Advanced Algorithms I

Todd Davies

December 29, 2015

Introduction

This course unit has two objectives. The first is to introduce the student to a range of advanced algorithms for difficult computational problems, including matching, flow networks and linear programming. The second objective is to outline the mathematical techniques required to analyse the complexity of computational tasks in general. There are two pieces of assessed coursework, and an exam at the end.

Aims

This unit provides an advanced course in algorithms, assuming the student already knows algorithms for common computational tasks, and can reason about the correctness of algorithms and understand the basics of computing the complexity of algorithms and comparing algorithmic performance.

The course focuses on the range of algorithms available for computational tasks, considering the fundamental division of tractable tasks, with linear or polynomial-time algorithms, and tasks that appear to be intractable, in that the only algorithms available are exponential-time in the worst case.

To examine the range of algorithmic behaviour and this fundamental divide, two topics are covered:

- Examining a range of common computational tasks and algorithms available; We shall consider linear and polynomial-time algorithms for string matching tasks and problems that may be interpreted in terms of graphs. For the latter we shall consider the divide between tractable and intractable tasks, showing that it is difficult to determine what range of algorithms is available for any given task.
- Complexity measures and complexity classes: How to compute complexity measures of algorithms, and comparing tasks according to their complexity. Complexity classes of computational tasks, reduction techniques. Deterministic and non-deterministic computation. Polynomial-time classes and non- deterministic polynomial-time classes. Completeness and hardness. The fundamental classes P and NP-complete. NP-complete tasks.

Additional reading

Title	Author	Year
Core Algorithm design: foundations, analysis and internet examples	Goodrich, Michael T. and Roberto Tamassia	2002
Introduction to the theory of computation (3rd edition)	Sipser, Michael	2013

Contents

1 Algorithmic Wisdom **3**
- 1.1 Different types of algorithms 3
- 1.2 Computability 3
- 1.3 Asymptotics and optimisation 4

2 Graphs **4**
- 2.1 Connectivity 5
- 2.2 Representing graphs 5
- 2.3 Classifying Edges 6
- 2.4 Graph Algorithms 6
  - 2.4.1 Depth first search 6
  - 2.4.2 Dijkstra's algorithm 7
  - 2.4.3 Tarjan's Algorithm 7

3 Linear Programming **8**
- 3.1 Background mathematics 8
- 3.2 The maximum problem 10
- 3.3 The minimum problem 11
- 3.4 The Simplex Method 11

4 Turing machines and complexity measures **11**
- 4.1 Formally defining a Turing Machine . . . 12
  - 4.1.1 Acceptance, computability and decidability 12
  - 4.1.2 The Universal Machine 13
  - 4.1.3 Nailing that TM definition 13
  - 4.1.4 Enumerators 13
- 4.2 The Halting problem 14

Scooping the Loop Snooper
An elementary proof of the undecidability of the halting problem **15**
- 4.3 Complexity 15

1 Algorithmic Wisdom

Lectures 2 & 3

The first two lectures of the course describe different types of algorithms, what is computable, where to optimise algorithms, asymptotics and heuristics.

1.1 Different types of algorithms

There are three types of algorithms that are mentioned:

You should know all of this from COMP26120.

Divide and conquer

These algorithms continually break a problem down into smaller parts, which are easier to solve, until eventually, the problems are trivial and easily solved. This is often used when the data you're operating on is in a recursive datastructure such as a tree. If you're writing an algorithm to find how many nodes there are in a tree, then you could use divide and conquer:

```
int countTreeSize(tree) {
int size = 1;
if (tree.left) size += countTreeSize(tree.left);
if (tree.right) size += countTreeSize(tree.right);
return size;
}
```

As you can see from the example, divide and conquer algorithms are usually recursive.

The divide and conquer technique can be applied to graphs, but in order to do this, you must keep track of which nodes you've visited with a flag on each node. If we wanted to count the nodes in a graph, we could do:

```
int countGraphSize(graph) {
if (graph.visited) return 0;
graph.visited = true;
int size = 1;
for (child in graph) {
  size += countGraphSize(child);
}
return size;
}
```

Mutual Recursion

Mutual recursion describes an algorithm that operates on data where one type of data can reference another, and the other can reference it. The example given is that of statements and expressions in programming languages; statements contain expressions, and expressions can also contain statements. Parsing such a structure might involve two algorithms that recurse on each other!

Dynamic Programming

Dynamic programming exploits the fact that when some problems are broken down into smaller sub-problems, some of the sub-problems are identical. Dynamic programming algorithms start from the very smallest sub-problems and build up to the final solution, and usually cache results to sub-problems in a table so that work is not done twice.

1.2 Computability

There are many different definitions of computability, including lambda calculus, Turing machines, rewriting rules, random access machines and (many) more. The idea that relates all of these

things, is that they all have the same capabilities. That is to say that if you can compute something using one of these ideas, then you can also compute it on all the others too.

There are also a class of 'alternate' computing mechanisms, such as quantum computers and neural computers. These ideas have the potential to compute things that a Turing machine (or its equivalents cannot), but they are significantly harder to build, and functional implementations do not exist yet.

1.3 Asymptotics and optimisation

When you have to get a computer to perform a task, implementing a simple algorithm first is a good idea, since you will at least have something to demonstrate to people, and you will gain a good understanding of the problem at hand. However, simple algorithms are often slow; how should we evolve our implementation to be as fast as we need it to be?

Profiling can tell you where your code is spending most of its time. Sometimes your algorithm will be really fast, and the processor will spend most of its time waiting for IO to give it more data; this is often the case with GPU computation.

Assuming you find some CPU bottleneck in your code, before you spend hours making it faster, consider whether it is worth the effort. If this part takes up 10% of your runtime, and you make it twice as fast, your program will only run 5% faster. This is an example of the Law of Diminishing Returns.

As well as optimising specific parts of an algorithm, you also should consider its asymptotic run time. An algorithm that runs in $O(n^2)$ time is probably going to be better than one that runs in $O(nlog(n))$ time. However, this isn't always the case; some algorithms (often ones with good asymptotic run times) take a long time to set up, usually when you have to transform the data into some different datastructure. If you are running your algorithm on a small amount of data, then an algorithm that you can run on your data *as is* might outperform a fancier algorithm that you have to invest more overhead in.

Sometimes a good solution is to use different algorithms depending on the input. If there are only a few cases that produce worst-case performance, you could even hard-code solutions to those!

The average case runtime of a algorithm is also important. Haskell uses a type checker that runs in $O(2^{n^n})$ time in the worst case, but for every program that isn't made specifically to mess with the compiler, runs in linear time.

2 Graphs

A graph is a pair $G = (V, E)$ where V is a finite set, and E is a set of pairs between items in V. Elements in V are *vertices* or *nodes*, and elements in E are *edges*.

Different mathematicians have different rules about what exactly can go in a graph. For the purposes of this course, the graphs shown in Figure 1 aren't allowed.

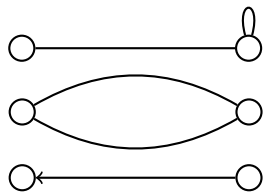

Figure 1: Self loops, duplicate edges and arrows are not allowed.

A **directed graph** is just like a normal graph, except the edges do have arrows. The only mathematical difference is that the set E is a set of ordered pairs.

The *degree* of a node in a graph, is the number of edges that are adjacent to (touching) it. If the graph is directed, then it is the number of edges originating from the node.

A weighted graph is one where each edge is associated with a value representing its weight. The length of a path between nodes is simply the sum of the edge weights connecting the nodes.

2.1 Connectivity

A node a is **reachable** from another node b if there is some sequence of nodes connected by edges that go from the a to b.

A graph where every node is reachable from every other node is **connected**. A strongly connected graph is a **directed graph** where each node is reachable from each other node.

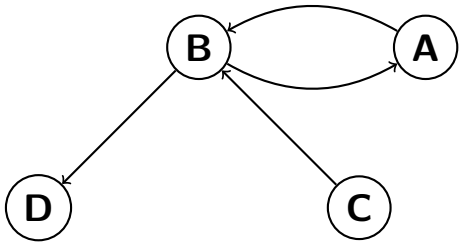

Figure 2: This graph is not strongly connected, but the subgraph with only nodes A and B is. If we were to remove the arrows, then the graph would be connected.

2.2 Representing graphs

We can store graphs in computer memory in two ways:

Adjacency List

There are two different kinds of adjacency list, the first is like so:

```
public class Graph {
  List<Vertex> nodes;
  List<Edge> edges;
}
public class Edge { Vertex a, b; }
public class Vertex { List<Edge> outlist; }
```

Here, we keep a list of all the nodes, and a list of all the edges. From any edge, we can see what nodes it connects, and from any node, we can see what edges it connects.

The other type of adjacency list is a bit simpler, but less efficient in some cases:

```
public class Graph<T> {
  Set<T, List<T>> adjList;
}
```

Adjacency Matrix

Here, a matrix indicates whether there is an edge between two nodes:

	A	B	C	D
A	0	1	0	0
B	1	0	0	1
C	0	1	0	0
D	0	0	0	0

This adjacency matrix represents the graph in Figure 3. You can see that it is fairly wasteful in terms of memory $O(|E| * |V|)$, though with bit arrays, it has a very low constant overhead.

2.3 Classifying Edges

During a *depth first traversal* (Section 2.4.1) of a graph, we can classify each edge into one of four types. When doing the traversal, we process each edge from left to right (from the viewer's perspective), and we define an edge to be an *ancestor* of another edge if there is a path from the ancestor edge to the descendent edge and the ancestor is traversed first. The four types of edges are:

Tree edges
: These edges lead to a new node during a search. If you remove all the edges from the graph except tree edges, then you get a tree!

Forward edges
: These go from ancestor edges to descendent edges, but are not tree edges (i.e. the descendent node has been visited already).

Cross edges
: These go between edges where no node is the ancestor of the other.

Back edges
: An edge that goes from a descendant to an ancestor.

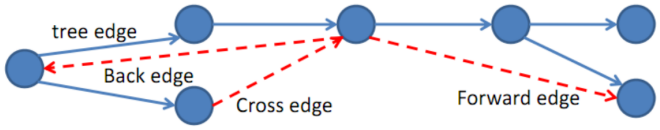

Figure 3: A pictorial illustration of the different edge types.

2.4 Graph Algorithms

You will have covered some of the algorithms featuring here in previous courses, or perhaps seen them in the wild. They are however interesting and useful, so it's worth a recap even if they're not new! I've ordered these in roughly increasing order of mental strain.

2.4.1 Depth first search

Depth first search (DFS) is an algorithm to find a node in a graph starting from another node. It works on both directed and undirected graphs, and runs in $O(|V| + |E|)$ (linear) time. The psudo code looks like this:

Note how we mark nodes as having been visited (by adding them to `visitedNodes`). This is so if there is a loop in the graph, the algorithm doesn't run indefinitely.

```
Node dfs(Node haystack, Node needle) {
  Stack<Node> toVisit = new Stack<>();
  Set<Node> visitedNodes = new Set<>();
  toVisit.push(haystack);
  while(!toVisit.isEmpty()) {
    Node node = toVisit.pop();
    if (visitedNodes.contains(node)) continue;
    visitedNodes.add(node);
    if(node.equals(needle)) {
      return needle;
    } else {
      for(Node child : node.children) {
        toVisit.push(child);
      }
    }
  }
```

```
    return null;
  }
```

A Breadth First Search is the same, except you use a `Queue` instead of a `Stack`.

To see if a graph is connected, do a depth first search as in the example code, but don't stop when you find a needle, only stop when the `toVisit` stack is empty. If `visitedNodes` contains all of the nodes in the graph, then the graph is connected.

Depth first search also lets you find if one node is reachable from another in linear time, and also if a graph is connected in linear time too, with a few modifications.

We can also find out if a directed graph is strongly connected in $O(|V| + |E|)$ time using Tarjan's algorithm, which we'll see later.

2.4.2 Dijkstra's algorithm

Dijkstra's algorithm finds the undirected shortest path between two nodes in a graph. Here is the psudo-code:

My best advice for learning an algorithm like this, is to get a whiteboard, draw out the problem, and then run the solution manually on the whiteboard. Then you have a pictorial and visual explanation of how the algorithm *really* works.

```
function Dijkstra(Graph, source):
  create vertex set Q

  // Initialization
  for each vertex v in Graph:
    dist[v] = INFINITY
    prev[v] = UNDEFINED
    add v to Q

  // Distance from source to source
  dist[source] = 0

  while Q is not empty:
    u = vertex in Q with min dist[u]
    remove u from Q

  for each neighbour v of u:
    alt = dist[u] + length(u, v)
    // A shorter path to v has been found
    if alt < dist[v]:
      dist[v] = alt
      prev[v] = u

  return dist[], prev[]
```

Listing 1: Dijkstra's algorithm (from Wikipedia)

If we use a Fibonacci heap, for the priority queue, then the runtime of Dijkstra's algorithm is $O(|E| + |V|log(|V|)$. If we use a normal heap, then the runtime is $O(|E|log(|V|))$.

2.4.3 Tarjan's Algorithm

Before we delve into Tarjan's Algorithm, we need to discuss strongly connected components. You will remember from Section 2.1, that a graph is strongly connected if all nodes are reachable from any other node.

A *strongly connected component* is a subset of edges within a graph, where the subset is strongly connected. If a graph has only one strongly connected component, then it is strongly connected.

The psudo code here is from Wikipedia. If you're reading this after week 8 of the `COMP36111` course, then you probably have your own implementation in C.

```
algorithm tarjan is
input: graph G = (V, E)
output: set of strongly connected components (sets of vertices)

index := 0
S := empty
for each v in V do
  if (v.index is undefined) then
    strongconnect(v)
  end if
end for

function strongconnect(v)
  // Set the depth index for v to the smallest unused index
  v.index := index
  v.lowlink := index
  index := index + 1
  S.push(v)
  v.onStack := true

  // Consider successors of v
  for each (v, w) in E do
    if (w.index is undefined) then
      // Successor w has not yet been visited; recurse on it
      strongconnect(w)
      v.lowlink  := min(v.lowlink, w.lowlink)
    else if (w.onStack) then
      // Successor w is in stack S and hence in the current SCC
      v.lowlink  := min(v.lowlink, w.index)
    end if
  end for

  // If v is a root node, pop the stack and generate an SCC
  if (v.lowlink = v.index) then
    start a new strongly connected component
    repeat
      w := S.pop()
      w.onStack := false
      add w to current strongly connected component
    while (w != v)
    output the current strongly connected component
  end if
end function
```

3 Linear Programming

Linear programming is an optimisation problem, where we want to find the 'best' solution to a set of equations. We're going to solve it using the *simplex* method, but before we do, I think it's a good idea to recap some high-school mathematics first. Feel free to skip the next subsection if you're feeling confident with it.

3.1 Background mathematics

An inequality relation is just like a normal equation, except the equals sign is replaced by either $<$, $>$, $\leq$ or $\geq$. When you solve an inequality, you generally want to get all of the similar terms

on one side (e.g. move all the variable terms over to one side, and all of the constants to another side). This *mostly* works like a normal equation; you can add and subtract from both sides just like normal, however, if you want to divide or multiply **by a negative quantity**, then you need to **reverse the equality**.

$$\begin{aligned} -2x &> -2 \\ 2x &< 2 \\ x &< 1 \end{aligned}$$

This equality is satisfied whenever x is less than 1. If we have two terms in the equality (something similar to $ax + by \geq c$), it becomes slightly harder to solve. To solve this we can:

- Plot a graph of the line $ax + by = c$.
- Pick a test point (x, y) not on the line, and plot it on the graph.
- If the point (x, y) satisfies the inequality, then shade the opposite side of the line to which the point is on, otherwise, shade the same side.

For example, given $3x + 4y \leq 6$, and choosing the point $(-2, 1)$ we end up with what's in Figure 4.

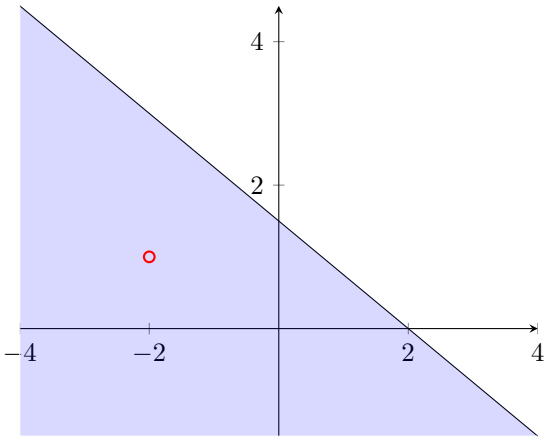

Figure 4: A graph of $3x + 4y \leq 6$, with all the valid values shaded blue.

If we have multiple inequalities, we want to *find the region where all of them are satisfied*. This involves plotting each line, and shading the regions where they're not satisfied, which means the blank bit is the bit we want.

If we have the equations $3x + 4y \leq 6$, $2y - x \leq 2$ and $x \geq 0$, we will get something like in Figure 5.

The corner points are especially important to us, since that's often where the useful numbers are (we'll see more of this later). In order to find the corner points, we solve the two lines that form them. Solving the following equations gives the points in Figure 6.

- $x = 0, 3x + 4y = 6$
- $x = 0, 2y - x = 2$
- $2y - x = 2, 3x + 4y = 6$

So, now we know how to solve these types of problems, lets look at how to decompose a problem statement into a set of equations.

> You can buy wood in 11 meter lengths, or 6 meter lengths. A 11 meter piece of wood can be cut into two lengths of 5 meters, and one length of 1 meter. A 6 meter piece of wood is cut into one length of 5 meter, and one length of 1 meter.

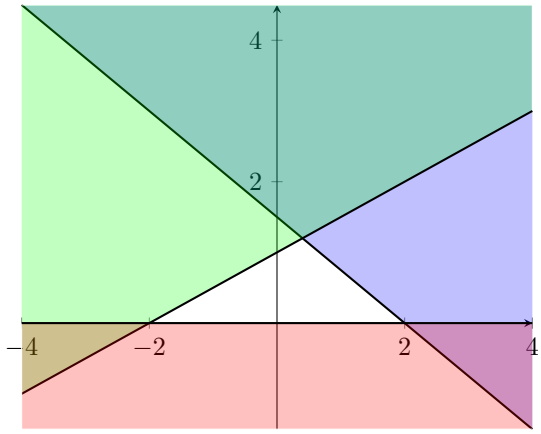

Figure 5: A graph plotting $3x+4y \leq 6$, $2y - x \leq 2$ and $x \geq 0$, where the points not satisfying the inequalities are shaded in blue, green and red respectively. The clear region satisfies all points.

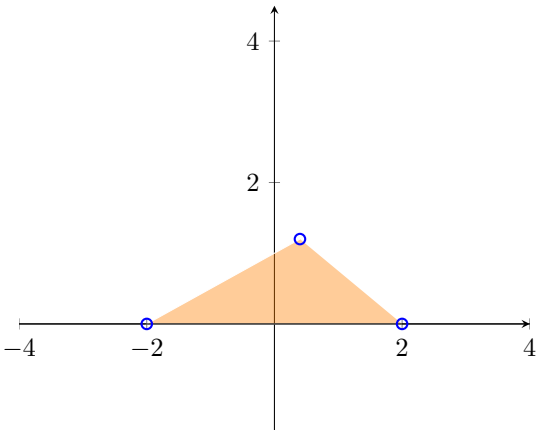

Figure 6: The valid region in the three inequalities from Figure 5. The corner points are $(2,0), (-2,0)$ and $(\frac{2}{5}, \frac{6}{5})$.

If we have room for twenty 1 meter pieces and thirty 5 meter pieces, how many 11, and how many 6 meter lengths should we buy? We cannot go over that amount of pieces, but we can settle for less than that amount.

To answer this, we can make a table to describe it:

Bought Length	Required length	
	5 meter	1 meter
11 meter	2 per board	1 per board
6 meter	1 per board	1 per board
Amount needed	30	20

This decomposes into the following equations:

- $2x + y \leq 30$
- $x + y \leq 20$
- $x \geq 0$, $y \geq 0$

Notice how our shapes always have a convex hull...

Plotting this on a graph gives us what's in Figure 7, where we can see a feasible region between $(0,20), (10,10), (15,0)$ and $(0,0)$. The three corner points are all of the non-zero points (zero eleven meter and zero six meter pieces is an invalid solution), and anywhere on the outer edge of the region is a solution.

3.2 The maximum problem

Sometimes, we want to maximise some particular variable. For example, companies want to maximise their profit. If this is the case, then we can come up with a function for the profit, something like:

$$P = 7x + 3y$$

Where x is the number of 5 meter boards, and y is the number of 1 meter boards. We can factor this equation into our solution, to find the most profitable wood to buy.

If you think about it, the further from the origin we get, the more profit we get. At $(0,0)$ we have no profit, since we have done nothing, but at $(5,0)$ we have 35 profit since we have bought

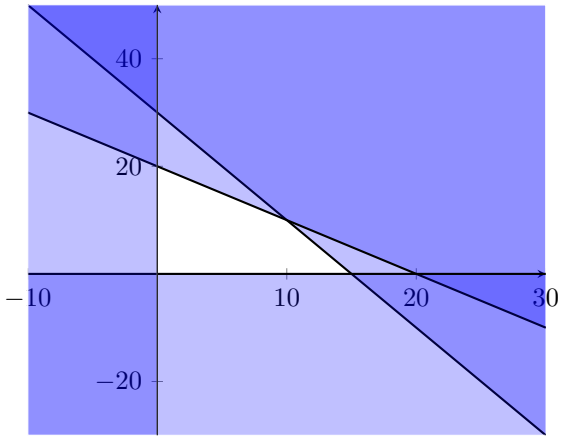

Figure 7: A graph of $2x + y = 30$, $x + y = 20$, $x > 0$, $y > 0$.

five, 5 meter length boards. Since our lines always intersect to produce a region with a convex shape, the points furthest from the origin are the corner points. To figure out how much profit we can make, we just need to find the maximum of these points:

Corner point	**5 meter boards**	**1 meter boards**	**Profit**
$(0, 20)$	20	20	200
$(10, 10)$	30	20	270
$(15, 0)$	30	15	255

Therefore the most profitable choice is to buy 10 11 meter length boards and 10 6 meter length boards.

3.3 The minimum problem

This is similar to the maximum problem, except instead of maximising profit, you'll be minimising cost, while still fulfilling some criteria. You plot a graph, find the corner points of the feasible region and see which one minimises the cost.

3.4 The Simplex Method

The simplex method automates finding the optimum value of variables. This is how it works:

- Select a corner of the feasible region of the graph
- Choose an edge that is connected to this point that will increase the objective function.
- Go to the next corner on that edge
- Keep doing steps 2 and 3 until you find an optimal solution.

4 Turing machines and complexity measures

A Turing machine is a theoretical device that implements a model of computation. It operates on a tape of infinite length, which is divided into cells, and can read and write symbols from/to each cell when the *head* of the machine is positioned over the cell. The Turing machine also has internal state, which is finite and determines what actions it takes (reading, writing, moving the tape etc).

In this course, we will be thinking about multi-tape Turing machines. Here, the there are many tapes, numbered $1 \ldots K$. Tape 1 is the input tape and tape K is the output tape. The tapes

inbetween the input and output tapes are the work tapes. Any algorithm can be expressed as a multi-tape Turing machine

4.1 Formally defining a Turing Machine

A TM (Turing Machine) can be define as a quintuple, as shown in Figure 8.

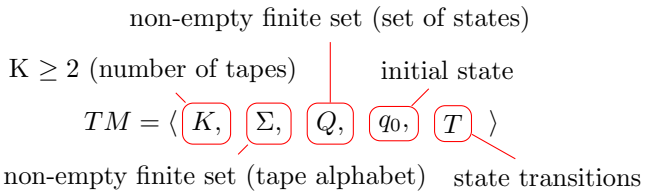

Figure 8: Part of a multi-tape Turing machine definition. We must also define the word *symbol*, which is a mark occupying one cell on the tape. The alphabet that a symbol can be from is $\Sigma \cup \{\text{␣}, \triangleright\}$, where ␣ is a blank cell, and $\triangleright$ is the start symbol (signifying the left edge of the tape).

Up to now, we've defined a TM that is perfectly formed... but does nothing. We need to define how it moves between states in Q. To do that, we define a transition, as in Figure 9.

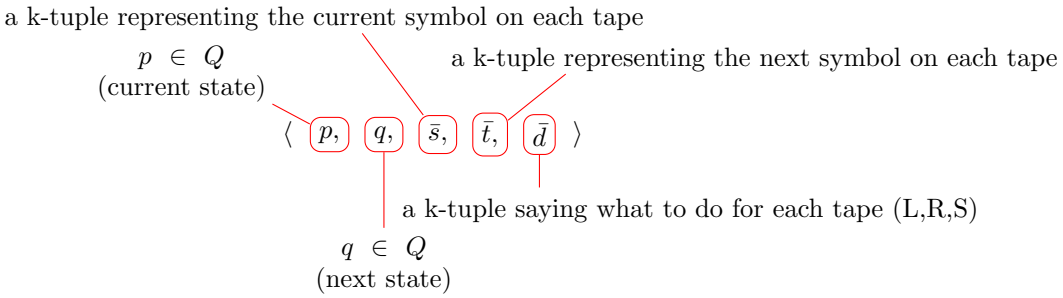

Figure 9: A transition of a Turing Machine

Informally, this transition applies only if the current state is p, and the squares on the tapes confirm to $\bar{s}$, then set the new state to q and write the symbols from $\bar{t}$ to the tapes and move the heads as directed by $\bar{d}$ (Left, Right or Stay).

Now we've nearly finished defining a multi-tape TM, the final points are:

- Since the TM is deterministic, there can be *at most* one transition for every (*state*, *current*$\bar{}$*symbols*) tuple.
- The tape never moves left $\triangleright$.
- Tape 1 is read only and tape K is write only.

4.1.1 Acceptance, computability and decidability

We say that a TM M *accepts* an input x if it halts on x. The language *recognised* my M is the set of strings that is accepted by M.

If a language is recognised by a Turing Machine M, then there is also a deterministic Turing Machine M' that will recognise the same language.

A language is *computable* (aka decidable), if there is a (deterministic) TM that will accept every string in the language, and reject every string not in the language.

Remember, when we say 'Turing Machine', we could mean either a deterministic TM or a non-deterministic one (unless we explicitly said which one we meant of course!).

Up to now, our definitions of recognisable and computable are pretty similar (if not the same). The difference is, that if an input string **isn't** in a computable language, then the TM will halt (and output 'no'), whereas if the input string isn't in a recognisable language, then the TM *may* halt and output 'no', or it could just carry on computing for ever. All a Turing Machine does with a recognisable language, is recognise if a string is in the language, it makes no guarantees about string not in the language!

4.1.2 The Universal Machine

If you're that way inclined, you can find lots of cool things about Turing Machines. Another such cool thing, is that every TM is a finite thing (it's just a quintuple as shown in Figure 8), which means we can come up with a way of encoding any Turing Machine into a string over the alphabet Σ and use it as input to another Turing Machine!

If we can do this, then we can construct a **Universal Machine** U. This machine takes two 'arguments', another TM M (encoded of course) and some arbitrary input x. If $M \downarrow x$, then U finishes with y on its output tape. If $M \uparrow x$, then U will never terminate. The input is coded as $M; x$, where ';' is a separator between the Turing Machine we're 'simulating' and the input we'll give it.

4.1.3 Nailing that TM definition

If you want your Turing Machines better defined than Arnie's Biceps, then this how you do it (alternately, read Chapter 3 of Sipser).

A run of a TM is terminating if it is finite. If the input to a deterministic TM M is x, and the run is finite, we write $M \downarrow x$ otherwise, if the run is infinite, we write $M \uparrow x$.

Let M be a deterministic TM that knows the alphabet Σ, and its input be $x \in \Sigma*$ (i.e. the input is zero or more symbols from the alphabet). If $M \downarrow x$ then the output tape of M will contain some string $y \in \Sigma*$ once M has finished computing. The cool thing, is that we can treat M like a function, lets call it f_M, that operates over $\Sigma* \rightarrow \Sigma*$:

$$f_M(x) = \begin{cases} y & if M \downarrow x \\ \text{undefined} & if M \uparrow x \end{cases}$$

In this case, M computes the function f_M.

4.1.4 Enumerators

Enumerators aren't in the course slides, but they do appear in the reading. Skip this section if you're doing last minute cramming.

An Enumerator is a Turing Machine with an attached printer. Now, I know what you're thinking; *I'm a computer scientist, I hate printers. Why would anybody combine a complicated enough idea like a TM with a printer?!* Enumerators have an important theoretical use though, and don't require us to install drivers like we do for printers, and they certainly don't need to work over WiFi (so this section should be a piece of cake really).

The TM inside an enumerator is able to send a string to the printer whenever it wants to print out a string. If the TM does not halt, then the printer may print an infinate list of strings. The language that is *enumerated* by the enumerator, is the collection of strings that it eventually prints out. Since there are no rules about repetitions or ordering in the output list of the printer, you might want to think about the enumerated language as a set.

So, as I said before, enumerators *are* useful. Namely a language is Turing recognisable if and only if, there is some enumerator that enumerates it.

Its easy to show that a TM can recognise a language that is enumerated by an enumerator, just build a TM that does this:

1. We have an input word w
2. Run the enumerator E:
   (a) Every time E outputs a new string, compare it to the word w
   (b) If they are equal, then accept the word.

Note that there is the potential for this TM to run for an infinate length of time, if the enumerator generates an infinate list, and w is not in the language it enumerates.

We can also make an enumerator enumerate any language that is recognised by a Turing Machine M:

1. Ignore any input
2. For $i = 1$ to $i = \infty$
   (a) Run the TM M for i steps for all possible strings of length i in the language (Σ^i)
   (b) If any computations accept, then print out the corresponding string.

This enumerator will never halt (since it loops infinately many times over an infinately large set of input strings), but eventually, it will print out all of the words in the language (even though there will be *a lot* of duplicates!).

4.2 The Halting problem

The Halting problem asks:

> Given a Turing Machine M and a string for its input x, return:
>
> **Yes** if $M \downarrow x$ (M decides x)
>
> **No** otherwise, where M would compute forever

Turing managed to prove that there is no Turing Machine that will decide the Halting problem. The proof is fairly simple, but it takes a while to get your head around it when its written in a mathematical form. Luckily, I found a poem, written in the style of Dr. Seuss, that provides a better textual explanation than I ever could; see Figure 10.

As nice as it is, I think it's unlikely that the poem would be accepted as an answer in an exam, so we'd better learn the formal definition of the Halting problem too.

Suppose we have a TM P that can determine if another TM can halt. We're going to make another TM Q that is defined so:

1. Duplicate the input x TM we're given.
2. Run P on the duplicated input $(x; x)$.
   $P(x; x) = Y$, then Loop
   $P(x; x) = N$, then Halt

Now, if we give Q the input Q, then the embedded P will receive (Q, Q), meaning that if:

$$Q \downarrow Q \implies Q \uparrow Q$$
$$Q \uparrow Q \implies Q \downarrow Q$$

Scooping the Loop Snooper

An elementary proof of the undecidability of the halting problem

No program can say what another will do.
Now, I won't just assert that, I'll prove it to you:
I will prove that although you might work til you drop,
you can't predict whether a program will stop.

Imagine we have a procedure called P
that will snoop in the source code of programs to see
there aren't infinite loops that go round and around;
and P prints the word Fine! if no looping is found.

You feed in your code, and the input it needs,
and then P takes them both and it studies and reads
and computes whether things will all end as they should
(as opposed to going loopy the way that they could).

Well, the truth is that P cannot possibly be,
because if you wrote it and gave it to me,
I could use it to set up a logical bind
that would shatter your reason and scramble your mind.

Here's the trick I would use - and it's simple to do.
I'd define a procedure - we'll name the thing Q -
that would take any program and call P (of course!)
to tell if it looped, by reading the source;

And if so, Q would simply print Loop! and then stop;
but if no, Q would go right back to the top,
and start off again, looping endlessly back,
til the universe dies and is frozen and black.

And this program called Q wouldn't stay on the shelf;
I would run it, and (fiendishly) feed it itself.
What behaviour results when I do this with Q?
When it reads its own source, just what will it do?

If P warns of loops, Q will print Loop! and quit;
yet P is supposed to speak truly of it.
So if Q's going to quit, then P should say, Fine! -
which will make Q go back to its very first line!

No matter what P would have done, Q will scoop it:
Q uses P's output to make P look stupid.
If P gets things right then it lies in its tooth;
and if it speaks falsely, it's telling the truth!

I've created a paradox, neat as can be -
and simply by using your putative P.
When you assumed P you stepped into a snare;
Your assumptions have led you right into my lair.

So, how to escape from this logical mess?
I don't have to tell you; I'm sure you can guess.
By reductio, there cannot possibly be
a procedure that acts like the mythical P.

You can never discover mechanical means
for predicting the acts of computing machines.
It's something that cannot be done. So we users
must find our own bugs; our computers are losers!

Geoffrey K. Pullum, Scooping the loop snooper: An elementary proof of the undecidability of the halting problem. Mathematics Magazine 73.4 (October 2000), 319-320

Figure 10: A poetic explanation of the Halting Problem.

4.3 Complexity

You already know that there are two types of Turing Machine; deterministic and non-determinitic TM's, but now its time to think about the resources that Turing Machines consume. We don't think of TM resources as commodities such as metal, electricity etc, since TM's are abstract, theoretical devices. Instead the two commodities we're interested in with Turing Machines are **time** and **space**.