the $i^{th}$ dish. For example, suppose the dishes are Chicken, Dessert and Salad. Suppose $P_c = 10$ and $C_c = 40$ are the processing and cooking times for the chicken noodle soup. Note that for the dessert $P_d = 15$ and $C_d = 20$. For the salad, $P_s = 25$ and $C_s = 0$.

Order all the dishes in decreasing order of cooking time and prepare the dishes in this order. This algorithm is simple, and moreover produces an optimal solution. In other words, this means that if we cook dishes in decreasing order of cooking time, we will obtain the solution that minimizes the amount of time Martha spends in the kitchen. For the above example, since the cooking time of the chicken noodle soup is the highest, we will cook that dish first, and then work on the dessert and finally work on the salad.

We now proceed to prove that this solution is better than any other solution. Suppose we number the dishes in the order in which they were processed. The time at which the $i^{th}$ dish is completed is defined as follows: $P_1 + \ldots + P_i + C_i$, since we process the $i−1$ dishes before the $i^{th}$ dish and only start processing the $i^{th}$ dish at time $P_1 + \ldots + P_{i−1}$ . We do assume that all these dishes can cook at the same time on different burners. The time spent in the kitchen is the maximum completion time over all the dishes we are preparing. For example, the completion time of the chicken soup is 50 mins, the completion time of the dessert is 45 mins, since its preparation time and cooking times are 15 and 20 mins respectively, but we only start it after the chicken soup is ready to start cooking. The completion time of the pudding is 50 mins, since we only start it after 25 mins, and it takes 25 mins of preparation time. Martha should feel happy that she not only got her dinner ready quickly, but can brag to her friend Paula about the optimality of her solution!

However, Paula is not convinced that Martha used an optimal algorithm. Paula claims that she has another order of preparing the dishes that takes less time. We are now going to consider
Figure 13: Cooking a feast.
the general problem with $n$ dishes. Suppose the last dish to complete was the $k^{\text{th}}$ dish (it does not matter for the proof if there are several dishes that completed last, as in our simple example, both the chicken and salad were the last to complete). Let $S$ be the set of the first $k$ dishes scheduled by Martha. Suppose $P(S)$ is the sum of the processing times of this set of dishes. Let the completion time of the $k^{\text{th}}$ dish be $T = P(S) + C_k$. Paula also cooks the same set of dishes, but perhaps in a different order. We ask: among the dishes in set $S$, which is the last one cooked by Paula? Suppose that is dish $j \in S$. This last dish that Paula, will have a completion time of at least $T$ since it will have completion time of at least $P(S) + C_j$. However since $C_k$ is the dish with the least cooking time in set $S$, we must have $C_j \geq C_k$. Hence $P(S) + C_j \geq T$. Hence Paula spends at least as long as Martha in the kitchen!

The problem becomes significantly more complicated when we have a small number of burners. Consider the following situation. Suppose we would like to make chicken and pasta, with $P_{c} = 30$ and $C_{c} = 30$. Suppose $P_{p} = 5$ and $C_{p} = 25$. With two burners, we will first work on the chicken, and then start cooking it after 30 mins. The chicken is ready in an hour. We start the pasta after 30 mins, and it is ready when the chicken is ready. Suppose now that we have only one burner. If we do things in this order, it will take 85 mins and dinner is ready only at 6:25pm, since we will only start boiling the pasta after the chicken is cooked. We should first work on the pasta and then put it to boil. If we start at 5pm, the pasta is ready at 5:30pm and the chicken goes in at 5:35pm and is ready at 6:05pm. With one burner it only takes 5 mins longer than it did with two burners if we change the order of preparing the dishes. This shows that the algorithm we developed earlier does not give the best solution with a single burner. The general problem with a small set of burners is considerably more complex\footnote{This might explain why people are buying 6 and 8 burner stoves. However, with a good algorithm a 4 burner stove may suffice.}. Can we develop a general method that will deliver the best solution quickly? Or a method that will develop a solution that is “close” to the best possible.
26 Binary Arithmetic

We first discuss representation of numbers – the numbers we are used to are numbers such as 137, or 2358. The first number is really $1 \times 100 + 3 \times 10 + 7 \times 1$. The second number can be written as $2 \times 1000 + 3 \times 100 + 5 \times 10 + 8 \times 1$. We can also write this as $2 \times 10^3 + 3 \times 10^2 + 5 \times 10^1 + 8 \times 10^0$. These numbers are represented using 10 digits - 0,1,2,3,4,5,6,7,8,9.

In contrast, computers represent numbers using base 2 and as a result use only two digits – 0 and 1. All numbers can be represented using only these two digits. Each digit is referred to as a bit, or binary digit.

The number 1011 (in binary) is really $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$ or the number $8 + 2 + 1 = 11$ (in decimal).

This is how we represent 16 numbers using 4 bits.

<table>
<thead>
<tr>
<th>binary code</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
</tr>
</tbody>
</table>

Using the fingers on one hand you should be able to represent all numbers between 0 and 31 since you can really represent a bit with each finger.

27 Communicating over a Noisy Channel

The world has changed rapidly with the availability of high bandwidth fiber-optic connections linking different parts of the globe. Today we can use a cell phone to send an SMS from a phone in a small village in Africa to a cell phone user who may be sitting in Kathmandu. We are able to use communication networks to send messages that travel rapidly over a variety of networks (perhaps even using satellites) and get delivered very quickly. Cell phones communicate with a cell tower using a wireless transmitter, and such transmissions are rather “noisy” - what this means is that the bit transmitted over the channel often gets modified in transmission. All messages are represented using binary digits. I might transmit a “0”, but you might receive a “1”. You would never know that an error occurred during the transmission! Since each character has a 7 digit binary code changing even a single digit can change the interpretation of the character significantly. How do we deal with transmission errors? We will first examine a method that does not correct errors, but is able to detect an error when it occurs. So for example, when errors
are relatively infrequent, then this method works well. If an error in transmission is detected, then we can ask the sender to re-transmit the information. Another method could be to ask for every message to be sent twice, if only both the messages match then we know that there is a relatively low chance that an error occurred. In any case we will deal with single bit errors only. More complex schemes are used in practice to decode errors when channels are noisy and many bits can get corrupted.

Simple examples abound – suppose I say “I would like to take a bath” and you hear “I would like to take a path”, you would think I was rather odd. If you heard “I would like to take a math” then you would likely realise that you did not hear correctly. Ideally if we had to redesign the English language with error correction in mind, what we would like to only have those words in the language so that there was no way of converting one word to another unless we changed at least two letters. This would ensure that if at most one letter in a word got modified the message could not be corrupted without us knowing that it did. Unfortunately english is not a language designed with this in mind.

Computers encode all information using a binary code. Everything is represented using a sequence of binary digits, called bits. The ASCII character set encodes characters using a 7 digit binary code. For example the letter “d” is represented as “1100100”, or the number one hundred written in binary. A special digit is appended as a prefix on this code. The general rule for the special digit would be to set it to either 1 or 0, ensuring that the total number of 1’s in the 8 bit code (a byte) is even. So if we had to transmit a “d” we would transmit “11100100” (the special digit is shown in italics). Note that if any digit is changed during transmission (say 1 → 0 or 0 → 1) then this will be easily detected. For example if we get “10100100” then since there is an odd number of 1’s we know that an error occurred during transmission, without knowing which bit was corrupted. Now suppose that the special bit itself gets corrupted and we receive a “01100100”, in this case we again declare error since the number of 1’s is odd. However, there was no corruption in the data itself. Even if a 0 gets changed to a 1, we will detect the error. However, if two bits get corrupted then the error cannot be detected. For example we might receive “00100100” (here both the special bit and one of the data bits got changed), and will not detect this error.

We next discuss one scheme that allows error correction to take place. In other words if there is a bit that gets corrupted using transmission we will be able to figure out which bit got corrupted. This works only when there was an error in transmitting some data bit, but not when the errors strike the checking bits.

Suppose we have a message of 7 bits \( (b_7b_6b_5b_4b_3b_2b_1) \); this encodes a single character (which could be a letter of the alphabet, or just a character such as “:”). We use three extra bits as the “checking” bits. We call these bits \( x_1x_2x_3 \). In general each \( x_i \) bit does a parity check for a subset \( S_i \) of the input data bits.

The bit \( x_1 \) does a parity check for the four bits \( b_7b_6b_5b_4 \). In other words we make sure that the total number of 1’s is even, in the five bits \( x_1b_7b_6b_5b_4 \). Recall that the 7 digit code for the character “d” is “1100100”. Here \( b_7b_6b_5b_4 = 1100 \); thus we set \( x_1 \) to 0 (giving us a total of 2 1’s). Now suppose there is an error in the transmission for \( b_5 \) and instead of 1 we receive 0. In this case we receive “1000100”. By looking at \( x_1 \) and noticing that the number of 1’s in \( x_1b_7b_6b_5b_4 \) is 1, we realise that there was an error in the transmission. However currently we still are not sure which bit was transmitted incorrectly among the four bits \( b_7b_6b_5b_4 \).

The bit \( x_2 \) does a parity check for the four bits \( b_7b_6b_3b_2 \). In other words we make sure that the total number of 1’s is even in the five bits \( x_2b_7b_6b_3b_2 \). Since the code for the character “d” is
“1100100”. Here $b_7b_6b_3b_2 = 1110$; thus we set $x_2$ to 1 (giving us a total of 4 1’s). Now suppose there is an error in the transmission for $b_6$ and instead of 1 we receive 0. In this case we receive “1000100”. By looking at $x_2$ and noticing that the number of 1’s in $x_2b_7b_6b_3b_2$ is 3, we realise that there was an error in the transmission. Now we know that the error was in one of the four bits $b_7b_6b_3b_2$. Looking at the information we had earlier, its evident that the error was in either bit $b_7$ or $b_6$.

The bit $x_3$ does a parity check for the four bits $b_7b_5b_3b_1$. We make sure that the total number of 1’s is even in the five bits $x_3b_7b_5b_3b_1$. Since the code for the character “d” is “1100100”. Here $b_7b_5b_3b_1 = 1010$; thus we set $x_3$ to 0 (giving us a total of 2 1’s). Now suppose there is an error in the transmission for $b_6$ and instead of 1 we receive 0. In this case we receive “1000100”. By looking at $x_3$ and noticing that the number of 1’s in $x_3b_7b_5b_3b_1$ is 2, we realise that there was no error in the transmission for these bits. Now we know that the error was in the bit $b_6$ since it could not have been in the bit $b_7$.

More generally, by considering the 3 checking bits $x_1x_2x_3$ we can decode the error immediately using the bit pattern. Define $x'_i = 1$ if and only if $x_i$ indicates an error in the parity check for $S_i$. The bit pattern $x'_1x'_2x'_3$ will tell us which bit in the input data is corrupted. If there is no data transmission error then $x'_i = 0$ for all $i$. When $b_6$ was the corrupted bit, then $x'_1 = 1$ and $x'_2 = 1$ but $x'_3 = 0$. This is the sequence 110 - this is the bit representation for 6, and tells us that the error is in $b_6$!
We shall now examine the question of how a modern electronic computer actually computes functions. Let us consider a simple problem that involves adding two numbers. In ruby, this is easily accomplished by simple saying “x=a+b” and this will store in a memory location named “x” the sum of the values in memory locations “a” and “b”.

However, a computer needs to perform this operation “somehow”. Ruby is a “high level language” and lets us write programs in a way that makes it easy to talk about memory locations by name and also specify specific operations on these values by simply using the operator “+”. This is not how computer hardware works however. Ruby is eventually translated to some lower level machine code that the computer can easily interpret and execute.

However, this still begs the question - how does a computer actually add numbers? This is also complex - but the objective is to examine some basic principles and to show you a glimpse of what is going on inside the machine.

We talked about binary representation earlier as well - a number such as 7283 is really $7 \times 10^3 + 2 \times 10^2 + 8 \times 10^1 + 3 \times 10^0$. However, we need to use 10 types of digits when we work with base 10 numbers – 0,1,2,3,4,5,6,7,8,9. When working with binary arithmetic we have only 0 and 1 (these are our binary digits, or bits) and all information and values are represented using bits. The primary reason is that electrical voltages are used to represent 1 and 0, and so each wire has two possible states. A 2V line may represent 1 and a very low voltage may be used to represent 0.

Suppose we have the numbers 5 and 4, these are represented as 0101 and 0100 (why is this the correct binary representation for these two numbers?). Adding them gives 9, or 1001.

1
0101
0100
----
1001

How shall a computer actually perform this operation? This operation involves lining up the two bit sequences and then performing the addition operation, generating a “carry bit” every time we need to. The carry bit is then fed to the pair of bits to the left as we add from right to left.

To understand how to implement addition of two bit sequences we will first have to take a short detour into some very basic Boolean Algebra. This is a way to define operators that work with boolean variables - these are variables that are used to generally denote truth values of statements and can be either TRUE or FALSE. We represent TRUE as the bit 1 and FALSE as the bit 0.

Transistors were used to create logical “gates” – these are basic circuit elements that are used to define boolean functions and operators. We will use these logical gates as basic circuit elements to create a circuit that will perform the operation of adding.

The first operator we discuss is the AND gate. Its takes two inputs, X and Y, and outputs 1 if and only if X=1 and Y=1. We can write this as $X \land Y = X.Y$.

The second operator we discuss is the OR gate. This also takes two inputs X and Y, and outputs 0 if and only if X=0 and Y=0. We write this as $X \lor Y = X + Y$.

The last operator is the logical NOT gate that is used to flip a bit. Thus $\overline{X}$ is 1 if and only if X is 0.
The table that shows the precise behaviour of both the AND and OR functions are shown below.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X.Y</th>
<th>X+Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

We first consider a 2 bit adder (called a HALF ADDER). Suppose we wish to add two 1 bit numbers X and Y. The output is two bits – an S bit (sum) and a C bit (carry).

The logical functionality of S and C are obvious.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The carry bit is very easy to implement since \( C = X.Y \) since we generate a carry only when both X and Y are 1. We now need to figure out a boolean expression for the S (sum) bit – note that the S bit is 1 under precisely two conditions – when X is 0 and when Y is 1, and when X is 1 and Y is 0. Thus we can set \( S = \overline{X}.Y + X.\overline{Y} \). (You should verify this.) We could try to set \( S = X + Y \) but this is not correct, since S is supposed to be 0 when X and Y are both 1 (but the way the + operator is defined, 1+1 yields 1).

The logic to build a HALF adder is thus very easy. We can do this with some wires, two NOT gates, three AND gates and an OR gate (see Fig. 14).

However, to actually build something that will add say two 4 bit numbers, we need something slightly more sophisticated due to the carry bit. Suppose we wish to add \( X_1X_2X_3X_4 \) and \( Y_1Y_2Y_3Y_4 \), then for example while adding \( X_3 \) and \( Y_3 \) we also need to add a carry bit in case both \( X_4 \) and \( Y_4 \) are 1 since in this case they will generate a carry. This is accomplished by building something called a FULL ADDER. This takes as input three lines – X, Y and \( C_i \) (carry-in) and generates both an S bit and a \( C_o \) bit (carry-out), that in turn can be fed into the carry in line for the pair of bits to the left etc.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>C_i</th>
<th>S</th>
<th>C_o</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

We can construct this in logic by defining S and \( C_o \) as follows.

\[
C_o = X.Y + X.C_i + Y.C_i
\]

The logic behind this is that the carry bit is 1 if and only if at least two of the input bits are 1. We can also derive this and simplify as follows. Note: we are taking the conditions when the \( C_o \) bit is 1, and setting an OR of all these conditions.

\[
C_o = X.Y.C_i + X.Y.\overline{C_i} + X.\overline{Y}.C_i + \overline{X}.Y.C_i
\]
Figure 14: Figure showing ADDER circuits.

\[ C_o = X.Y.(C_i + \overline{C_i}) + X.Y.C_i + \overline{X}.Y.C_i \]
\[ C_o = X.Y + X.C_i + Y.C_i \]
\[ S = \overline{X}.Y.C_i + \overline{X}.Y.C_i + X.Y.\overline{C}_i + X.Y.C_i. \]

However, we can also use two HALF-ADDERs to build a full adder as follows. The idea is very simple – we first add the bits \( X \) and \( Y \), we feed the carry-out line of this into a second HALF-ADDER that adds the original carry bit. With a small amount of effort you can verify that this circuit has the desired behavior by trying each 3 bit sequence for \( X,Y,C \).

Finally, a four FULL-ADDERs can be lined up together to create a circuit that will add two four bit numbers (you basically have to feed the carry out line to the next FULL-ADDERs carry in line.

29 Conclusion

This document tries to show that several interesting results can be motivated, explained and taught to non-specialists, and these will give people a glimpse into the mind of a computer scientist. Computer Scientists have to be able to think “big” (modeling, conceptualizing and imagining useful software tools) as well as “small” (getting everything to work correctly when put together). An individual can have a huge impact. In what other field can one person, or a small group of people, literally create a billion dollar industry in their basement?