14. All Sorts of Everything (Sort Algorithms)

All things began in order, so shall they end and so shall they begin again.
Sir Thomas Browne, *The Garden of Cyr*us (1658) Ch. 5

We often need to sort things into some given order, and there are very many ways of doing it. The solitaire card game Patience is a sorting game. The rules are a partial algorithm for sorting cards. They are step by step instructions that if followed lead to the cards being sorted into order within suits. However, the rules of Solitaire are not all we might desire of an algorithm. In many rounds of Patience you will end up being stuck, with no rule giving you instructions of how to continue (You just lost). The rules are not **complete**. You can also have a game that never finishes in that if you follow the rules blindly you will never stop – though eventually you will get bored and give up. This happens when none of the cards in the discard pile can be placed. If you strictly follow the rules you will just keep checking the same sequence of cards over and over again, none of them with a place to go. The rules do not always **terminate**. The rules are not **finite**. Worse, the rules do not specify exactly what to do in each case – you have choices where two or more moves are possible – make the right one and you win ending up with a sorted pack, make the wrong one and the pack remains unsorted. The rules are **non-deterministic**. The rules do not tell you which to do because that is what makes it an interesting game. However, if the aim is not to have fun but to actually sort things, we need a proper algorithm that guarantees sorting every time.

Patience is also obviously quite a slow way to sort; ideally we want a fast algorithm. Sorting is relatively simple if there are only a few things to sort, however, as the number of items increases, the time taken increases disproportionately. By changing the sort method, big time savings can be made.

Bucket Sort
One of the simplest ways to sort things is to do a **bucket sort**. Each item to be sorted is allocated a bucket, and when you get to that item you toss it in the bucket. Once all the items are in a bucket, the buckets form a sorted list. My father sorts the large and random collection of nails, screws, washers, etc that he has acquired over the years in this way in his garage.

Imagine you have a shuffled pack of cards that you wish to sort. If you have enough space on a table or on the floor, you could just start laying out the cards in its appropriate position on the table. Mentally allocate separate rows for each suit, with the Ace to go at one end and the King at the other. Then just turn the cards up on at a time and slot it in its correct position. Once all the cards are laid down, scoop up the rows, put them back together and you have a sorted pack. You have just done a bucket sort. This is a very quick way of sorting as each card is looked at only once. It takes only as long to do as it takes to work through the pack once.

The post office has a similar sorting problem sorting letters. Thousands of letters arrive at the sorting office (a whole building dedicated to sorting!) as they are collected from post offices. They have to be sorted into piles corresponding to the

areas they are to be delivered to, so they can be put in appropriate vans. Airports have a similar problem sorting checked-in suitcases onto trolleys to be put onto the correct plane. When things go wrong, you end up in Florida, with your luggage in Cairo. What went wrong? Someone tossed your case into the wrong bucket!

This approach was also used during the original compilation of the Oxford English Dictionary in the 1800s. This was the first dictionary to aim to have entries for *every* word to have appeared in print in the English language. It was a monumental task that took 75 years to complete. The editors made use of thousands of volunteers who read books, making lists of all the words within them. The words found were then to be sent to the editors, who wrote definitions and compiled them. One of the volunteers stood out to the editors as being amazingly useful. His name was Dr William Minor and he was a convicted murderer who at the time was confined in Broadmoor Asylum for the Criminally Insane (Winchester, 1999). His being as mad as a Hatter did not stop him devising the most efficient way of working on the Dictionary that surpassed the approach taken by every other volunteer. His secret was to create an index of words (we will look at indexes later) and use bucket sorting to create the lists. The way Minor worked was to carefully read a book looking for interesting words. When he found one he would write it in his notebook. However, he did not just write it anywhere, but in a very precise position. As Winchester describes (Winchester, 1999, page 123), suppose he came across the word buffoon, he wrote it in small neat letters in a precise position towards the bottom of the page. The position was chosen based on the number of words he expected to encounter that should appear before or after it. When he later came across another word it would be written in a position estimated to leave enough room for the number of words that might come between them. In other words, he had mentally divided his notebook into a series of buckets, one for each word he would encounter. By the time he got to the end of the book he was reading, he had a sorted list of the interesting words in it.

Straight Selection Sort

As it is so quick, bucket sorting seems an ideal way to sort things until you consider the space problem. You need a separate bucket for each (class of) item you are sorting. With lots of things to sort this can be a problem. Suppose you were in the back of a car, squashed in the back seat with your brother and sister on a long journey and decided to sort that pack of cards. How would you do it now? You do not have the space anymore to do a full bucket sort.

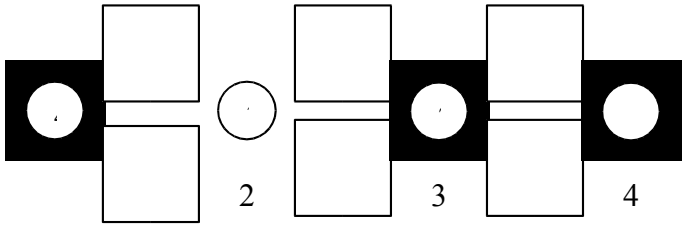
A simple approach known as **Straight Selection Sort** would be to scan through the pack until you found the Ace of Spades and put that to the front. Next you would look for the Two, and so on. Instead of just working through the pack once, you would have to scan through it 51 times since on each pass one more card is put in the correct position (Note it is not 52 passes since it is impossible to have all but one card in the wrong position). You would not need to look at the ones that were already in position each time, so every pass would be quicker than the pass before. Still, sorting just became very slow in comparison to the bucket sort, though you needed no extra space beyond the pack itself to do the sort

Bubblesort

The above way of sorting can get tricky with only two hands, trying to move a card from the back to its correct position without scattering half the pack over the floor.

The **Bubblesort** algorithm gets over this problem, by requiring that cards only ever be swapped with their neighbour. Work through the pack a card at a time, checking it is in the correct position relative to the next card. If the next card is the biggest leave them alone, otherwise swap them over. Move on to the next card and do the same again. Repeat this until you get to the end of the pack, then go to the front and start again. Do this over and over until the pack is sorted.

On each pass you will have noticed that the big cards bubble down the pack until they meet something bigger that then takes over. Once you get to the biggest card on the first pass (the King of Diamonds, assuming you were putting them in Hearts, Spades, Clubs, Diamonds order), it stays with you until the bottom of the pack. So after 1 pass, 1 card (the King Of Diamonds) is in the correct position. After 2 passes, the Queen of Diamonds will have done the same trick. Again each pass puts one more card in the correct place so by the end of 51 passes they will all be in the correct position. It thus takes a similar amount of time to the earlier algorithm, but with less likelihood of the cards spraying over the driver.
The following sorting puzzle can be solved using bubble sort. Four pieces numbered from one to four are placed in reverse order on the black squares of the following board as shown (so that piece 1 is placed on square 4, piece 2 is placed on square 2 and so on).



A move consists of moving a piece from one square to an adjacent square of the opposite colour. The aim is to finish with each piece on its own numbered black square in 24 moves with the following restriction:

- each piece can only stay on a white square for one move of another piece before moving on.

The white squares act as swapping points for adjacent pieces. The layout of the board and rules of the puzzle mean that the restrictions on swapping pieces are those used by bubble sort. These same restrictions mean that the other sorting algorithms described here cannot be used. The choice of algorithm is often restricted by the conditions under which it must operate in this way.

Binary Bucket Sort
You could still get part of the way bucket sorting if you had a little extra space. This is easier to describe if you have 32 cards rather than 52, so consider a pack with only the 1-8 cards present. Instead of having a bucket for each card, allocate two – red on one knee, black on the other. One pass through the pack has now sorted the pack into two reds versus blacks (so this is going to be much slower than a full bucket sort). Collecting the cards back up into a single pack, now allocate the knees differently: one Spades, the other Clubs. Repeat the process as you work through the black cards. When the blacks are done, collect them together in a single pile placed at the back of

the pack in your hand. Next, reallocate your knees with one knee Hearts, one Diamonds. When you gather the piles up, you must do so in the correct order.

The cards are nearer to being sorted than they were, as now all the cards of each suit are together after only two passes. What we have done is similar to binary search, we have thought of a question each time that divides the part of the pack we are looking at in two, then used that question to split them. We continue in the same way. We now have the pack in 4 suits of 8 cards (remember we are sorting a reduced pack to make it easier to describe). We need a question to split those groups of 8 in half. We ask whether the card is greater than 4, with one knee for those greater and one for those less. After each group of 8 has been split in two, put the two piles to the back of the pack in your hand (in the correct order). After we have gone through the whole pack once more, it will consist of a series of groups of 4. The cards in the 4 are still in the wrong order, but the group as a whole is in the correct position with respect to the other groups. Next do the same with each group of 4, this time asking whether the card is greater than the mid-card of the group (either the 2 or the 6 depending on the group). At the end of this you will have a series of groups of two cards, with those pairs possibly swapped, but the pairs themselves in the correct relative position. One more pass working on the groups of two and the pack is sorted.

As mentioned the approach used was very similar to binary search. We repeatedly broke the problem in half, giving a similar but simpler problem. Algorithms that work in this way are called **Divide-and-Conquer** Algorithms. Because of the divide and conquer approach we only need 5 passes through the pack instead of 31. If we had been sorting a full pack instead of a reduced pack, the complication would just have been that some of the groups would have been different sizes. It is easiest to do when the number of things being sorted is a power of two, because of the way we halve the pack each time.

Radix Sort

The above **binary bucket sort** is more or less what I do after every exam I am the examiner for. When the exam scripts come back from the exam hall, my first job as an examiner is to sort them into student number order. Since there can be up to 400 scripts, a full bucket sort is out of the question – I would need a sports hall floor to spread the scripts out onto. It would also take far too long to use a naive sort. After all I have only a week or so to both sort and mark them – the more time spent sorting the less time I would have to mark them. Instead I use the digits in the student numbers to form the buckets. The first 2 digits correspond to the year: in the 98/99 year the first set of piles correspond to 98, 97 and 96-and-earlier. Then each of those piles is split using the next digit, normally requiring 10 piles. These piles are then divided based on the next digit, one at a time and so on. Eventually the whole pile is sorted, using no more than 10 piles, plus a few others for the piles waiting to be split. I thus need nothing more than a large table to do the sort. Using successive digits as the buckets in this kind of repeated bucket sorting is known as **Radix sorting**.

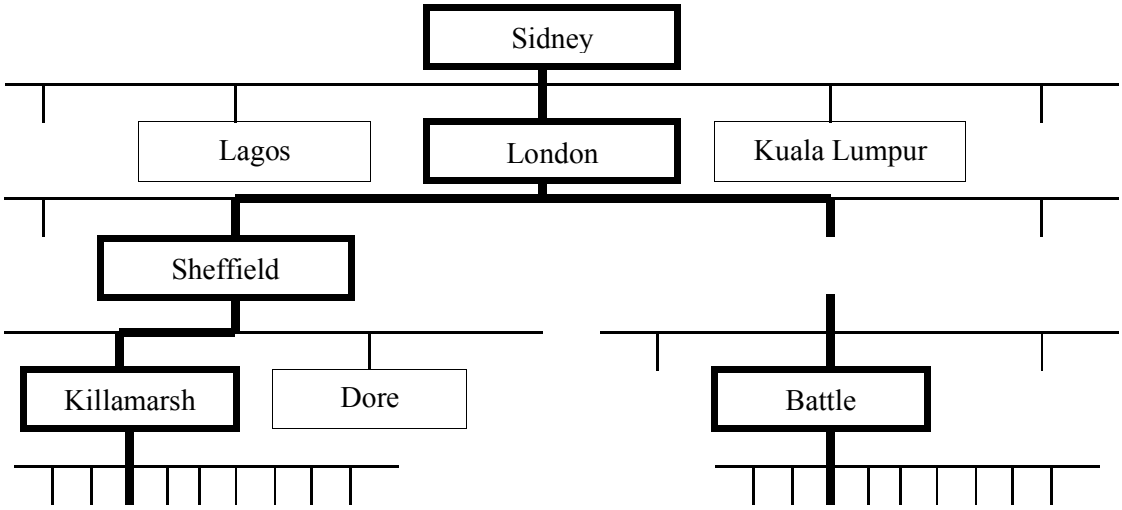
Tree Sorts

As we saw in the last chapter, tree searching is similar to binary searching. A tree structure is introduced and the questions asked during binary search are associated with a tree node. We can similarly use a tree structure to do a variation of bucket

sorting: **tree sorting**. Each question that is asked is associated with a node of a tree, and instead of adding the things being sorted to piles, they are sent down one of the edges of the tree, towards the leaves.

This is more or less what the Post office is doing when they deliver letters. Sorting offices are joined together in a tree structures. The sorting offices are nodes, the links provided by planes, trains and vans. They do not sort all the letters in a single sorting office, but ask a single question: Which country should this letter go to? Letters are then sent along the appropriate link (i.e. put on the appropriate plane) to get to the main sorting office in that country. At that sorting office they ask: What city should this letter go to, and they are sent on their way by train to the next node in the tree. At the city sorting office, they split the letters again by postal district and put them in vans that take them to the local sorting offices, where the postman picks them up to deliver them to the leaves of the tree: our homes.

Suppose, when on holiday in Sydney, I post two postcards one to *Killamarsh*, a village near Sheffield, the other to *Battle*, a village near Hastings. They will travel together on a plane to the London sorting office, then be sent down different branches, one to Sheffield, the other to Hastings. From there they will travel to local post offices, to be delivered to the appropriate homes.

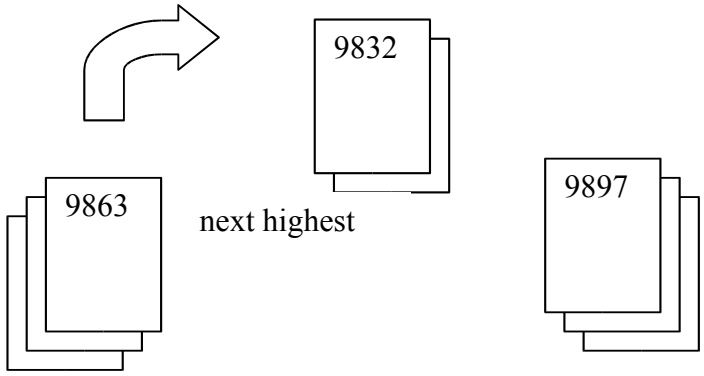


There are several different tree-sorting algorithms. In the post office one above, the things being sorted are placed at the leaves of an existing tree. In the variations, the things being sorted are stored in the nodes, and the tree is built as the items are put into it. This is similar to the post office having a strategy of only building a sorting office in a place the first time someone sends a letter to that city! In these alternative algorithms, sorting is done by putting things into the tree one by one according to some rule, then taking them out in an appropriate way again one by one, so that they come out in sorted order. We will not go into details here

Merge sorts

When sorting exam scripts I usually have someone to help. Ideally we wish to divide the task up, so that we can both work independently and are never idle. **Merge sorting** is one way to do this. We just split the unsorted pile in half, and sort them completely independently, to give two sorted piles. We could even sort our own piles using

different algorithms if we wished. Once we both have a fully sorted pile. We need a way of putting them back together. If we just put one pile on top of the other, we would not be much better off. Instead we **merge** the two piles into a third. The merge algorithm is as follows. We each look at the script on top of our pile. Whichever is largest is placed first on the new pile. We repeat this again only looking at the top two scripts until one of the piles has gone. The remainder of the other pile is then added one script at a time onto the new pile. This works because, since the two piles are sorted, the next one must be on top of one of the two piles, so we only ever need look at the top scripts.



In the above we only did a single merge. If we had 4 people to do the sort, we could split the pile in 4 and each sort one pile. We could then merge each pair of piles together to give 2 sorted piles. They could then be merged as above. With enough helpers, each person could be given a pair of scripts to put in the right order. Then the piles are gradually merged together, pairs of piles at a time.

This idea of repeatedly splitting and merging can be extended to the whole pile to give a sort algorithm called **mergesort**. Lets go back to sorting the packs of cards in the car. Suppose there is even less space. This time you only have one free knee (your baby sister is asleep on the other knee). Go through the pack two cards at a time, merging them into order in their pairs, using your free knee to create the new pile. Once you have gone through the whole pack, start again, taking adjacent pairs (one pair in each hand) and merge the pairs into sorted groups of 4 cards, putting them onto a pile on your knee. Keep doing this for turning groups of 4 sorted cards into groups of 8 and then groups of 16 until the whole pack is sorted. The whole sort was achieved by doing nothing but merging! You just used another divide-and-conquer algorithm. Think of what you did backwards. The last thing was to merge two halves of the pack. How were these sorted? – by splitting them into two piles, sorting and merging them back into order. You were repeatedly breaking the pack in two. You used a different divide-and-conquer algorithm though as you did different things with the two halves, and put them back together in a different way.

Multiple views and Indexes

Lists are often ordered in a specific way to make searching or other processing easy: alphabetically by name as in a telephone directory, by record sales as in the Top 40 singles list, by order of arrival as in the queue at a supermarket. This is fine if the lists are small or if that is the only order we wish to access them. Often, however, we really want to access the same list in different ways at different times. Suppose the phone rings, but the person hangs up before you get to it. In Britain you can ring 1471 to find the number of the person who just called. However, if you do not recognise the number, it might just have been a telesales company and the last thing you want to do is phone them back. I have also had people phone me back late at night after I dialled a wrong number. The resulting conversation was very surreal. The phone book is in alphabetical order because it is intended that you look up the numbers of known people. It would be nice in our situation above if the phone book were also in number order. You could then easily find the name of the person who called. As it is finding the name is near impossible! In fact BT rely on this as they are not allowed to give out information that would allow a name to be derived from a number to protect the privacy of the customers.

In other situations, it is even more desirable to get at the elements of a list in different orders. Railway stations use multiple views of timetable information. The full timetable is usually given in order of destination, since that is the information that you usually know about the train you wish to catch. On the screens as you enter the station, timetable information is given in order of departure time however, since people are likely to arrive shortly before their train departs. Such a list is therefore the quickest way to find the data they want.

In the above case the data is copied out again in full. This does not need to be so, however. In other situations, only one full copy of the data is kept to save space. **Indexes** are used for this purpose. An index is just an alternative ordering of the same information (or at least part of it) created by giving pointers into the original. A book, for example, is just a list of words in a particular (literary) order. However, in textbooks, it is useful to be able to access the text in other orders: looking up a given word. For example, in a book on Data Structures, you may wish to read about trees. Rather than read the whole book, you just wish to read that section. The index provides you with a quick way of finding it. It orders the main words of the book in a different order to the original (alphabetically). The index gives references to the places where each word occurs in the normal order.

A *concordance* is a whole book that is just an index of another book – usually the Bible. If you want to know about all the passages in the Bible where the word *plague* appears, just look in a concordance and it will tell you. The academics working on the Dead Sea Scrolls (ancient fragments of biblical texts that were painstakingly pieced back together) did not wish to give out copies of the works, as they wanted exclusive rights to study them. As a compromise, however, they did release a detailed concordance. However, since a concordance is just a list of the same words in a different order with references to the correct order, some other academics took the concordance and reconstructed the original order of words from it. They were happy to release the texts to the world and did so!

Travel agents have a similar queuing problem to Banks. However, they do not always make their customers stand in a queue. The people seem to be sat in a completely random order. An index is being used to maintain two orders – the order of arrival and the order of seating. Each person takes a ticket with his or her arrival position on. This means they can sit in any free chair and do not need to keep moving. The ticket numbers are providing an index. Cheese counters in large supermarkets often use a similar system as it is too difficult to make people stand in an orderly queue in this situation – they want to be able to wander up and down the counter deciding what to by whilst waiting.

When I get my hair cut a similar problem occurs. There are four seats for those waiting and people sit in them at random as they come in. When the next person's turn comes we do not all move up a seat but instead stay put. Each person knows how many people were in front of them when they arrived and remember this number. Each of the other people are similarly remembering their positions. When the next person's turn arises they know because the are thinking "I'm now first". As they move to the Barber's chair all the others mentally subtract one from their number. In effect there are two arrays. There is an array of people that stays unchanged other than when a person arrives or leaves. There is also a second array of the numbers (that bears no relation to the chair positions) in the heads of the people in the queue. This is in essence an index array. As with other index arrays it is being used to avoid moving something big around: the people. Instead the easily exchanged numbers are juggled in the peoples heads leading to less effort for all.

You do not have to restrict yourself to one index. Libraries keep several different indexes to their collections of books. The University library I used stored books on the shelves in the order that they were acquired. This was sensible as space was short and manpower limited, so books were just added to the end of the next free shelf, labelled with an acquisition number, so they could be returned to the correct position. However, as a student I usually knew the author of the book I wanted, not its acquisition number, so finding it by browsing the stacks would be near impossible. The library provided a filing drawer by author – a set of index cards in alphabetical order by author giving acquisition numbers of each book. At other times I knew the title but not the author. The librarian was well prepared, however, as a second index was in order of title. And just in case a student knew neither an author nor a title, there was a third index in order of subject. Nowadays, the index cards have been replaced by computers, but those computers have to do the same thing – keep lots of index lists.

The word lists that the insane dictionary writer William Minor created were also indexes. It was this that made him the most useful volunteer to the editors. One of the special things about the Oxford English Dictionary is that every word has quotations from the literature to illustrate its meanings. The editors had asked the contributors, on finding a word, to write it on a slip of paper, together with a full reference of the source and a quotation of the sentence in which the word was found. These slips were then to be sent in to the editors to compile. This led to the editors having a paper mountain of thousands of word slips to sort many of which duplicated ones they already had, and many that would not be needed for years given how slow the progress of compiling every word in the language was. Minor worked differently. For the first few years of work he sent in no words at all. He just compiled his lists of

words. However, each time he wrote a word in his notebook, he also noted the page number it was found on. In other words, he created an index for every book he read. He then told the editors of the dictionary to let him know the word whose entry they were currently compiling whenever they were short of a quotation. He would then look up the word in his indexes, and if he had ever encountered it, which was most of the time, he sent in a quotation by return. Because of his carefully compiled indexes, he could provide words and quotations exactly when the editors required them. He also only needed to copy out quotations and full details of sources of words that were actually to be used in the Dictionary, unlike the other contributors much of whose efforts would ultimately never be used. Insane or not, Minor had a firm grasp of the power of choosing appropriate data structures and algorithms for a task.

Summary
Sorting is another common operation we perform. As with searching there are many different sort algorithms.

Bucket sorting involves having a fixed position for each element to be sorted. The things to be sorted are processed in turn, placing each in its correct position. The things are then just collected in order of position – sorted. This can be very fast, but usually impractical due to the amount of space required.

There are various ways of sorting naively – ie using very slow algorithms. One involves scanning through the things in a series of passes from start to finish, each time looking for the next thing in order and putting it in its correct position.

Bubble sort is similar to the above naive sort. However, on each pass adjacent things are swapped if out of order relative to each other, so that many of the elements move towards the correct position on each pass. On each pass the next biggest thing automatically will end up in its correct position, so does not need to be examined on subsequent passes.

Binary Bucket sorting involves partitioning the objects into two buckets so that the two halves are in the correct order relative to each other. Those halves are recursively sorted in the same way, until the halves consist of single things that require no sorting. **Quicksort** is another algorithm that works on a similar principle, but uses a slightly different partitioning algorithm.

Radix Sorting is similar to binary bucket sorting except that 10 buckets are used in each round corresponding to the digits in the numbers being sorted. In each round the sort is performed on the next digit.

Tree sorting algorithms do the sorting by placing the elements to be sorted into a tree structure as used for searching. By removing the elements from the tree in an appropriate way, they come out sorted. There are many variations.

Merge sorting is similar to binary bucket sorting in that it uses a divide-and-conquer approach. The things to be sorted are split in to and sorted separately (by recursively doing the same thing). Once the two halves have been sorted, they are recombined by merging – using the principle that the next thing in the sorted sequence will be the next in one or other of the two halves.

Often we wish to have a set of things in many different orders at the same time. An **index** is used for this purpose. By having multiple indexes we can have multiple orders without changing the actual order of the things of interest.