13. The Search is On (Search Algorithms)

Canst thou by searching find out God?
The Bible, Job Chapter 11, verse 7.

Linear Search
Take a pack of cards. Shuffle it thoroughly. Now find the Four of Spades. How did you do it? Put that card back and shuffle the pack again. Now find the Nine of Hearts. Did you look for the card the same way as the first time? Suppose the card you were looking for was missing. How would you have discovered this fact when searching? At what point in the search would you have known for certain that the card was not there?

If you used some methodical way of finding the card that you followed each time then you were using a search algorithm. Most people would search for a card in a shuffled pack by starting at one end of the pack, and checking the cards one by one until they came across the card they were looking for. This method of searching is well known in Computer Science. It is called **Linear Search**. It is one of the simplest ways of searching. You know when the thing you are looking for is not there when doing a linear search if you have reached the end and have not seen it. This is obviously so, because by that point you have inspected every card in the pack.

Suppose like me you have a pile of CDs next to your Hi-fi. Each time I play a CD I put the old one that was in the machine back on the top of the pile. This means the pile ends up in a totally random order. Suppose I want to find the CD *Regatta de Blanc* by the Police. How do I find it? I start at the top of the pile and check them in order until I get to it. If I got to the bottom of the pile I would know it was not there, and was probably in my CD walkman. I am again using the linear search algorithm.

Suppose you need a taxi from home to the airport. How would you find the phone number of one? Get a copy of the Yellow Pages and go to the Taxi section. Now pick a taxi firm. You could just take the first, but its probably best to get one that is based near to where you live so that it is more likely to be on time. Chances are you would start with the first one and check through them in turn until you came to one with an address near to you. The owners of taxi firms know this is what you are likely to do. They also know that the Yellow Pages puts firms in alphabetical order. That is why so many taxi firms are called things like AA Taxis. Guesthouses use the same trick. I have stayed in one called A&B Guest house, for example. The names of many businesses are thus the way they are because of the linear search algorithm!

On the corner of my desk I have a pile of papers that represent my "TO DO" list. New things to be done get added to the pile, which I gradually work through. Suppose my Boss came in asking me why I never returned the form he sent me that should have been filled in the day before. I would have to quickly find it. Unless I had some idea where it was in the pile, I would have to start at the top and work down the pile one at a time. Again I am using linear search.

The book, *The Diving-bell and the Butterfly*, (Bauby 1998) was written (or at least dictated) using linear search. It is the autobiography of Jean-Dominique Bauby, the

former editor-in-chief of the French magazine Elle. He suffered a massive stroke and was left totally paralysed, unable to move or speak, other than being able to move a single eyelid. Rather than give in to his disability, he instead decided to write his autobiography. In the book he describes what life is like for someone who is totally paralysed, including how he communicated with people and so managed to write the book. He dictated it to a secretary, just using blinks. This is where linear search came in. The secretary read through the alphabet a letter at a time. When she read out the letter that came next in the word Bauby wanted, he would blink, and she would write it down. She would then start at the beginning again looking for the next letter. Letter by letter, the whole book was written in this way. Each letter was found by a linear search of the alphabet. The algorithm was improved slightly from a straight linear search of the alphabet in that the letters were ordered by frequency in French (Bauby's native language and that in which the book was written). E is the most common letter so it was first in the list, then S, A, R and so on finishing with W, the least commonly used letter. All Bauby's communication to the outside world was done in this way.

The game of Hangman is played using a variation of linear search. The problem is to work out a word one of the other players has thought of. You only know the number of letters in the word to start with. The search task is to search through the letters of the alphabet and work out which are in the word. You might start by guessing E as it is the most common letter in the English language. Next you try S, then perhaps A and R. With each letter tried, you are told either that it is not in the word (and lose a life) or the positions that word occurs. In the earlier stages of the game you have effectively lined up the letters of the alphabet in order: E,S,A,R... and are doing a linear search down them, until all the letters of the word have been found. Of course if you are playing well, you will not have the alphabet in a fixed order but will start to guess the word and change the order of the letters based on the results of the previous guesses.

The game of I-spy is also played by something similar to linear search. One person thinks of a word and tells the other players its first letter: “I spy with my little eye something beginning with T”. The players then list the things they can see that start with the letter T: “Toe”, “Toy”, “Television”, ... When a person names the thing the first person had thought of the search (and the game) stops. Otherwise the game continues until the players have tried every possibility they can think of. The search is being done in a linear fashion: one by one, and on each search question one possibility is ruled out. The difference here is that you do not start with a full list of the things to search. You may not ever come up with the answer if you do not think of it. You are performing a linear search of the things you can think of and see that start with the given letter.

Linear Search is so commonly used because it is simple and relatively quick provided the amount of things to sort through is small. It is a natural way to search when the things being searched through are in random order, and you do not expect to have to search for things very often. Does the fact that it is so common mean that it is the only way to search for things, or that it is always the best search algorithm? Things certainly do not always seem to go as well as they should. How often have you searched for something in one place after another, only to have the thing you were looking for in the very last place you could possibly have looked? That is one of the disadvantages of linear search. In its worst case you have to check everything.

Let us try and write out the algorithm for linear search. If in all the above situations we are following the same algorithm, then we ought to be able to write one set of instructions that works for all. Lets start with the problem of searching through a pile of books. We are doing something over and over again – so we have some form of repetition. We probably do not know how many books there are to search through so it is not a counter controlled loop. We stop if we have found the thing we are looking for or we have run out of books to check. In other words we keep going while the current book is not the one we are after and while there are still books to look at. Our algorithm is going to look something like:

while you are **not** holding a book **and**
your finger is **not** at the bottom of the pile
do the following repeatedly
....

What do we do repeatedly? We check the current book and ask if it is ours or not. We are making a decision and will do different things depending on whether it is ours or not. That sounds like a 2-branched if statement.

while you are **not** holding a book **and**
your finger is **not** at the bottom of the pile
do the following repeatedly
if your finger is against the book you want
then ...
else ...

What do we do if the book is the one we want? We take it out of the pile – we have found it. What do we do if the book is not the one we want? We move on to the next one.

while you are **not** holding a book **and**
your finger is **not** at the bottom of the pile
do the following repeatedly
if your finger is against the book you want
then take that book from the pile
else move your finger to the next book.

So we check that we are not holding the book, and have not run out of books to check. If so we ask if the current book is the one we want. If it is we take it from the pile. If not we move to the next book (the current book is now the next book). We then go back to the loop question and see if we have finished yet.

There is one thing still missing: initialisation (remember I warned you about forgetting that!) What is our finger doing before we start? (it could be painful if it is picking our nose at the time when we start to follow the algorithm). We have not explicitly said where in the pile of books to start: we must start at the top as otherwise we will not check some of the books so fail to find the book we are looking for (so its important!). We also have not said whether or not we are holding a book at the start.

To find a book in a pile **do the following:**

1. Put down any books you happen to be holding.
2. Put your finger against the top book.
3. **while** you are **not** holding a book **and** your finger is **not** at the bottom of the pile **do the following repeatedly**
   if your finger is against the book you want

then take that book from the pile
else move your finger to the next book.

Try this algorithm out on a real pile of books to make sure it really does work – remembering to follow the instructions rather than doing what you think you have to do.

Problem 1

Write out a version of the linear search algorithm for the situation when you are searching for a given DVD in a pile.

Problem 2

That should have been easy as you just have to replace the word "book" for "DVD". Write out a version for a Nurse who Jean Dominique Bauby is trying to communicate a single letter to (assume normal alphabetic order is used).

That needs slightly more changes but the basic structure of the algorithm should be the same – the same loop with similar tests and a similar if then else statement inside.

To find a letter being thought of from the alphabet **do the following:**

1. Take a blank piece of paper.
2. Say the letter A.
3. **while** you have **not** written a letter down **and**
   you are **not** at the end of the alphabet
   do the following repeatedly
   if the person blinked
   then write down the last letter you said
   else say the next letter of the alphabet.

The structure of this algorithm is identical to the version for books. Notice we have still accounted for the situation when you end up not writing a letter down – if you get to the end of the alphabet the search failed. Perhaps he was not trying to communicate anything to you! However I have been slightly lazy in that I have assumed that if the last letter said is Z then when given the instruction to say the next letter the person will say something sensible like "there's nothing left" or just say nothing. Strictly I ought to have spelled this out. In fact whatever is said then is a sentinel value – not a letter of the alphabet but something else that means the end. By "at the end of the alphabet" I mean the person has just said that sentinel value.

Problem 3

Modify the above algorithm to explicitly use the sentinel value "END".

Since it is a single algorithm, we should be able to write a general version of it that works whatever kind of thing we are searching through. We can then use it whenever we wish to do a linear search. We do this by taking the parts of the algorithm that are specific to a particular problem (like books or letters) and change them to something more general, but leaving the structure alone. For example we could replace the word

"letter" by "thing". We also need to change various of the sentences to make sense when talking about general things. Here is one more general version of the algorithm.

To find a *thing* from a *series of things* **do the following:**

1. Note that you have not found the *thing* yet.
2. Set the current *thing* to be the first *thing*.
3. **while** you have **not** found the *thing* you are searching for **and**
   you are **not** at the end of the *series of things*
   do the following repeatedly
   if the current *thing* is the *thing* you are searching for
   then you have found the *thing* you are looking for
   else make the current *thing* the next *thing* in the sequence

We now have a general set of instructions (algorithm) called linear search that we can use to search for things. There are other ways of writing it that amount to the same thing and so are still linear search. However linear search is just one way of searching (and actually not that good a way in many situations). We will now look at some others.

Binary Search

There are actually many search algorithms, used for different purposes, and you probably use variations of several without even thinking about it. Have you ever played the game of 20 Questions? This involves one person thinking of a famous person. The other player has to work out who it is just by asking Yes and No questions. You try to do it in as few questions as possible. If you take more than 20 questions you lose. This is just a search problem – we are searching for the name of a person out of the millions of famous people there are in the world. If the game is just a search problem then we could play it by doing a linear search. How would this work? We would ask a series of questions of the form "Is it X?" as with I-spy. For example a typical game might go:

"Is it Nelson Mandella?"
- *"No"*

"Is it Arundhati Roy?"
- *"No"*

"Is it Freddy Mercury?"
- *"No"*

If you played this way, you would probably lose every time except for perhaps on a few very lucky occasions. These lucky occasions would also probably convince you that you could read minds! You would only win if the correct answer were one of the first 20 people you thought of. Given there are millions of people to choose from you do not have much hope (unless of course you *can* read minds!) So why do people ever play? And how come they often manage to get the right answer in much less that 20 questions? The problem with linear search is that you rule out only one answer (or thing in the pile you are looking through) at a time.

Suppose you were playing 20 questions with me. What would you ask first? A common first question is

"Is the person male?"

Other questions that you might ask early on are

" Is the person alive?" and " Is the person fictional?"

Why are these questions good ones to ask and in particular better than giving a series of names? They all have the obvious disadvantage that you have no chance of winning on that question itself. Are they then wasted questions?

The big advantage of such questions that makes it worthwhile "wasting" the chance of winning on that turn is that they rule out large numbers of people in one go. Most importantly they do this whatever the answer to the question. Questions such as "Was the person a member of the rock group Queen?" rule out a large number of people if the answer is YES. However, they only rule out the 4 members of Queen if the answer is NO. Since the latter is the most likely outcome until, perhaps, the later stages of the game, it is not a good question. Thus the ideal question is one that rules out half of all people if the answer is yes, and the other half if the answer is no. That means it does not matter to you what the answer is, you are equally close to the person's identity. The "Queen" question would be worthwhile if somehow you had narrowed down the search on the previous questions to being a member of either Oasis or Queen for example. The best of the above questions to ask first is thus the male/female one. The more questions that divide the remaining possibilities in half that you come up with, the quicker you will get to the answer. Halving the population repeatedly very quickly takes you to a single person. If there were a million possible candidates, it only takes 20 such questions to *guarantee* narrowing down the search to a single person: and that really is a guarantee! Compare that with linear search. If you are very, very lucky you might get the right answer the first time (I once won in 2 questions!). In the worst case, however, the correct person could be the last one you asked about, assuming you had the time to ask a million questions. The skill of the game is of course to come up with good questions that do keep splitting the field in half. If a question does not split the field in half every time it could take more questions, and you no longer have that guarantee.

Does this approach to searching only work for 20 questions or can it or variations work on other kinds of thing being searched through? Get a residential telephone directory and find a friend's telephone number in it. How did you do it? Did you use linear search – starting at the first page and checking every entry until you found your friend's? If you did, it probably took you a very long time (unless their name happens to be something like Aahann, Aammir or Aaronovitch – and it is very bad news if their name is Zwiebel, Zygovistinos or Zykun). It is more likely that you used a variation on the 20 questions algorithm (if a little haphazardly). If the name was Steinbeck, for example, you would not start at the first page, but open the directory somewhere in the middle. There is little point starting at the beginning, after all, as a name starting with S is more likely to be near the end. However, it is possible that you have over shot the page you wanted. You would therefore check what the names started with on the page you had opened the directory at. If you are at a page before the name you want, then you can rule out the first half of the directory and concentrate on the second half. This is possible because the telephone directory is sorted. If it were in a random order, you would be no nearer finding the right entry. If you have overshot, then you can rule out the second half and concentrate on the first half. Now you have only half as many entries to search, and can continue in the same way, go to a page roughly half way through the part you have not discarded and discard another half. Keep doing this until you get to the single entry that is your friend's name.

Let us suppose I opened the book at McDonald. It is before Steinbeck, so I ignore all pages before that point. I move to a page half way between McDonald and the end of the book. This time the first name on that page is Tambe, so I overshot. I now go half way between McDonald (assuming I remembered to keep my finger in that place) and Tambe finding Poulter: I overshot in the other direction this time. I go forwards again to a point halfway between Poulter and Tambe and find Shah. Not far enough, so I split between Shah and Tambe. Smith: not far enough. Split between Smith and Tambe: Stanton. I am now on the correct page (so could continue with the same process working through the page). With 7 questions I have narrowed down a whole telephone directory to a single page. A few more and I would be down to a single entry. With Linear Search I would only have made it as far as Aarrons in the same time!

This approach of searching things by dividing the field in two is called **binary search.** As with good 20-questions play, this search algorithm halves the number of entries to search on each "question". The great thing here, though, is you just keep asking the same question every round! The question in this case is "Is the entry in the place I opened the book earlier than the one I want". It is making use of the fact that the information to be searched has been pre-organised: it is sorted.

Let us return to searching a pack of cards. If you thought the pack was shuffled, you would probably start the search by linear search. However, if after checking a few cards they seemed to be in order, you would quickly abandon that approach and switch to something similar to binary search, jumping ahead and ruling out whole portions of the pack in one go.

The reason we can find entries in books such as dictionaries so quickly is because time was spent by the editors organising the entries. It is only because they are organised in a known (alphabetical) order that we can use variations on binary search. Sorting the entries will have been a great deal of work for the editors, but by doing that work once, the time taken by the many people subsequently using the dictionaries to search for things is much shortened. A similar approach is used in a variety of algorithms: you spend longer at the start organising the information once and for all, so as to save time doing something that will be repeated many times later. Provided the repeated part is done often enough this will save time overall. Today it seems inconceivable that a dictionary or telephone directory would be in anything but alphabetical order. However, one of the first ever dictionaries (in the 16th century), John Withals' *Shorte Dictionarie for Yonge Begynners*, was actually ordered by subject (Winchester, 1999), so the realisation that alphabetical order would be the most useful organisation was clearly not immediately obvious.

Jean-Dominique Bauby, the person who was totally paralysed but still managed to write a book, could have made his task much easier if he had known of binary search. Instead of having the person he was communicating with work through the alphabet a letter at a time, they could have started with M. Instead of a blink meaning "Yes", it could have meant "later in the alphabet", with no blink meaning "earlier in the alphabet". A double blink would mean "that is the letter". Each time the next letter read out would be roughly half way through the interval remaining. This would have taken the secretary longer to learn to do, perhaps, but it would have meant *every* letter of the alphabet could have been identified in only 5 blinks. With Bauby's algorithm,

only the letters E,S,A,R and I could be identified that quickly. The least common letters would take up to 25 blinks. The book is139 pages long with around 180 words per page and perhaps 7 letters per word: something over 175 000 letters. Binary search would have saved an awful lot of blinks. That is just for the book. If all Bauby's communication had been with binary search, communication would have been significantly less frustrating both for him and for his friends and relatives. An understanding of algorithms can thus make real difference to a person's life.

The German developers of a brain scan based system developed for sufferers such as Bauby did know of Binary search. They developed an electroencephalogram (EEG) headset that can read the wearer's mind, at least as far as recognising when they think "yes". They needed a way of turning this small ability into a full communication system. Here is a description of what they did with the person's thoughts:

> *"If the subject's brain activity gave a "yes" signal, the group of letters was split, and then split again until only one letter was left – a letter that began to spell a word, and then a sentence."* (Radford, 1999)

Thus binary search has even be used to read minds.

It probably occurred to you that when you actually search for a name in a telephone directory, (or a word in a dictionary) you would probably do it slightly differently. If looking for Steinbeck, you would not open it in the middle as that would almost certainly take you to the middle letter – M – as it did in the example above. S is nearer the end of the alphabet so you might open the directory two-thirds of the way through. You would do this on each step: using your knowledge of the positions of letters in the alphabet, to make a better guess than halfway each time. This is an optimisation of binary search where you are using even more knowledge (the positions of letters in the alphabet) about the thing being sorted to speed things further.

Here is a new game I have just invented that is similar to 20-questions. It is called 10-questions. Just as in 20-questions, one person thinks of a famous person. The rules for questions are now slightly different. The questions no longer have to be ones with YES/NO answers but can have up to four alternatives (including as one of the alternatives, "none of those"). The person giving the answers must say which of the alternatives is the case. For example the person guessing might ask as their first question: "Is the person you are thinking of a) Asian b) African c) European d) none of those". The answer given might be "a) Asian". A question later in the game might be: "Is the person you are thinking of a) Nelson Mandella b) Archbishop Tutu c) neither of those. Notice that you do not have to always give 4 alternatives, the questions could have two three or four alternatives, as long as one of the alternatives is "none of those". However, you now only have 10 questions to get the answer. Play a game or two to get the idea. We decided that a perfect player of the original 20-questions would come up with questions that divided the possible answers in two. Why? Because that way, whatever the answer you always rule out the same number of people. Does the same thing apply to this game? Is the best strategy still to ask essentially yes/no questions? That is still allowed but is it still best? Perhaps one of the options should cover half the alternatives and the others can be anything?

Problem
Before reading on decide what you think the best questions to ask are.

Problem
Is it easier or harder to get the answer in 10 questions than in the original game with 20 questions to ask?

Suppose with earlier questions we have narrowed down the search so that we now know the answer is one of the four Beatles. We have several questions left but obviously want to be sure to get the answer as quickly as possible. Here are some questions we could ask.

Is it a) John Lennon or b) one of the others?
or we could ask
Is it a) John Lennon or Paul McCartney, or b) one of the others?
or we could ask
Is it a) John Lennon or b) Paul McCartney or c) one of the others?
or we could ask
Is it a) John Lennon or b) Paul McCartney or c) Ringo Star or d) the other one?

Which of these questions would you ask?

If you ask the first question then you would not even be playing good 20-questions! If you are right and it is John Lennon then fine. However if wrong you still have 3 choices left, which is not so good.

If you ask the second question then you are playing good 20-questions. That question splits the options in half, so whatever the answer there will be two choices left and one more question is guaranteed to get it. That sounds good, but we only have half as many questions over all so really we need to be able to do better than 20-questions.

If you ask the third question you are trying to make use of the extra flexibility you have but not fully – perhaps you thought John Lennon and Paul McCartney are the most famous so its worth naming them. However, if that hunch is wrong we still have two possibilities left and need a further question.

Of course the last question is the best question. Whatever the answer it rules out three people: three quarters of the alternatives. We are sure of the answer with only one question. Suppose the question before we knew the answer was either a member of the Beatles, a member of the Spice Girls, a member of Queen or a member of REM (all pop groups with 4 members). Obviously the question to ask is

"Is the person in a) Queen, b) REM c) the Beetles or d) none of the above (ie the Spice Girls)?"

Why because whatever the answer we rule out three-quarters of the options. We will then be sure of getting the answer in one more question. The tactic to use is to try and come up with questions that *always* rule out roughly *three-quarters* of the population whatever the answer. That is each choice should cover an equal number of people. This is actually the same reasoning as with the original game. We only have 2 choices of answer so we want to be left with half of them with each question. Now we have 4 choices of answer so we aim to have only a quarter of the people left with each question. In both situations we want each possible answer to cover the same number of people as that is the only way we can be sure that whatever the answer we get the

same outcome. Suppose we were allowed only questions with a choice of three possible answers, what fraction of people would we want each answer to cover?

So when playing the game of 10-questions, the best players will try to come up with questions where each option covers a quarter of the people left. This rules out more people with each question than in 20-questions so it must take fewer questions to get the answer. How fewer? Suppose we have asked questions in both games so that there are exactly 64 possible people left.

With the rules of 20-questions and each question ruling out half the alternatives. The first question leaves 32 people (dividing 64 in half), the second question leaves 16 people, the third question leaves 8 people, the fourth question leaves 4 people, the fifth question leaves 2 people and with the sixth question there is only one person left – we know who it is in 6 questions.

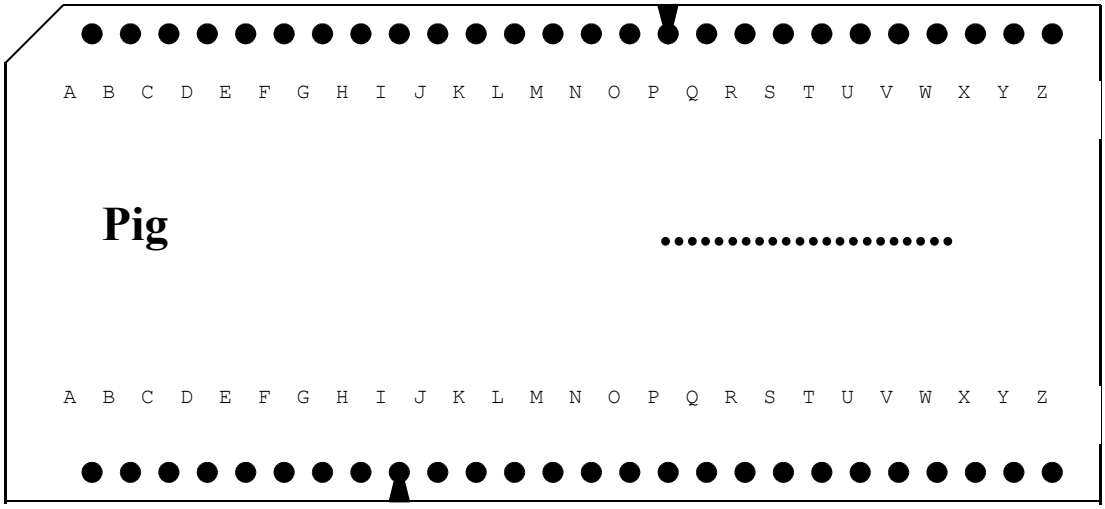
With the rules of 10-questions on the other hand and each question ruling out three quarters of the alternatives. The first question leaves 16 people (dividing 64 in 4), the second question leaves 4 people, and with the fourth question there is only one person left – we know who it is in only 3 questions. It is twice as fast. If you do the same reasoning starting with a million or so people you will find that 10 questions in the game of 10-questions is exactly as efficient (ruling out the same number of people) as 20 questions in the game of 20-questions. Just as 20-questions played well corresponds to binary search. The game of 10-questions corresponds to a faster algorithm of dividing into four. The more categories you can divide into in a single question the faster the algorithm will be. When we open a dictionary in roughly the correct place for the first letter of the word we are looking for, we are very roughly doing this. The "question" is which of the 26 letters of the alphabet does it start with. We then go directly to that section.

Could we use the observation that 10-questions is faster than 20-questions to help come up with a faster algorithm for someone with locked-in syndrome to use to communicate? This would require the paralysed person to be able to indicate one of four options. If they could only blink one eye then you would need some system of single or double blinks. If they could blink both eyes then there are four signals possible: no blinks, blink left, blink right, blink both at once. The question now asked would not be first or second half of the alphabet remaining, but, first quarter (no blinks, second quarter (left blink), third quarter (right blink) and fourth quarter (both blinks). The first question would narrow the search down to A-F, G-M, N-S, or T-Z. Each of these has 6 or 7 possibilities in. Suppose no blinks were communicated, this would mean A-F. The next question might be AB, CD, E or F. (As there are not 8 alternatives we do not have 2 choices in each category). That question might give us the answer, but if not (say left blink was communicated meaning C or D) a third question (C or D?) would give the answer. At worse we get the answer in 3 questions but at the expense of being able to blink with both eyes and it being harder to determine what the answer means.

Binary search works well for computers because the basic "questions" computers can ask are binary yes/no ones as we saw when discussing if-the-else statements. There are situations where we can overcome this restriction as we will see. Of course, if we would ask a question that can have any number of options as answer then it would

tells us which of the alternatives the answer is directly in one question. As we will see that idea leads to a different algorithm (lookup-tables and bucket searching) that have other disadvantages.

When I was at School, our R.E. teacher arranged for a Missionary who had just returned home from Papua New Guinea to talk to us about his life there. The most interesting thing he described (to me at least) was the way he learnt the local language. He did not have an English-Papua New Guinean dictionary as none existed at the time so he had to write one himself as he went along. I kept a similar vocabulary book of new words when I was learning French at school. The problem with my vocabulary book, however, was that the words were not in alphabetical order but in the order I came across them. This was very infuriating, as often I knew I had come across a word, but just could not find it. The Missionary used a different storage mechanism (data structure) which allowed him to use a search algorithm similar to binary search even though the words were never sorted. Each time he came across a new word, he would write it and its translation on a card, so that he had one card for each word. These cards acted as his dictionary. He did not keep the cards in alphabetical order as in a normal dictionary, however. Instead each card had a series of 26 holes along each edge, with each hole labelled by a letter of the alphabet.



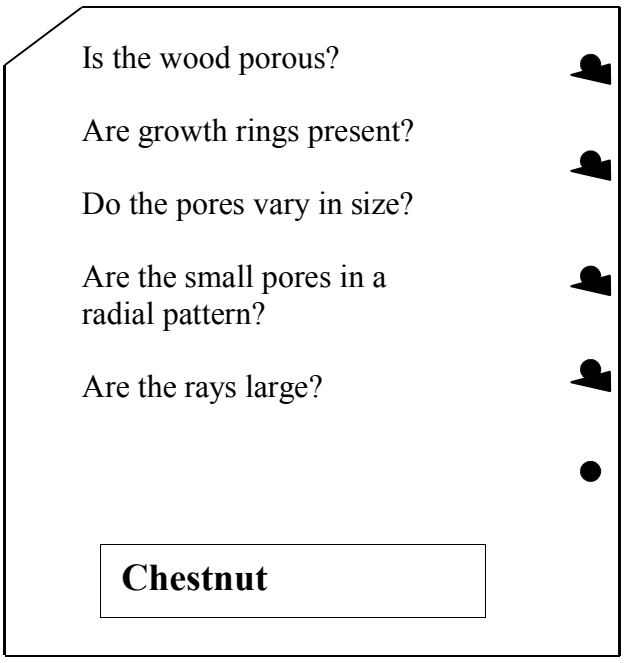
The topside of the card represented the first letter of the word. The bottom side of the card represented the second letter of the word on the card. Suppose the Missionary was told the Papua New Guinea word for Pig. He would cut a notch in the edge of the card into the hole labelled P. He would then cut a notch in the bottom edge against the letter I (as in the above diagram).

The word Pig and its translation would be written on the card and it would be put in the pile of other words. If at some later time, he needed to know the word for Pig, he would take the pile of cards and place a knitting needle through the hole labelled P on the top edge. He would then hold up the knitting needle and shake it. All the cards with a notch cut in the P (cards for words starting with P) would fall to the ground, leaving all the others on the knitting needle. They would be discarded, and the knitting needle would be put through the hole labelled I on the bottom edge of the cards that had fallen to the ground. After another shake only cards for words starting

PI would fall down. Usually this left a small enough number of cards that the correct card could be found using linear search.

The cards are being used to do a variation on binary search in a similar way to 10-questions. The first "question" narrows down the search task to words starting with P. The second to those starting PI. By putting more holes round the cards, further letters of the words could have been labelled. The first question leaves you with a 26^{th} of the original possible words, and the second a 26^{th} of those (assuming their are roughly the same number of words starting with each letter).

The same search algorithm using cards can be used to help identify objects – in the same way as 20 questions works. In the 1950s, the Forestry Products Research Establishment used a similar card system to classify the thousands of different species of trees that exist. Given a series of known facts about a tree the cards were used to work out which species it was. Here, the holes on the cards represented YES/NO questions, such as "Are growth rings present?" and "Is the wood porous?" A hole with a notch represented "YES" and no notch represented "NO" as the answer to the question. The name of each tree species was written on a card, and notches cut in the holes for which the answer to those questions was YES for that tree. The same approach, inserting a knitting needle through the pack was used to answer the series of questions about the tree to be identified. If the answer to the question for the wood under scrutiny was YES, then the cards that dropped were kept. If the answer was known to be NO, then the cards on the knitting needle was kept. Eventually, only one card would remain and it would hold the name of the tree under scrutiny. Note that the order the questions are asked is important. This is because the answer to one question may lead to another being superfluous. For example, there is no point asking if the smell is leather-like if we already asked if the pores are in a radial pattern since there are no trees with these two properties. We will see later how our questions actually form a tree data structure.

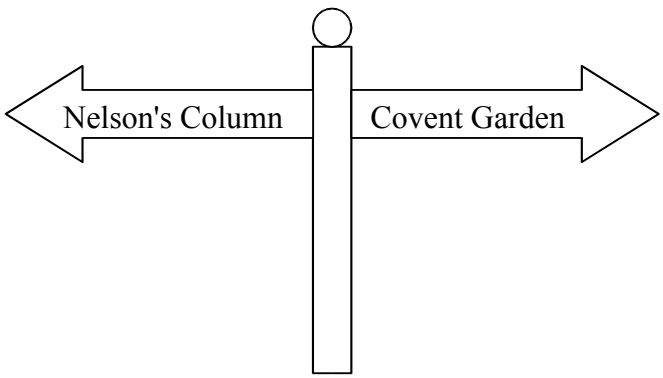


Tree Searching

Binary search is fast because it makes use of the existing organisation of the data: that the data is sorted. However, the information is still placed in a long list both for linear search and for binary search. We can organise data in other ways, however. A tree is a way the data can be organised to give a search algorithm similar in some ways to binary search.

Imagine you are a tourist on a day-trip to London. You arrive at Charing Cross Station and wish to visit Nelson's Column. As you have never visited London before, you do not know which way to go. However, a friend has told you it is in easy walking distance. You are faced with a search problem. How to find Nelson's column among the thousands of other places in London. You could buy a map, but they can be difficult to follow. You could ask for directions. However, perhaps you are a foreign tourist and do not speak English very well so you are not confident that you would understand the instructions. Perhaps like me you know that even if you did understand you would still get mixed up and go left-right-right instead of left-left-right. You could get a taxi, but that is expensive (and you are worried that you might be making a fool of yourself if it turns out that Nelson's Column is actually just round the corner!)

Luckily the London Authorities want you to find Nelson's Column too along with many of the other places you might have been wishing to visit such as Covent Garden, Leicester Square, the National Gallery, etc. They have therefore done some organising in advance. This is only worth their while because they know there are lots and lots of people attempting the same search problems every day – all those tourists. As you walk out of the station, and are wondering which direction to go in, you see a signpost. It does not give you all the directions, just the first stage: which street to go down immediately. It also gives directions to other places too, so you might find yourself with people going to Leicester Square, though the tourists wanting Covent Garden have gone the other way. As you arrive at the next junction your step starts to falter, as you realise you do not which way to go again. However, just as the doubt sets in, you see another signpost. The Leicester Square tourists now split off another way, and you continue. At each junction you are told the way to go next. You are never given the whole instructions, just the information needed immediately at each point.



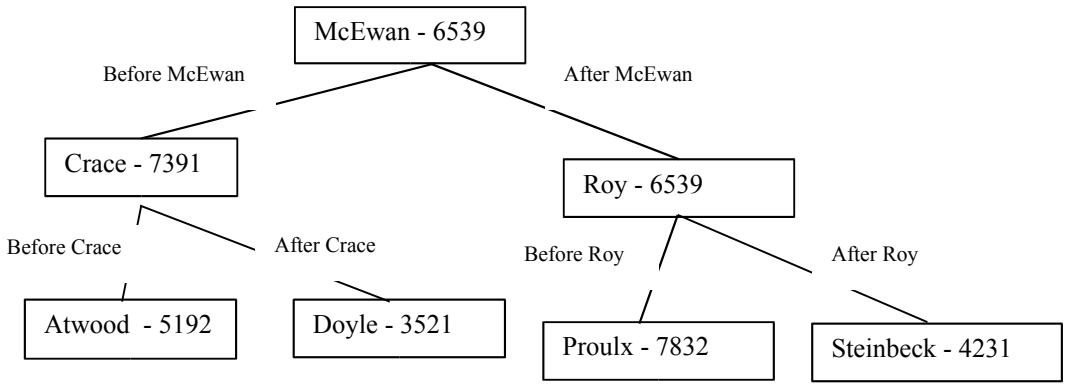
This differs to the approach used by a taxi driver. London cab drivers spend months learning the best ways from any place in London to any other. They spend their days

before being licensed, riding round on mopeds learning the streets. Before they become a cab driver they must pass a test to prove they do know the way from anywhere to anywhere else. All the information of all routes is stored in one place: the cab drivers head. With the signposts, we have spread the information out amongst the street junctions. The workman who put up the signs was effectively building a tree of information. Each junction is a node in the tree, and each road an edge. It allows you to search for Nelson's Column using an algorithm called **tree searching**. Think of a (simplified) view of the streets from Charing Cross Station. Charing Cross is the root of a tree-like structure. There are two branches out of it – the street in each direction. Each road junction is a node in the tree: at each junction there is a split and two or more different ways to go.

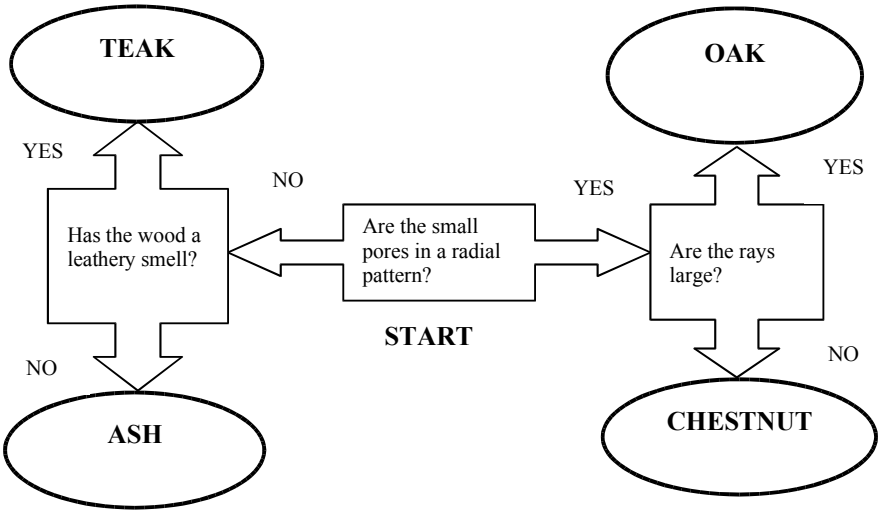
Tree searching involves building a structure like the tree of roads and junctions out of the things to be searched through, rather than just putting them in a long list. The data is placed in the nodes (junctions) along with signposts indicating what is down each path onwards out of the node. By starting at the root, the thing being searched for is found by following the signposts.

If there were lots of data, then this suggests that each branch would have lots and lots of signs – one to each destination. Imagine having a signpost outside Charing Cross Station that gave the direction to go in for every single tourist sight in the whole of London. It would be a gigantic signpost (and finding the direction to your destination would be a time-consuming search problem in its own right). To avoid this problem signposts give directions to classes of places. Leaving London on the M1 the signposts list the nearest cities. All the others are classed together as, for example, "The North". In the opposite direction, they are grouped as "The South".

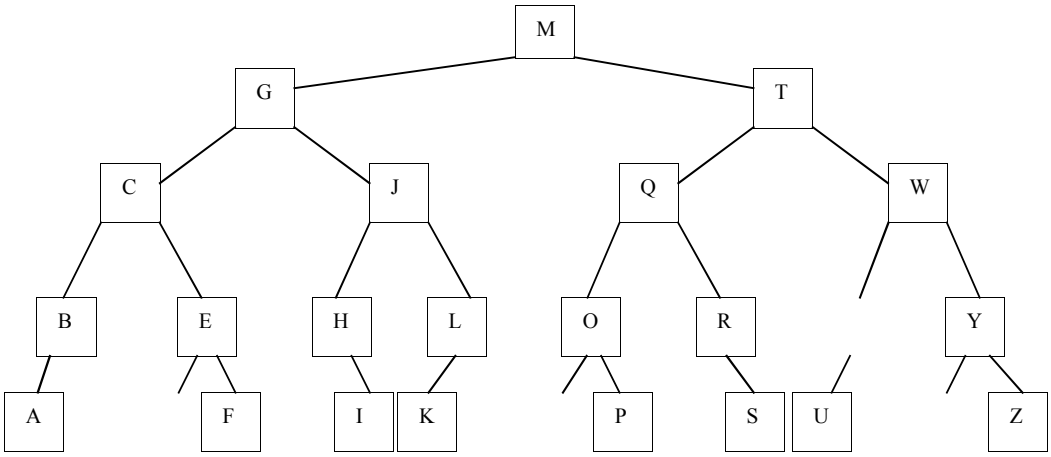
In a tree built for tree searching, we do the same trick. Suppose we are searching for names in a tree-based directory. We start at the root node (which might hold the telephone number of McEwan say) and look at the signpost. It would not list all the names in the directory, but would be arranged so that all the names before McEwan in a dictionary order lie down the left branch, and all those above McEwan lie down the right branch. Unlike with tourist attractions, we can put things where we like in our tree. Organising things in this way is similar to the way a normal telephone directory is sorted: names are not just entered in the order that people applied for telephones. Thus we are using two "classes" of names. Those alphabetically later than the name in our node and those alphabetically earlier. Our signpost in the first node just needs two signs "Less than McEwan" and "Greater than McEwan". Each node is organised similarly. It holds a name and the corresponding phone number. One branch leads to all the names that come earlier alphabetically than the name in this node. The other leads to all the names that come later. You compare the name you are looking for with a node, and if its not yours move on to the node indicated by the signpost. In this way, you quickly find the phone number you are after. Why is it quick? Just like in Binary Search, we discard half of the data with every decision. In a tree search we discard the whole sub-tree down the branch we ignore. Once we decide that the name we want is after McEwan we have discarded in the following example, Crace, Atwood and Doyle in one go.



Search trees are also used for classifying objects as an alternative to the card system we saw earlier. For example, the diagram below is adapted from one in an encyclopedia about timber (Bramwell, 1976) for identifying wood. We show only the branch for porous trees with growth rings. Even though at first sight this diagram does not look very tree shaped, it is a tree in our sense. It has a root node (labelled "start") and each node has two branches out each leading to either a leaf or another node. By adding more questions at the start (root) node, we would be able to classify a wider variety of wood. We start at the start node and ask the question there. The answer sends us one way or the other down the search tree, ruling out half the possibilities as we go. Eventually we get to a leaf of the search tree where we are given the name of the wood we are trying to identify.



Jean-Dominique Bauby could have made use of tree searching to aid his communication with visitors. We saw earlier how binary search would have made it quicker for him to communicate. The problem with this is that without practice, working out the next letter to ask, halfway through the interval left, is hard. Bauby had a similar problem with his method, since few people can recite the alphabet by letter frequency: E, S, A. R, etc. The problem was solved by giving visitors a list with the correct order on. We would similarly solve our problem by giving visitors a tree. Instead of working along a list, they follow the branches of the tree. Bauby's blinks now indicate whether to go down the left branch or the right branch.

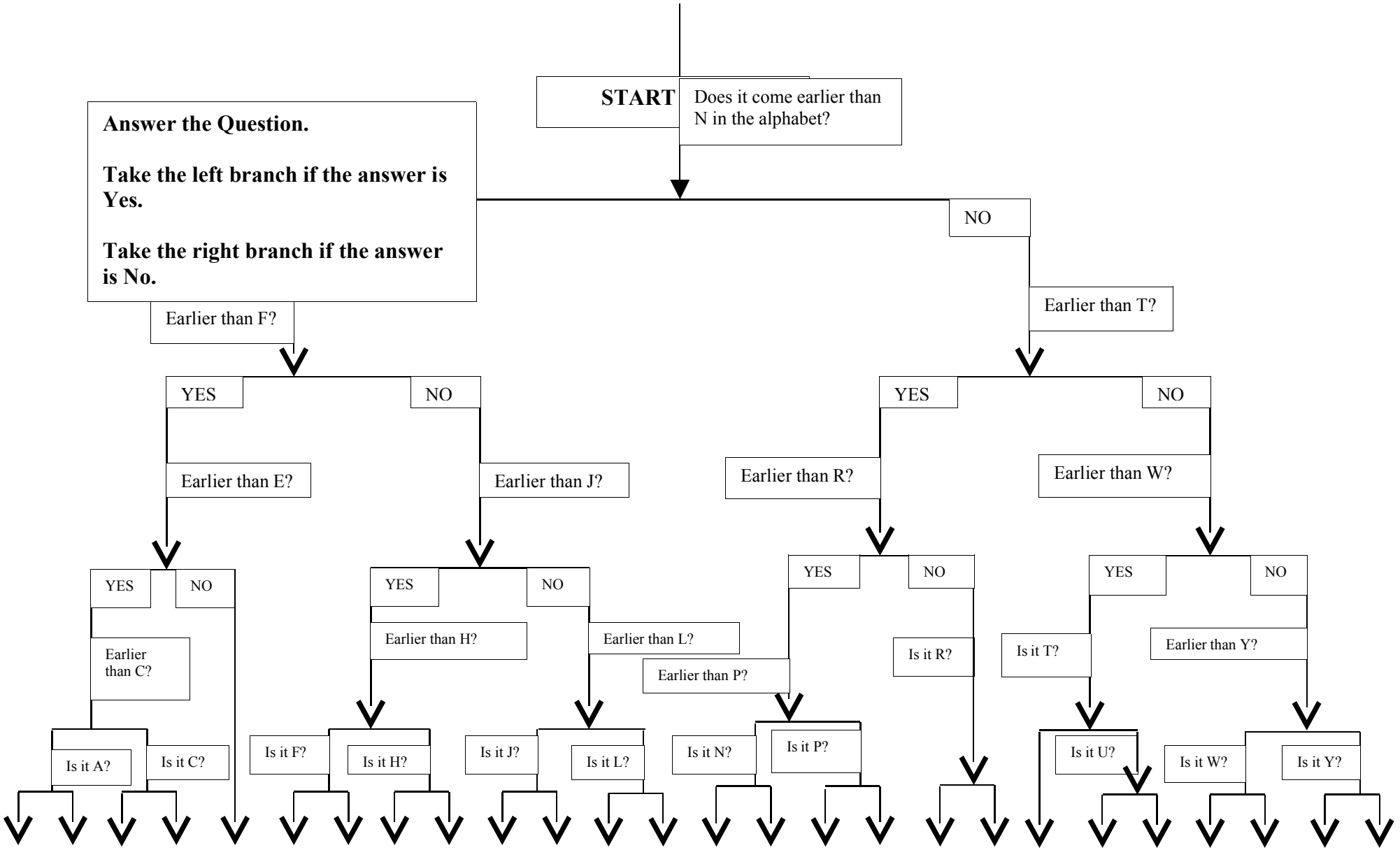


This tree could be optimised for letter frequency in the same way as Bauby did with his list. As it stands, infrequent letters like W and Q are at the top of the tree, and common ones like E, S, A and R are at the bottom so take more blinks to get to. By rearranging the order we could have common letters at the top. "Earlier" and "later" in the alphabet would now be different so Bauby would need to learn the tree himself so he could blink appropriately, rather than just knowing the normal alphabetic order.

In the above style of tree searching, the things we are looking for are stored all the way through the tree – in its nodes. At each stage we therefore have a choice of three possibilities: “it is here”, “it is down the left branch”, and “it is down the right

branch". The advantage of this is that it means some things are found immediately (the letter M in the above tree for example). However it comes at a cost: you need to ask 3-way questions with 3 possible answers instead of binary ones. Bauby would therefore need 3 signals such as no blink, 1 blink and 2 blinks to indicate which way to go. You therefore lose some of the advantage of reducing the questions asked by needing more blinks per question. An alternative is to just put binary YES/NO questions in the nodes and put the things being searched for at the ends of the branches: in the trees leaves (as in the search tree for identifying wood given above). Now a blink means go one way and no blink the other. However you have to descend to the leaves of the tree whatever the letter. With an alphabet of 32 letters you would always need 5 questions as the tree would be 5 layers deep. With our alphabet of only 26 letters, some of the branches can be shorter than this, however. This again gives us an opportunity to optimise the questions asked so that the common letters such as E can be found more quickly.

Search trees are, in one sense, the data structure equivalent to if-then-else statements. If you wish to make a decision based on information such as that for identifying timber, you could write an algorithm as a series of if-statements. However, such an algorithm can only make decisions about a fixed set of things such as trees. To make decisions about other kinds of object you would need to write a new set of instructions – a new algorithm. Alternatively, you could put the same information in a search tree data structure. Now the general tree searching instructions allow you to make decisions. To make decisions about a different kind of object you use the same algorithm but applied to a different search tree – containing different data. You do not need to learn and follow a new set of instructions, just follow the old instructions but on a different diagram.


START
Does it come earlier than N in the alphabet?
Answer the Question.
Take the left branch if the answer is Yes.
Take the right branch if the answer is No.
NO
Earlier than T?
Earlier than F?
YES
NO
YES
NO
Earlier than E?
Earlier than J?
Earlier than R?
Earlier than W?
YES
NO
YES
NO
YES
NO
YES
NO
Earlier than C?
Earlier than H?
Earlier than L?
Earlier than P?
Is it R?
Is it T?
Earlier than Y?
Is it A?
Is it C?
Is it F?
Is it H?
Is it J?
Is it L?
Is it N?
Is it P?
Is it U?
Is it W?
Is it Y?

Bucket Search

If we have plenty of space and are going to do lots of searching for the same things, a very fast way of searching is to do a **bucket search**. This involves pre-computing the answer to the question "is it there?" for each thing we might search for. We store the answers in an array with one entry for each thing we might search for (the array is effectively a **look-up table** containing answers to the search question). The array subscripts are thus labels naming each thing we might search for. In the simplest case the array entries are just YES or NO, indicating whether the thing was found or not. This is exactly what we were suggesting when taking binary search to the extreme and saying we want a way so that in one question we can be told the answer to our search question.

This is one of the most common ways that people organise appointments (meetings, parties, etc) they must keep. One way to do this would be to keep a list. Every time you found out about some event, you would add it to the list, noting the date. Finding if you have an appointment on some date would take a lot of time. You might try and keep this list sorted into order, but you would have to keep writing it out over and over as new appointments were added. Not surprisingly most people use a completely different way of organising appointments. We buy and use diaries and calanders. These are booklets with one entry for each day of the year (office diaries even have an entry marked out for every hour of the year). That is it has one entry for every possible day you could have an appointment, whether or not you do have appointments on that day. Initially all entries are empty, but as you are invited to parties or whatever, you fill in the appropriate entry. To discover if you have a party on a particular day, you simply go straight to that entry. It takes no time at all, because you do not need to scan through all the entries. Now a diary can actually be very wasteful – perhaps you are different but I do not have parties (or even other appointments) every day. Many of the entries in my diary remain blank. In a sense a diary is therefore a waste of paper. Putting appointments in one long list as I found out about them as suggested originally would use far less paper. This is the downside to being able to check for appointments quickly. The fact that diaries are so popular suggests that the trade-off is considered worthwhile in this case.

Consider the following memory game. A pack of cards are shuffled and you are given12 cards. You have 30 seconds to memorise those cards during which time you must place them face down (anywhere) on the table. You are then asked to turn up a given card, or state that it is not on the table. If you turn up the wrong card, or incorrectly say it is not on the table, you lose. One way to do well at this game, would be not to spend your 30 seconds memorising the cards, but instead spend the time organising them. Mentally divide the table into a series of card size slots with one row for each suit, Kings at one end, Aces at the other. Put the cards down on the table in place corresponding to their slot. You have just created a bucket array. To find a given card, you just look at its slot. If the slot is empty, the card is not on the table. If the slot is full, that is the card you are searching for. The skill of the game is then in visualising the array of cards rather than in remembering things. Organisation beats rote-learning any day!

When I was at University, my Hall of Residence used something similar so that people could tell whether I was in or out. At the entrance was a board with everyone's name on, with an IN/OUT sign partially covered by a slider. When I was out I would

leave OUT visible by my name, and when I returned I would slide it across to leave IN visible. This is acting like a bucket array, in that visitors did not need to do a full search – i.e. go all the way to my room to find whether I was in or out – the answer is pre-computed on the board. However, some extra work is required beforehand in that someone had to set up the board, and presumably put everyone's names to OUT before we arrived at the start of term. Also each time I went in or out, I had to do some work moving the slider.

Hash Tables

The trick to making bucket search fast is having a quick way of getting to the correct bucket. If you cannot go straight to the correct bucket, you have just replaced one search problem by another. If the buckets are numbered and in numerical order (ie the subscripts are numbers) it is relatively easy to go straight to the correct one. If the buckets are labelled some other way (for example by names) then you need a way of turning that label into a position quickly. Otherwise how do you find the right bucket! That is effectively what you are doing when you do a search – finding the correct position. The examples of lookup tables we saw earlier have this problem. When searching for a symbol in a map legend, we more or less have to search through each key of the table. We cannot work out from the symbol exactly where its position in the table will be.

Hash functions give a way of doing this. When a hash function is used in a lookup table, the table is known as a **hash table** rather than just a lookup table to show it is a special sort of lookup table. In the first instance the hash function is used to determine where to store things in the lookup table when constructing it (the pre-processing phase). You must put things in a place where you will then be able to find them. If you have read the Harry Potter books (if not why not?), then you have come across something very much like a hash function: the *sorting hat* (Rowling, 1997). Every year when the new batch of first years arrive at Hogwarts School of Witchcraft and Wizardry, the first thing that happens to them is that they are sorted into houses. Every pupil is in one of the houses: Gryffindor, Hufflepuff, Ravenclaw and Slytherin. Each house has its own tower with dormatories containing a bed for each pupil, so by organising the pupils into houses means (amongst other things) that they can be found when needed (eg to punish them). The four houses are thus like a lookup table with 4 entries – one for each house. Look in the Gryffindor tower and you will find all the students that were put there. The ceremony that allocates them is a pre-processing process. The pupils are processed one at a time. Each goes forward and puts on the "Sorting Hat". It is a magical hat that given a pupil can tell which House they should go in. It is like a **hash function**: a function that given a pupil works out where they should go. The pupil then goes to the house that the hash function (the "sorting hat") says they belong. The four house towers together are the **hash table**. As there are lots of students but only four houses, there are obviously lots of students in each house (unlike a straight lookup table where each entry has one thing in it). Thus having found the right house, a little more searching is needed some other way to find the right pupil. Normally hash functions are used in situations where there are more places to put things than things to put – eg more houses than there are pupils so that each pupil currently at the school gets their own house. For the "Sorting Hat" to be a proper hash function is would need to also be able to answer questions about its decisions. If one of the teachers (Professor McGonagall, say) asked it which house a pupil was in (say Harry) then it ought to be able to tell them. (I will not spoil the book

by saying where Harry Potter ended up, in case you somehow inexplicably have not read the book yet.) That way the Professors could use the hat to quickly find any pupil it had originally "sorted". We will see later that this kind of "sorting" is slightly different to what computer scientists usually call sorting (putting things into order, rather than putting them in the right place as here) – so do not get confused.

Thinking of hash tables as being like lookup tables where a number is needed top work out where to go. That is as though each Hogwarts house had a number (eg Gryffindor is House 1, and so on. Then the Sorting Hat would just give a number for each pupil. Knowing the number you would know which house to go to.

The sorting hat uses magic to determine where to look. Computers cannot do that. They do things by *calculation* instead. We are free to order the objects in a lookup table in any way we like. If they are placed appropriately we can often *calculate* the position to look at from the label. Address books do something similar. Addresses are put into slots in the book calculated from the first letter of the name. You then go straight to the correct page, assuming you know the order of the alphabet. Bookshops use a similar idea to help you find books. Instead of ordering books completely alphabetically, they are organised into shelves labelled by subject: Computing, Politics, Mathematics, Literature, etc. To find a particular book you first have to work out its subject, then go straight to that shelf. Map makers similarly try to help by grouping similar symbols together in the Legend, for example, however it can still take several seconds to find the correct symbol.

With both address books, bookshops and maps, the calculation does not take us to a unique book, just one of many that we must then sort through some other way. If you had a very large number of pages in your address book, it could be organised in a different way so that you could go straight to the correct place without needing much if any further searching. You would just need to do a slightly harder calculation to get there. Since there are in most search problems a massive number of possibilities, having a lookup table with one entry for each possibility is impractical. Imagine having an address book, with one pre-written entry for each possible name any human was called, with telephone numbers filled in only for the people you know. It would be enormous in size, so is just not done. We could come up with a compromise that was better than just labelling sections by the first letter.

For example, suppose you knew the positions of every letter in the alphabet off by heart (A=0,B=1, ... Z=25). The normal method used in address books effectively involves turning the first letter into a page number in this way. You could alternatively turn the first two letters into a position and have one page per pair. Assuming dictionary order is still used with BA following AZ, for example, the page would be calculated by turning each letter into its position in the alphabet as above then multiplying the first by 26 and adding the second. Thus, for example, AZ is converted to (**0** × 26) + **25** = 25 and BA is converted to to (**1** × 26) + **0** = 26. The address of someone called *Azar* would therefore be placed on page 25. How do you find the same person's address later? You just calculate the number 25 in the same way and go straight to that page. As long as two people you know do not have the same first two letters in their name, you will get a unique address. A function for calculating a position in this way is called a **hash function**, and a lookup table where the

information is accessed via a hash function is called a **hash table**. When two entries end up in the same place it is called a **hash collision**.

The above method uses up quite a lot of space. Many pages are likely to be blank as there will be many pairs of letters that are not the start of the name of anyone you know. Hash tables make a trade-off between search time and space wasted. At one extreme everything goes into one bucket (page), so you only need as much space as you have entries. However, you have to search through them all, which is slow. At the other extreme there is one page for each name that exists in the world. Then you can go immediately to the correct entry, but you are wasting a vast amount of unused space.

Using two letters is a compromise - there probably are many names that start the same way, so there will still have collisions where you have several names on one page, but hopefully you will only have a few entries to search. Since many members of my family have the name Curzon, however, the CU page in my address book would still be full. The calculation we used makes a poor hash function for names. We could avoid the problem by using a different hash function. For example, using initials together with the 3rd letter of the surname might be better. A good hash function is one that spreads the data evenly throughout the table, and so avoids clashes.

A hash table is thus a lookup table, where a calculation must be performed to find the correct entry, and where entries are shared by several different things, on the assumption that most of the time only one will be present. When a collision occurs, linear search is used to find the correct entry between those that have collided.

Summary
Searching is a commonly performed operation. There are many different algorithms for searching with different properties.

Linear search is the simplest search algorithm. It involves checking the elements being searched one at a time in order. It is, in general, a slow way of searching.

Binary search is a faster method of searching that uses the fact that the things being searched are sorted. We first check the central element, and decide whether the thing being searched is in the top or bottom half, then just search those elements in the same way.

Tree Searching first involves placing the things to be searched into a tree structure. A signpost is placed at each node indicating which part of the tree the things to be searched will be found – larger things are down one sub-tree, smaller ones down the other. The things being searched could be placed in all the nodes or just in the leaves of the tree.

Bucket Searching involves allocating a fixed position or "bucket" for each thing to be searched for, for example in a lookup table. The things are placed in their appropriate positions. Searching then involves going directly to the correct bucket. If the thing is there it is found, if not then it is not possessed.

Hash tables are lookup tables used for a form of bucket search where a calculation (given by a **hash function**) is performed to determine the place to look in the table. A hash table does not have one entry for each thing as would a full lookup table. Instead different elements make share the same location. This saves space as the table is smaller, but means **hash collisions** can occur, and so takes longer to find the element being searched for when this happens.