CM50117: Algorithms and Data Structures

Russell Bradford

1 Web Page

`http://www.bath.ac.uk/~masrjb/CourseNotes/cm50177.html`

2 Why study algorithms?

This course is about how to select the best tool for the job. When you have a problem to solve there may be many ways to do it

- which is the the "best"
- how to choose
- and what do we mean by "best"?

It is important not to implement the first thing that comes into your head. Different algorithms can have different orders of magnitude of speed. You may be able to solve a small problem with an "obvious" method, but which can completely fail for a large (production) problem.

Some algorithms are very complicated, very non-intuitive, very clever and very non-obvious. We shall study simple ones: sorting and searching. We shall be sorting and searching mostly numbers, but these apply to strings, or any other kind of data object.

These techniques contain enough to point out the essentials.

2.1 . . . and data structures?

As above. Sometimes forgotten, but can be equally important in an implementation. Some datastructures you know: arrays, probably. Others are much more flexible: lists, trees and hash tables.

> Some languages give us these advanced datastructures as a feature, like C gives us arrays as a feature. In C, though, we have to build them ourselves.
>
> "Advanced"? Lisp gave us lists in the late 1950s, barely younger than arrays.

2.2 Best?

We should select the algorithm best suited for *the current task*. For example, quicksort is usually better than bubblesort (see later). So we usually use quicksort, except when we are in one of the (rare) situations that bubble is better. Sometimes we don't know in advance: we could just hope for the best, or apply a bit of intelligence.

2.3 Measures

And when we are choosing the best algorithm there are many criteria to consider.

- Speed. Often the criterion of choice.
- Size. Memory usage can be important on small machines. Or on big machines if the data are truly enormous.
- Cost. E.g., the varying costs of transport on a computer network.
- Energy use. Some interplanetary probes must use as little energy as possibly, even if it means running programs really slowly.
- And so on.

2.4 Comparison

"My algorithm is better than yours."

- In what circumstances?
- What measures are you using?
- What machine/language/implementation, etc.?

It is best to compare algorithms in the *abstract*, and assign some mathematical measure. Then we can be assured of the behaviour of an algorithm: it's not just that it was implemented by a bad programmer, it really is slow.

Beware someone who says "Quicksort is the best sorting algorithm". They don't really understand the complexity of the problem.

2.5 Complexity

Complexity is the way we can tell whether one method is better than another. We get a measure of how long an algorithm will take to get the answer.

We seek a measure of the size of a problem: e.g., number of items to sort, numbers of items to search, but this can be any other measure we wish. We need to know (a) how long it will take to solve, and (b) how much memory the solution will need. It's good to have an estimate of these, the *time complexity* and the *space complexity*.

A simplifying asumption we shall usually make is the size of the problem is *large*. We shall see that the behaviour of lots of algorithms on small datasets is muddied by all sorts of irrelevant details: only when we have large datasets does the behaviour become clear and understandable.

Most of the time, we *do* have lots of data, so this assumption is fine. Sometimes, though, we have to consider the case of small datasets and so our analyses will not be valid for these particular cases. These are pretty much done case-by-case.

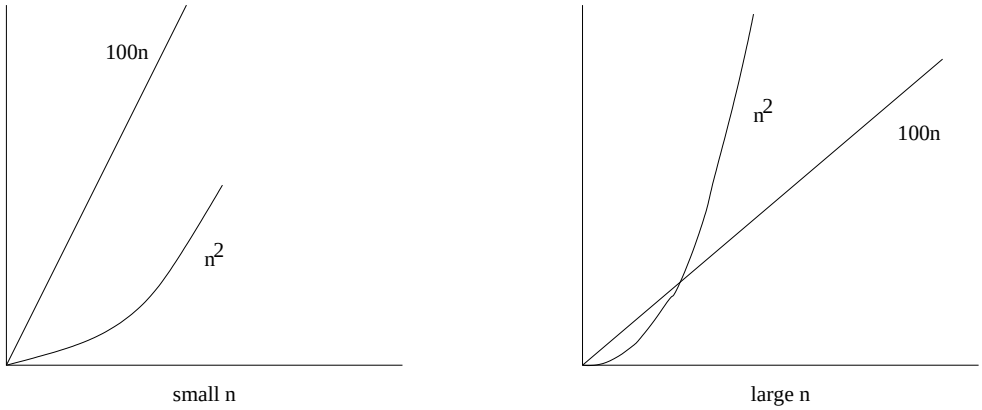

Figure 1: Quadratic and Linear

2.6 Basic Arithmetic

We need some basic vocabulary to describe the behaviours. The basic idea is that we have the size of the problem (n), and we want to determine how long it takes to run/how much memory it takes/etc., and how that depends on n.

- Constant. a
- Linear. $an + b$, constants a and b, e.g., $100n + 12345$
- Quadratic. $an^2 + bn + c$
- Polynomial. $an^r + bn^{r-1} + \cdots + s$
- Exponential. $a2^n$ or ae^n
- Logarithmic. $a \log n$
- Various. e^{e^n}. $n \log n$. $n^2 e^n + \log n$. In the last, the $\log$ term doesn't make much contribution to the time (or space, or whatever): the important bit is the $n^2 e^n$.

2.7 Big O

Suppose we find two algorithms that take times n^2 and $100n$ to run respectively. At first, the n^2 seems better, but the square term is eventually going to dominate:

n	$100n$	n^2
1	100	1
2	200	4
3	300	9
50	5000	2500
100	$10,000$	$10,000$
1000	$100,000$	$1,000,000$
$1,000,000$	$100,000,000$	$1,000,000,000,000$

That's a quadrillion. So the $100n$ algorithm is actually the better one (for large datasets).

Next, suppose we have an algorithm that takes time $n^2 + 100n$. For $n = 1000000$, the $100n$ is 100 million, which is just $1/100\%$ of a quadrillion.

So when we are talking about big values for n we need only think about the quadratic term as that forms the overwhelming part of the time. The linear term is important for small n, but we are really concerned with big n. We can simplify by ignoring the small stuff.

We say the problem has complexity $O(n^2)$. This notation means that

- the problem grows like $an^2 +$ stuff where a is some constant, and "stuff" grows slower than n^2, and so will be minuscule relative to n^2 for large n
- this statement is valid for all large enough values of n
- this statement says nothing about small n, or what "large enough" is supposed to mean
- we're not too concerned as to whether it is $100n^2 +$ stuff or $2n^2 +$ stuff
- when the problem doubles in size, the problem takes 4 times (roughly) longer to run as $(2n)^2/n^2 = 4$; if it triples in size, it takes 9 times as long, and so on.

The last is the important bit: if a problem takes 10s to run, then a problem twice the size will take about 40s; one 10 times the size will take 1000s.

The important part is how the time grows with n. All sorts of factors determine the actual time a program takes to run: the speed of the computer, how good the compiler of the language is, whether the processor has a good multiply operation, and so on. It is much more useful to compare relative times for various sizes of problem, i.e., n.

If the complexity is $O(2^n)$, doubling the size increases the time taken by much more than 4 times: $2^{2n}/2^n = 2^n$, meaning if a problem of size $n = 10$ is doubled it takes $2^{10} = 1024$ times as long. If a problem of size $n = 100$ is doubled it takes $2^{100} \approx 10^{30}$ times as long.

It's best to avoid solutions that take exponential time, but some problems don't appear to have faster solutions. For example the Travelling Salesman. Ditto factoring large integers: the security of the Internet relies on this! Note that just because they don't *appear* to have fast solutions doesn't mean they don't! New solutions to old problems are being found all the time.

It's good to find polynomial time solutions; the smaller the degree the better. Linear is nice, logarithmic is better.

2.8 Technically

We say $f(n) = O(g(n))$ if

- there is a positive constant c
- there is an N

such that whenever $n > N$ we have

$$|f(n)| < cg(n).$$

Notice that it doesn't matter (to the definition) what c and N are, only that they exist. The value c is called the ***hidden constant***.

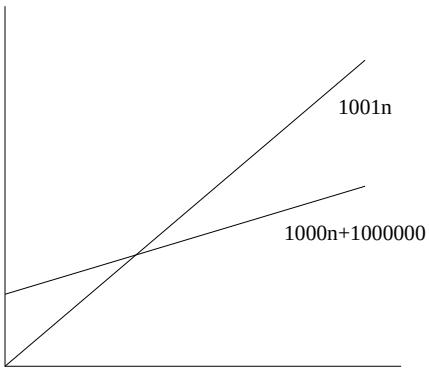

Figure 2: Linear

The O notation is a little strange, but turns out to be really useful. Even though you say "blah $= O$(blah)" it is really "blah$<$ blah".

Examples.

- $f(n) = 1000n + 1000000$. Then $f(n) = O(n)$, since for $n > 1000000$ we have $f(n) < 1001n$.
- $f(n) = 1000n + 1000000$. Then $f(n) = O(n^2)$, since for $n > 1618$ we have $f(n) < n^2$. Or for $n > 1$ we have $f(n) < 10000000n^2$. Either will do. Saying $f(n) = O(n^2)$ is a bit weak, as we know a better bound for f, namely $O(n)$.
- $f(n) = n^2$. Then $f(n) \neq O(n)$. For suppose $n^2 < cn$ for all $n > N$, for some c and N. But $n^2 < cn$ implies $n < c$, where c is constant, so n can't be big.
- $f(n) = 1000n + 1000000$. Then $f(n) \neq O((\log n)^r)$ for *any* power r.
- $f(n) = 2n^2 + n + 1$. Then $f(n) = O(n^2)$. I ought to find an N and a c, but I don't really need to because I can see the n^2 is the dominant term.
- $f(n) = 2n^2 + n + 1$. Then $f(n) = O(2n^2 + n)$. This is just a bit more precise than $O(n^2)$. Precision is occasionally useful, though not often.
- $f(n) = 2^n + 1000000n^{1000000}$. Then $f(n) = O(2^n)$. Again I know that exponentials always win over any power.
- $f(n) = 2^{n^2+n+1}$. Then $f(n) \neq O(2^{n^2})$. Unfortunately, it's not so simple as that.

There is a hierarchy:

$$\text{constant} < \text{logarithmic} < 1/r\text{th power} < \text{linear} < \text{quadratic} < r\text{th power} < \text{exponential}$$

where "$<$" means "is dominated by". Note there are many more speeds of growth than this: faster are e^{e^n}, $e^{e^{e^n}}$, and so on. Slower are $\log\log n$, $\log\log\log n$, and so on. And there intermediates as well: for example $n \log n$ lies between n and n^2.

In fact, any constant multiple of $n \log n$ is eventually dominated by $n^{1.5}$, $n^{1.01}$, $n^{1.00000001}$, and indeed *any* power of n bigger than 1.

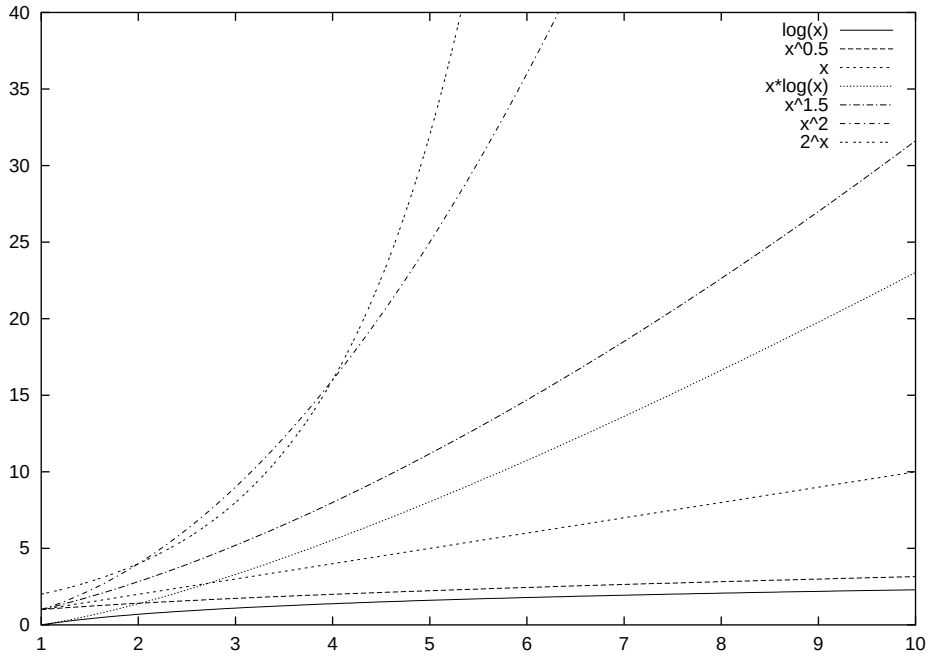

Figure 3: Orders of Growth

n	n^2	$\log n$	$n \log n$	2^n
2	4	1	2	4
5	25	2.322	11.610	32
10	100	3.322	33.22	1024
100	10000	6.644	664.38	1.27×10^{30}
1000	10^6	9.966	9966	1.07×10^{301}
10^6	10^{12}	19.932	1.993×10^7	9.90×10^{301029}

Mathematics tells us there are functions that grow faster than *any* depth of nested exponentials (e.g., $f(0) = 1$, $f(1) = 2$, $f(2) = 2^2$, $f(3) = 2^{2^2}$, $f(4) = 2^{2^{2^2}}$, and so on) and other functions that grow slower than any depth of nested logarithms.

Suppose a problem that has two parts, the first takes time $O(n)$, the second $O(n^2)$. Taken together, the whole problem has complexity $O(n) + O(n^2) = O(n^2)$, as the linear part can be disregarded for large n.

Note this does *not* imply that $O(n) = 0$! It means that $c_1 n + c_2 n^2 < cn^2$ for some c and large n.

Similarly, if a problem has two parts, both taking time $O(n)$, then the total is $O(n) + O(n) = O(n)$.

Which means $c_1 n + c_2 n < cn$ for some c and large n.

A problem of complexity $O(1)$ is one that takes no more than a constant time to solve, regardless of its size. Example: to retrieve an element from an array takes $O(1)$ time. Again, that the constant time could be 1ms or 1 million years: the important point is it is independent of n. Here the hidden constant can be very important.

Confusingly, we say a function is "slow" if it grows slowly (like $\log$). When an algorithm takes time that is a slow function, it is *fast*, i.e., the time it takes to run increases slowly.

2.9 Cases

Algorithms are measured in many ways, but some popular cases are

- Best case. What is the best possible situation for this algorithm? Data are allowed to be in just the right values and in just the right places to make the algorithm perform at its best.
- Average case. What happens in the average case? This should be the behaviour we would expect to see normally with data that are in no particular configuration.
- Worst case. What is the worst that can happen? When the data happen to be in just the wrong configuration.
- Common case. Is there something special we know about the data we can use to our advantage? For example, nearly sorted data. Ideally, we want the common case to be the the same as the best case.

Average case is what we normally expect to see in the absence of any special information, though we must keep an eye on worst case. If an algorithm takes 10 minutes on average but 10 years in the worst case we might think twice about using it.

2.10 Other Complexity Measures

There are other notions: particularly big omega Ω to mean "at least", where O means "at most". If something is $\Omega(n^2)$ it takes at least n^2 time (or space or whatever), possibly more.

Technically: if there exists a positive constant c and an integer N such that $f(n) > cg(n)$ for all $n > N$, we say

$$f(n) = \Omega(g(n))$$

Examples. $2n^2 + n + 1 = \Omega(n)$. $2n^2 + n + 1 = \Omega(n^2)$. $2n^2 + n + 1 \neq \Omega(n^3)$.

While O and Ω are upper and lower bounds of growth, we also have Θ to give a more precise estimate.

If there exists a positive constants c_1 and c_2 and an integer N such that $c_1 g(n) < f(n) < c_2 g(n)$ for all $n > N$, we say

$$f(n) = \Theta(g(n))$$

Now Θ says g is a good estimate for f, neither too big nor too small. If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ then $f(n) = \Theta(g(n))$.

Examples. $2n^2 + n + 1 \neq \Theta(n)$. $2n^2 + n + 1 = \Omega(n^2)$. $2n^2 + n + 1 \neq \Omega(n^3)$.

The most important measure is O. The others are used less as typically they are harder to compute and use.

2.11 Conclusion

If you want to get solutions to ever larger problems you can

- get a faster computer, or
- find a solution method with a better complexity.

Figure 4: Complexity measures

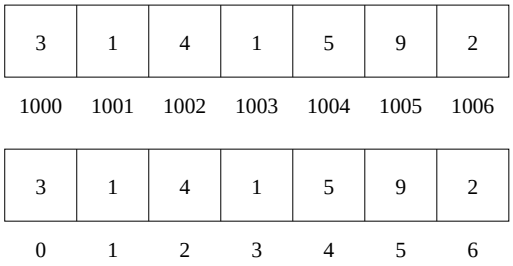

Figure 5: Array

The second may be harder, but will always win in the long run.

Algorithms with better complexity than previously known are still being discovered. In 2002 a significant new algorithm to determine whether a number is prime was discovered. It's importance is that its complexity is polynomial ($O(n^{\text{something}})$) rather than $n^{O(\log \log n)}$ which was the best previously known.

3 Data Structures

Abstract operations: insert, delete, search and retrieve.

3.1 Array

You are familiar with arrays. You reference an object in the by a integer, its index. Advantages are fast to insert and retrieve a value: $O(1)$ time access; arrays are compact, they take little more space than needed to store the data. Disadvantages are fixed size, slow to search without extra help.

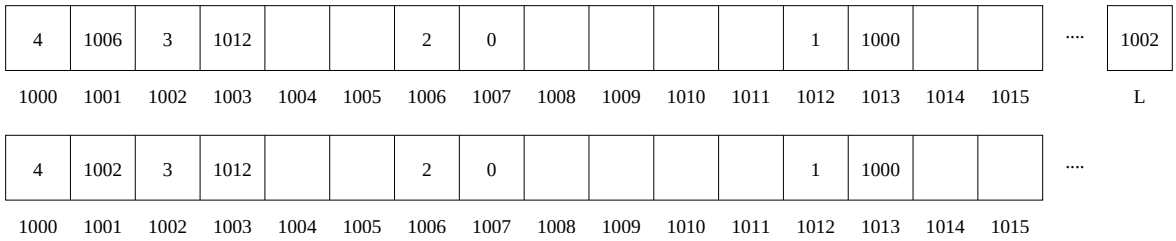

Figure 6: Linked List

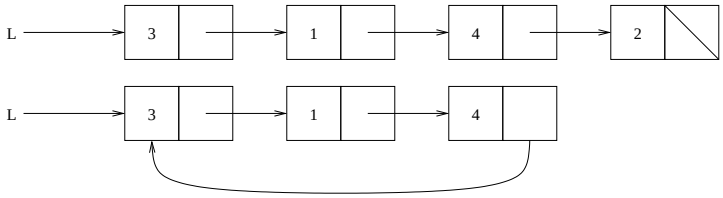

Figure 7: A common way to draw lists

3.2 Linked List

Linked lists are as old as computers. Unlike arrays, which are a fixed size, lists are *dynamic*. They can grow in length and can shrink and be rearranged and stuck together and taken apart.

A list *node* contains two things

- a value (or values), the datum we are trying to store, and
- a pointer to the next thing in the list

The last node in the list (if there is one…) contains a NULL pointer. This is a distinguished value that we know is not a pointer so it marks the end of the list. In C (and many other languages) the value 0 is conventionally taken to be the value of NULL. A list is a sequence of zero or more nodes, each pointing at the next.

A useful way of thinking about lists is

> a list is empty, or a node on the front of a list

This recursive view leads the way to many simple algorithms that process lists.

Function **printlist**:

- if the list is empty, return
- print the node value
- get the next node, and print it using **printlist**.

Example. The “3 1 4 2” list above prints “3 1 4 2”.

Example. The “3 1 4” loop list above prints “3 1 4 3 1 4 3 1 4 …” forever. A more clever version of **printlist** is needed!

A list can be traversed backwards with little extra effort: Function **printlistrev**:

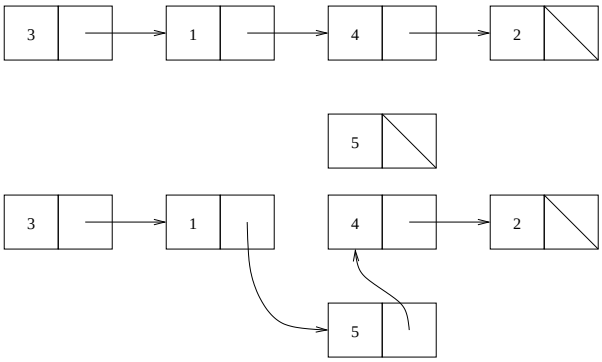

Figure 8: Adding nodes

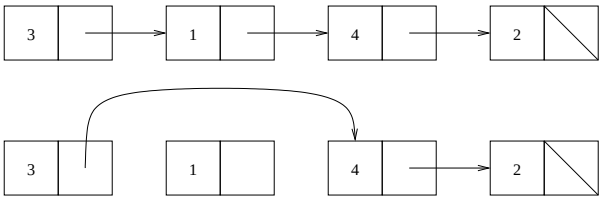

Figure 9: Removing nodes

- if the list is empty, return
- get the next node, and print it using **printlist**.
- print the node value

Using this function, the "3 1 4 2" list prints as "2 4 1 3", while the loopy "3 1 4" prints nothing at all. It is trying to print the last thing in an infinite list!

We can splice new nodes into the list at the start, the end, or in the middle. It's all a matter of setting the next pointers successfully. Similarly, two lists can be joined together.

Alternatively, nodes can be cut out of lists, and lists can be divided into separate lists.

At this point is is worth noting that in C (unlike some other languages) memory management must be done by the programmer. This means when a new node is needed the programmer has to choose where in memory it should be placed. Usually, the C function `malloc` is used to keep track of free and used memory. Similarly, when a node is removed from a list the programmer must explicitly notify `malloc` the memory is now reusable by calling the function `free`.

Function **findlist**: accessing a value in a list

- if the list is empty, return "not found"
- if the node value is the one we want, return it
- look, using **findlist**, in the rest of the list.

This is a $O(n)$ operation on average. Best case $O(1)$, worst case $O(n)$.

Many variants of simple linked lists exist.

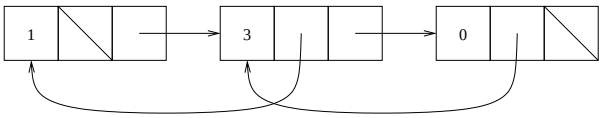

Figure 10: Doubly Linked List

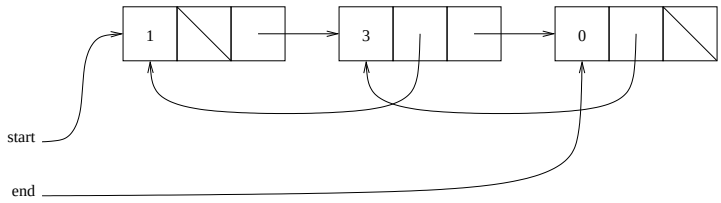

Figure 11: Pointer to end

- Doubly linked lists. We have two links, one forward and one back. This allows us to run both ways along a list. These are occasionally very useful, but a lot of hassle to manipulate all the pointers whenever we change the list. Usually, they are not worth the effort: doubly-linked code can often be made simpler and easier by rethinking in terms of singly-linked lists.
- Pointers to both ends. If getting to the end of the list is a frequent need we can keep pointers to both ends.
- Pointers to every mth element. If we are looking for values we know are deep within the list this allows us to jump directly to just before what we are looking for and then do a short linear search.

Stacks and queues are built from lists.

```
struct intlist {
  int val;
  struct intlist *next;
};

struct doubleintlist {
  int val;
  struct doubleintlist *next, *prev;
};
```

Lists are very flexible and extremely useful. There are whole languages built on the idea of lists as a primitive. Lisp (1959–present) is such a language. Lists are perhaps the second oldest composite datastructure after arrays.

In languages like C they are fiddly to program (watch out for the edge cases!), and accurate memory management is needed (`malloc` and `free`).

3.3 Tree

Trees are a natural extension of linked lists. Instead of one “next” node there are now two. These are *binary trees*. Of course, other kinds of trees can have more *children* nodes, but we shall concentrate on binary trees.

Now we see that

> a tree is empty, or a node on the front of two trees

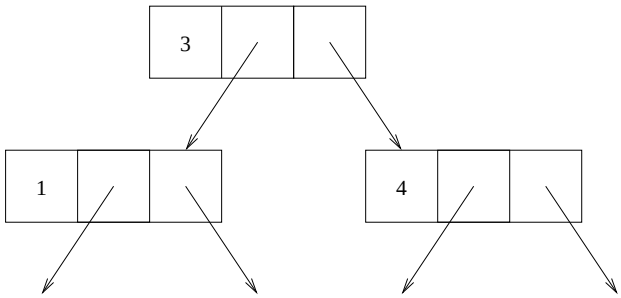

Figure 12: A tree

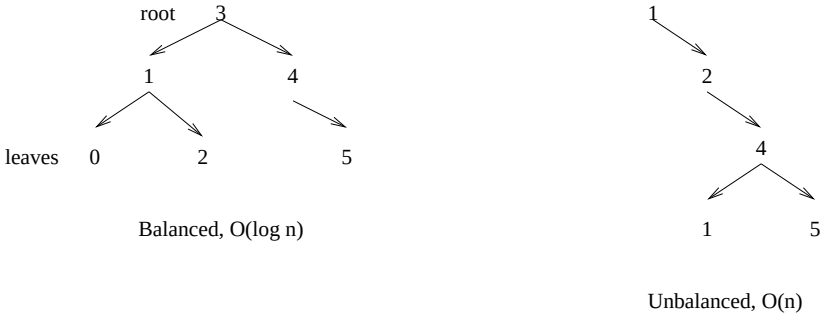

Figure 13: Balanced and unbalanced trees

This is how a tree is a generalisation of a list.

The good thing about trees is that you don't have to go very far from the root to get to any element. A balanced tree has *depth* (or *height*) $O(\log n)$ where n is the number of elements. See figure 13.

---

20 yes-no questions can specify a million things!

This means $O(\log n)$ access time.

	balanced	unbalanced (as list)	over all types
best	$O(1)$	$O(1)$	$O(1)$
average	$O(\log n)$	$O(n)$	$O(\log n)$
worst	$O(\log n)$	$O(n)$	$O(n)$

The items at the edge (the *fringe*) of the tree are called the *leaves*. The item a the top is the *root*. An item has a *parent* node above and some number of *child* nodes below. Nodes next to each other are *siblings*.

Trees are very flexible objects, though can be fiddly to program correctly. Consider the insertion of a new value: at the leaves it's not too bad. In the body of the tree is much harder, particularly if we want to keep the thing balanced. This means that insertion and deletion are $O(\log n)$.

With trees we don't allow loops like in lists. This is purely to keep management easy: the tree structure is much easier to manipulate if we don't have to keep worrying if we have visited a node before. In fact we tend to use a different name entirely if there are loops: we call it a *graph*.

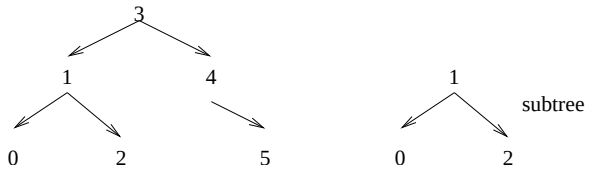

Figure 14: Subtrees

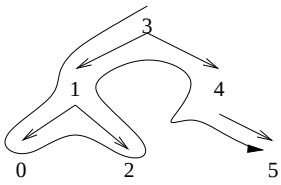

Figure 15: Traversing a Tree

3.3.1 Tree traversal

Trees are interesting structures as they have *subtrees* which are also themselves trees. This natural recursive structure means that it is natural to use recursion to manipulate trees.

Consider this algorithm **printtree1**:

- if the tree is empty, return
- print the root node value
- print, using **printtree1** the left subtree
- print, using **printtree1** the right subtree

This works because subtrees are trees. This is called a *pre-order depth-first traversal*. It is depth-first as we go downwards before going to a sibling.

On the simple example of figure 14 we get:

3 1 0 2 4 5

Consider **printtree2**:

- if the tree is empty, return
- print, using **printtree2** the left subtree
- print the root node value
- print, using **printtree2** the right subtree

This is a *in-order traversal*.

Now we get:

0 1 2 3 4 5

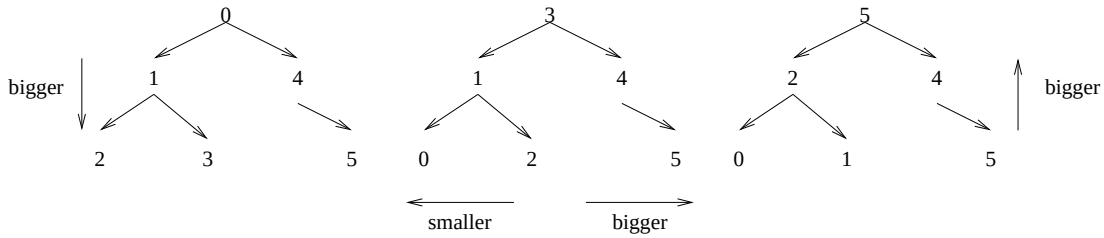

Figure 16: Orders in treees for the various traversals

Consider **printtree3**:

- if the tree is empty, return
- print, using **printtree3** the right subtree
- print the root node value
- print, using **printtree3** the left subtree

This is a *in-order traversal*, but reversed.

Now we get:

5 4 3 2 1 0

Lastly,**printtree4**:

- if the tree is empty, return
- print, using **printtree4** the left subtree
- print, using **printtree4** the right subtree
- print the root node value

This is *post-order traversal*.

Now we get:

0 2 1 5 4 3

We can also reverse the pre- and post-order traversals.

There is also a *breadth first* traversal which goes across the tree rather than down. This is useful in particular circumstances but is somewhat harder to program.

3 1 4 0 2 5

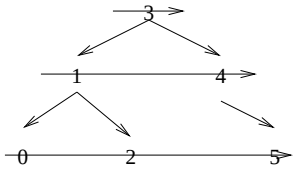

Figure 17: Breadth First

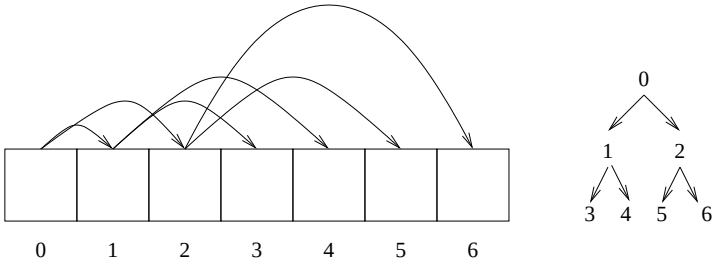

Figure 18: Tree in an Array

3.3.2 Implementation

```
struct inttree {
  int val;
  struct inttree *left, *right;
};
```

Or if we want to go upwards, too (c.f., doubly linked lists), though this is rare:

```
struct inttree {
  int val;
  struct inttree *parent, *left, *right;
};
```

3.3.3 Tree in an Array

Sometimes it is useful to have a tree in an array: element n has children $2n + 1$ and $2n + 2$. Element n has parent $\lfloor (n - 1)/2 \rfloor$ where the $\lfloor . \rfloor$ means "integer part of".

The root is element 0. This implementation has the advantages of an array: $O(1)$ access to any element and is it is very compact as no pointers are needed! Also, a breadth first traversal of the tree is a simple linear index along the array.

It also has the disadvantages of an array: fixed size so the need to preallocate a large enough array in advance.

4 Sorting

There are many algorithms that you might need. Two important classes are sorting and searching. Sorting is taking a bunch of things (numbers, names, whatever) and sorting them into some order (increasing, decreasing, alphabetic,

whatever). Searching is taking a bunch of data (possibly sorted, possibly not), and finding a particular datum within them (find the telephone number of this person, find the person with this telephone number).

We shall have examples based on sorting and searching numbers, but the algorithms work on any kinds of data that you can compare, e.g., alphabetic order for strings.

4.1 Selection

Possibly the first thing that comes into most people's head when asked to sort a bunch of numbers. It works by going through the array, finding the smallest value, and swapping it with the value in the first slot. The going though again, from the second slot onwards, finding the smallest value and swapping it with the value in the second slot. And so on.

Function **selectionsort**. Given an array A of n numbers, indexed 0 to $n-1$

- for i from 0 to $n-2$
- $k = i$, this is the index of the smallest value so far
- for j from $i+1$ to $n-1$
- if $A[j] < A[k]$ then $k = j$
- swap $A[i]$ and $A[k]$

Example. Sort 3 1 4 5 2

- $i = 0$ Set $k = 0$, so $A[k]$ is 3
- 3 1 4 5 2. Smaller, set $k = 1$, now $A[k] = 1$
- 3 1 4 5 2. OK
- 3 1 4 5 2. OK
- 3 1 4 5 2. OK
- Swap $A[0]$ and $A[1]$. 1 3 4 5 2
- $i = 1$ Set $k = 1$, $A[k]$ is 3
- 1 3 4 5 2. OK
- 1 3 4 5 2. OK
- 1 3 4 5 2. Smaller, set $k = 4$, now $A[k] = 2$
- Swap $A[1]$ and $A[4]$. 1 2 4 5 3
- $i = 2$ Set $k = 2$, so $A[k]$ is 4
- 1 2 4 5 3. OK
- 1 2 4 5 3. Smaller, set $k = 4$, now $A[k] = 3$
- Swap $A[2]$ and $A[4]$. 1 2 3 5 4
- $i = 3$ Set $k = 3$, so $A[k]$ is 5

- 1 2 3 5 4. Smaller, set $k = 4$, now $A[k] = 4$
- Swap $A[3]$ and $A[4]$. 1 2 3 4 5

The complexity of this algorithm is fairly easy to compute: the first j loop takes n steps, the next $n - 1$, and so on. The total is

$$n + (n-1) + (n-2) + \cdots + 3 + 2 = n(n+1)/2 - 1 = O(n^2)$$

Notice this time is independent of the data: even if the data is already sorted it takes $O(n^2)$ time!

- Time: best $O(n^2)$, average $O(n^2)$, worst $O(n^2)$.
- Space $O(n)$. More precisely: $n + O(1)$
- Other criteria: moves items directly to their destination, which can be important if the cost of moving is high. A quadratic algorithm, so bad for large datasets.

Sometimes the complexity of sorting algorithms is more finely subdivided. We might not just count the number of steps, but also

- the number of comparisons made, and
- the number of moves made.

For example, selection sort takes $O(n^2)$ comparisons but only $O(n)$ moves.

Sometimes moving an object is much more expensive than comparing (think of sorting a line of cars into numberplate order: you want to move the cars as little as possible). In such a case you might want an algorithm that uses as few moves as possible, even to the extent of doing a lot more comparisons.

Sometimes, a comparison is more expensive than a move. In this case you would want to minimise comparisons. For example, when sorting long strings of nearly-identical DNA into order, the comparison would be quite costly.

A slight improvement on the selection sort is the *shaker sort*. This saves some effort by gathering both the largest and smallest values in each sweep, putting the smallest at the bottom and the largest at the top. This is faster than the selection sort, but still $O(n^2)$ so poor for large datasets.

4.2 Insertion

The way people sort a hand of cards. The cards on the left are sorted, those on right as yet unsorted. We takes the next unsorted card and insert it into the sorted part in the right place.

Inserting a value into an array is a bit fiddly as it involves shifting up other values to make space. Insertion sort works better with a linked list where insertion is an easy operation.

Function **insertionsort**. Given an array A of n numbers, indexed 0 to $n - 1$

- for i from 1 to $n - 1$
- Set $s = A[i]$
- insert s into list $A[0]$ to $A[i - 1]$:

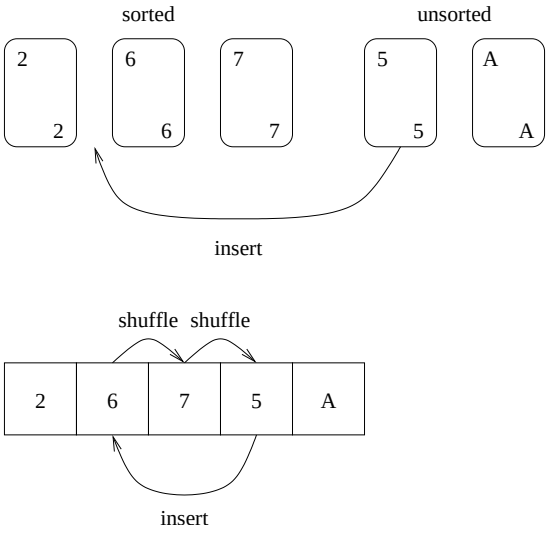

Figure 19: Insertion sort

- for j from $i-1$ to 0 by -1 while $A[j] > s$
- set $A[j+1] = A[j]$
- set $A[j+1] = s$

We compare right to left in the already sorted part, shuffing up as we go: we save the current value in s, and shift up the values in A until we get to the right place to put s.

Example. Sort 3 1 4 5 2

- The 3 is the sorted part, the 1 4 5 2 the unsorted part, shown 3 / 1 4 5 2
- Comparing 1: 3 / 1 4 5 2. Shift, 3 3 4 5 2
- End of loop, set value, 1 3 4 5 2
- Comparing 4: 1 3 / 4 5 2. Smaller, set value, 1 3 4 5 2
- Comparing 5: 1 3 4 / 5 2. Smaller, set value, 1 3 4 5 2
- Comparing 2: 1 3 4 5 / 2. Shift, 1 3 4 5 5
- Comparing 2: 1 3 4 5 / 5. Shift, 1 3 4 4 5
- Comparing 2: 1 3 4 4 / 5. Shift, 1 3 3 4 5
- Comparing 2: 1 3 3 4 / 5. Smaller, set value, 1 2 3 4 5

Notice how we overwrite values as we go, but always safely.

If the data are already ordered, the j loop exits immediately every time, and we take $O(n)$ steps. If not, the comparing against the sorted part takes

$$1 + 2 + 3 + \cdots + (n-2) = O(n^2)$$

steps again.

- Time: best $O(n)$, average $O(n^2)$, worst $O(n^2)$.
- Space: $O(n)$
- Other criteria: exceptionally good for nearly sorted data (e.g., we have a sorted list and we want to add a new element). Low overhead, so good for small datasets. Bad, since the $O(n^2)$ means this performs really poorly for large datasets. A lot of shuffling, thus is poor if moving objects is costly. Better suited for linked lists as they will require less shuffling as we insert.
- Comparisons: $O(n^2)$. Moves: $O(n^2)$.

4.3 Bubble

Another simple sort, often used by novice programmers due to its simplicity. The idea is to get more action in each sweep of the array, meaning more values are moved closer to their final positions than in selection sort.

Function **bubblesort**. Given an array A of n numbers, indexed 0 to $n-1$

- for i from 0 to $n-2$
- for j from 0 to $n-2$
- if $A[j] > A[j+1]$ then swap them

Each j loop passes along the array bubbling up the larger values as it goes and depositing them closer to their final place.

Example. Sort 3 1 4 5 2

- $i=0, j=0$: 3 1 4 5 2. Swap, 1 3 4 5 2
- $i=0, j=1$: 1 3 4 5 2. OK.
- $i=0, j=2$: 1 3 4 5 2. OK.
- $i=0, j=3$: 1 3 4 5 2. Swap, 1 3 4 2 5
- $i=1, j=0$: 1 3 4 2 5. OK.
- $i=1, j=1$: 1 3 4 2 5. OK.
- $i=1, j=2$: 1 3 4 2 5. Swap, 1 3 2 4 5
- $i=1, j=3$: 1 3 2 4 5. OK.
- $i=2, j=0$: 1 3 2 4 5. OK.
- $i=2, j=1$: 1 3 2 4 5. Swap, 1 2 3 4 5
- $i=2, j=2$: 1 2 3 4 5. OK
- $i=2, j=3$: 1 2 3 4 5. OK
- $i=3, j=0$: 1 2 3 4 5. OK.
- $i=3, j=1$: 1 2 3 4 5. OK.

- $i = 3, j = 2$: 1 2 3 4 5. OK.
- $i = 3, j = 3$: 1 2 3 4 5. OK.

This takes $O(n^2)$ steps and there is clearly lots of wasted effort. After each j loop, another value is bubbled up to its correct place, so it's pointless checking the last j pairs. This observation leads to

Function **bubblesort**. Given an array A of n numbers, indexed 0 to $n - 1$

- for i from 0 to $n - 2$
- for j from 0 to $n - i - 2$
- if $A[j] > A[j + 1]$ swap them

Example. Sort 3 1 4 5 2

- $i = 0, j = 0$: 3 1 4 5 2. Swap, 1 3 4 5 2
- $i = 0, j = 1$: 1 3 4 5 2. OK.
- $i = 0, j = 2$: 1 3 4 5 2. OK.
- $i = 0, j = 3$: 1 3 4 5 2. Swap, 1 3 4 2 5
- $i = 1, j = 0$: 1 3 4 2 5. OK.
- $i = 1, j = 1$: 1 3 4 2 5. OK.
- $i = 1, j = 2$: 1 3 4 2 5. Swap, 1 3 2 4 5
- $i = 2, j = 0$: 1 3 2 4 5. OK.
- $i = 2, j = 1$: 1 3 2 4 5. Swap, 1 2 3 4 5
- $i = 3, j = 0$: 1 2 3 4 5. OK.

This is good, but still $O(n^2)$. We can do better.

Example. Sort 1 2 3 5 4

- $i = 0, j = 0$: 1 2 3 5 4. OK
- $i = 0, j = 1$: 1 2 3 5 4. OK.
- $i = 0, j = 2$: 1 2 3 5 4. OK.
- $i = 0, j = 3$: 1 2 3 5 4. Swap, 1 2 3 4 5
- $i = 1, j = 0$: 1 2 3 4 5. OK.
- $i = 1, j = 1$: 1 2 3 4 5. OK.
- $i = 1, j = 2$: 1 2 3 4 5. OK.
- $i = 2, j = 0$: 1 2 3 4 5. OK.
- $i = 2, j = 1$: 1 2 3 4 5. OK.

- $i = 3, j = 0$: 1 2 3 4 5. OK.

After step $i = 0, j = 3$, everything is sorted, so we shouldn't have to go any further. But how do we know the array is sorted? One way is to compare each value with its neighbour to see if they are correctly ordered. But this is just what steps $i = 1$ do. If we get through a j loop without doing any swaps, everything must be in order, so we can stop.

Function **bubblesort**. Given an array A of n numbers, indexed 0 to $n - 1$

- for i from 0 to $n - 2$
- for j from 0 to $n - 2 - j$
- if $A[j] > A[j + 1]$ swap them
- if we have done no swaps in this loop, then stop

This is what is commonly taken as *bubble sort*.

Example. Sort 1 2 3 5 4

- $i = 0, j = 0$: 1 2 3 5 4. OK
- $i = 0, j = 1$: 1 2 3 5 4. OK.
- $i = 0, j = 2$: 1 2 3 5 4. OK.
- $i = 0, j = 3$: 1 2 3 5 4. Swap, 1 2 3 4 5
- $i = 1, j = 0$: 1 2 3 4 5. OK.
- $i = 1, j = 1$: 1 2 3 4 5. OK.
- $i = 1, j = 2$: 1 2 3 4 5. OK.
- No swaps for this i, so stop

If the data are already sorted, we make one pass through and stop immediately. Otherwise, we do about $(n - 2) + (n - 3) + \cdots + 3 + 2 + 1 = O(n^2)$ steps.

- Time: best $O(n)$, average $O(n^2)$, worst $O(n^2)$.
- Space $O(n)$.
- Other criteria: good for nearly sorted data. Low overhead, so good for small datasets. Bad, since the $O(n^2)$ means this performs really poorly for large datasets.
- Not so good for sorting lists.
- Comparisons: $O(n^2)$. Moves: $O(n^2)$.

The bubblesort is asymmetric: large values can move upwards several places in a single pass, while small values only ever move down one place. This means small values necessarily take many passes to get into place. The *bidirectional* bubblesort alternates passes up with passes down the array. This usually reduces the number of passes needed and so we can stop earlier.

4.4 Shell

A development of bubblesort, named after its inventor. A problem with bubblesort is that it takes many swaps to get an element to its final position: each swap moves it up just one place. The shell sort tries to move things further in each step with the intention that it is faster overall. So rather than comparing adjacent elements it compares values further apart.

4.5 Merge Sort

A *divide and conquer method*. It divides the data into two halves and sorts each half. The idea is that smaller chunks of data can be sorted disproportionately faster than large ones, so two smaller problems are better than one big one.

$$(n/2)^2 + (n/2)^2 = n^2/4 + n^2/4 = n^2/2 < n^2.$$

This one divides the data in two, sorts each half, and merges the halves together. Merging two sorted lists is easy: keep taking the smaller element from the starts of the two lists.

Function **mergesort**. Sort a list of n numbers

- if the list contains just one item, return it
- sort, using **mergesort**, the first half of the list
- sort, using **mergesort**, the second half of the list
- merge the two sorted lists together

Example. Sort 3 6 1 4 5 2 9 8

- sort 3 6 1 4
- sort 3 6
- sort 3: return 3
- sort 6: return 6
- merge 3 / 6: return 3 6
- sort 1: return 1
- sort 4: return 4
- merge 1 / 4: return 1 4
- merge 3 6 / 1 4: return 1 3 4 6
- sort 5 2 9 8
- sort 5 2
- sort 5: return 5
- sort 2: return 2

- merge 5 / 2: return 2 5
- sort 9: return 9
- sort 8: return 8
- merge 9 / 8: return 8 9
- merge 2 5 / 8 9: return 2 5 8 9
- merge 1 3 4 6 / 2 5 8 9: return 1 2 3 4 5 8 9

The complexity of this algorithm takes a little thought. Suppose sorting n object takes time $T(n)$. The algorithm sorts each half and then merges the two, so

$$T(n) = T(n/2) + T(n/2) + n = n + 2T(n/2)$$

Here, for now, we are taking the cost of merging to be n.

Similarly, $T(n/2) = n/2 + 2T(n/4)$, so

$$T(n) = n + 2T(n/2) = n + 2(n/2 + 2T(n/4)) = n + n + 4T(n/4) = 2n + 4T(n/4)$$

Continuing,

$$T(n) = 2n + 4T(n/4) = 3n + 8T(n/8) = 4n + 16T(n/16) = \ldots$$

so that

$$T(n) = kn + 2^k T(n/2^k).$$

Eventually, $n/2^k = 1$, which is $n = 2^k$ or $k = \log n$. Now $T(1) = 0$ (it takes no time to sort 1 object), so

$$T(n) = n \log n + 2^{\log n} T(1) = n \log n.$$

Problems: the merge is the hard part. This either requires an extra array to merge into, *or* it requires a lot of data shuffling. Consider merging 1 3 5 with 2 4 6. The 2 wants to go after the 1, which would require moving up the 3 5. This is poor in an array, but OK for a linked list. Shuffling in an array increases the complexity of the algorithm: in the worst case (e.g. merge 10 11 12 / 1 2 3) of merging two lists of length $n/2$ requires moving $n/2$ elements to insert the first element; then moving them again to insert the second; and so on. We make $n/2 + n/2 + \cdots = O(n^2)$ steps.

If we have to shuffle, then **mergesort** takes $O(n^2)$ time on average. It is still $O(n \log n)$ for best case: for sorted data an in-place merge is $O(n)$, e.g., merging 1 2 with 3 4, we just run along the first list checking that all its values are less than the first value in the second list.

Merging into a separate array is not necessarily the simple solution (the *double space merge sort*): we may not have the memory available (we might have a small machine or a huge dataset! In previous times, when machines were small, a version of merge sort the *tape sort* was used: this was when it was not possible to keep even a single copy of all the data in memory at once). Even if we do, we have to consider the cost of allocating and deallocating arrays.

Two arrays:

- Time: best $O(n \log n)$, average $O(n \log n)$, worst $O(n \log n)$.
- Space $2n$.

One Array:

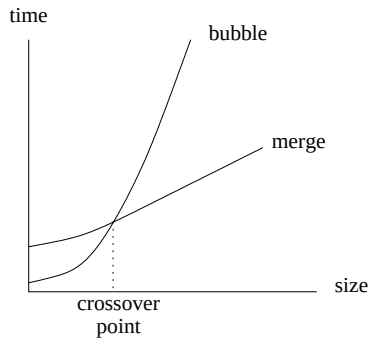

Figure 20: Crossover of Bubble and Merge sorts

- Time: best $O(n \log n)$, average $O(n^2)$, worst $O(n^2)$.
- Space n

Other criteria: not so good as bubble sort with nearly sorted data. Merge requires extra space or extra time to shuffle. Two array merge sort has very stable predictable behaviour.

One improvement is not to recurse all the way down to single elements: better is to switch to, say, insertion sort or even bubble sort when n is small enough. This takes advantage of the low overhead of such a sort and its speed on small datasets.

The *overhead* of an algorithm is the amount of messing about in the algorithm that is needed to make it work but doesn't really contribute much. For example, shuffling in the **mergesort** is overhead: we can't do it without it, but it's not really part of the sorting process. **bubblesort** only has a tiny amount of overhead (in the swapping of values), but is a poor algorithm. **mergesort** has more overhead, but is a better algorithm. This is why they have a crossover point: where the larger overhead is outweighed by the better algorithm. The choice of algorithm is never easy to make!

4.6 Quick

Another divide and conquer method, like **mergesort**, but the divide phase is somewhat different. Somewhat more sophisticated than merge sort it is good to use as long as certain situations are avoided. Quicksort can be an *in place* sort, meaning it does not need extra space like **mergesort**.

quicksort works by picking some value in the list, called the *pivot*, and *partitioning* the list into those values less than the pivot and those values greater than the pivot. These two smaller lists can be sorted (using **quicksort**!), then the sorted list is the small partition, then the pivot, then the large partition.

Function **quicksort**. Sort a list of n numbers

- if the list contains just one item or fewer, return it
- pick a *pivot*, e.g., the first item in the list
- put all the values that are less than the pivot into list A, and all the values that are greater than the pivot into list B
- sort, using **quicksort**, list A
- sort, using **quicksort**, list B
- output list A, the pivot, list B

Example. Sort 3 6 1 4 5 2 9 8

- choose pivot 3
- partition about 3: 1 2 / 3 / 6 4 5 9 8
- sub-problem: sort 1 2
- choose pivot 1
- partition about 1: / 1 / 2
- sub-sub-problem: sort 2
- return 2
- return 1 2
- sub-problem: sort 6 4 5 9 8
- choose pivot 6
- partition about 6: 4 5 / 6 / 9 8
- sub-sub-problem: sort 4 5
- choose pivot 4
- partition about 4: / 4 / 5
- sub-sub-sub problem: sort 5
- return 5
- return 4 5
- sub-sub-problem: sort 9 8
- choose pivot 9
- partition about 9: 8 / 9 /
- sub-sub-sub problem: sort 8
- return 8
- return 8 9
- return 4 5 6 8 9
- return 1 2 3 4 5 6 8 9

The major benefit that **quicksort** has over mergesort is that it can easily be done in-place.

We start with the pivot in array slot 0 (if it was not already there, simply swap it with the value that is there). Move up the array, swapping values to before the pivot wherever necessary.

See figure 21. We need to put the 4 before the pivot 5. So we must shift up the 5 by one place. Where to put the 7? Put it where the 4 was. Rotating the three values 4, 5 and 7 does the trick. Thus we can do the partition phase in one sweep along the array.

A value to the right of the pivot that is greater than the pivot remains where it is.

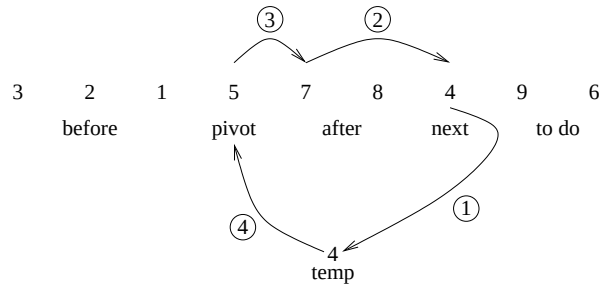

Figure 21: In-place partition for quicksort

```
pivot = 0;                  // index of pivot

for (i = 1; i < n; i++) {
  if (A[i] < A[pivot]) {   // if value less than pivot
    t = A[i];              // save current value
    A[i] = A[pivot+1];     // move value after pivot
    A[pivot+1] = A[pivot]; // shift up pivot value
    A[pivot] = t;          // store saved value where pivot was
    pivot++;               // new index for pivot
  }                        // else nothing to do when value greater than pivot
}
```

We can find complexity of this algorithm is like we did for **mergesort**. Suppose sorting n objects takes time $T(n)$. If we get a perfect split, we see that

$$T(n) = n + T(n/2) + T(n/2) = n + 2T(n/2)$$

where n is the division, and $T(n/2)$ is the time to sort $n/2$ objects. Just as with **mergesort**, we find the algorithm takes time $O(n \log n)$, *if we get even splits every time*.

On the other hand, we can get bad splits: consider sorting 1 2 3 4 5.

- choose pivot 1
- partition about 1: / 1 / 2 3 4 5
- sub-problem: sort empty list, nothing to do
- sub-problem: sort 2 3 4 5
- choose pivot 2
- partition about 2: / 2 / 3 4 5
- sub-sub-problem: sort empty list, nothing to do
- sub-sub-problem: sort 3 4 5
- choose pivot 3
- partition about 3: / 3 / 4 5
- sub-sub-sub-problem: sort empty list, nothing to do

- sub-sub-sub-problem: sort 4 5
- choose pivot 4
- partition about 4: / 4 / 5
- sub-sub-sub-sub-problem: sort empty list, nothing to do
- sub-sub-sub-sub-problem: sort 5
- return 5
- return 4 5
- return 3 4 5
- return 2 3 4 5
- return 1 2 3 4 5

So in the worst case we get a partition into 0 plus $n-1$ elements, and we get $T(n) = n + T(0) + T(n-1) = n + T(n-1)$, as $T(0) = 0$, the time to sort no objects. So now

$$\begin{aligned} T(n) &= n + T(n-1) \\ &= n + (n-1) + T(n-2) \\ &= n + (n-1) + (n-2) + T(n-3) \\ &= \ldots \\ &= n + (n-1) + \cdots + 3 + 2 + 1 \\ &= n(n+1)/2 \\ &= O(n^2) \end{aligned}$$

So, if we get bad splits, the time is $O(n^2)$.

Thus choosing the pivot is critical. The ideal pivot is a middling value that splits the problem into equal parts. Choosing a bad pivot leads to an uneven division, and the behaviour is then very bad.

When the data is already sorted or nearly sorted, **quicksort** does really poorly. Ditto for reversed or nearly reversed data. It is a strange feature of **quicksort** that sorted data is its worst case!

- One approach, rather counter-intuitively, is to randomise the order of the data before using **quicksort**: this might well give us a better run time!
- Some people take a random value from the array to use as the pivot. Others take three values from array and use the middle value as the pivot (*median* quicksort). This improves things a little, but is not perfect.
- Another variant is to take two pivots and partition into three parts, though the small gain in speed for the extra program complexity means that doing this is not usually worth the effort.
- If we have extra information about our data, say, it is already nearly sorted, we can use that information to help us choose a pivot. So, for example, with nearly sorted data we could choose a pivot from the middle of the array, which would likely be near the median value.

Picking a pivot from the middle of the array helps with the sorted data case, but trips up over other (less common) cases: e.g., sort 4 2 5 1 6 3 7. In the middle is the 1. The two partitions 4 2 4 and 6 3 7 have their smallest values in their middles, too.

It turns out that we can find the middle value of an array in $O(n)$ time. Consider the algorithm **select**:

1. consider the array in groups of 5 values: sort each 5 (using bubblesort, perhaps) and find the middle value of each
2. call **select** recursively on these middle values to find the overall middle value

Using **select** to find a pivot, we can guarantee that **quicksort** has a good behaviour, with additional $O(n \log n)$ time ($O(n)$ for $\log n$ sort steps) taken to find the pivot. Thus, still a total time of $O(n \log n)$, but somewhat longer than previously. Of course, it may or may not be worthwhile spending this amount of time finding the pivot when one of the methods above might be just as good: it all depends on the data.

Just as with any other divide and conquer method, there is no need to recurse all the way down to single elements in **quicksort**. Better is to stop at 3 or 4 elements and sort these directly; or revert to **insertion** (or **bubble**, **selection** or other low-overhead sort) when the list is short enough. This cut-over length must be determined experimentally as it will vary from implementation to implementation.

- Time: best $O(n \log n)$, average $O(n \log n)$, worst $O(n^2)$.
- Space $O(n)$.
- Other criteria: in place sort. When fast, is better than merge sort. Bad for sorted and nearly sorted data. Data from an actual implementation indicated that **quicksort** on 17 million numbers took about a minute if the data are randomly ordered, but would take about a year if they were already sorted.

If care is taken over the sorted case, **quicksort** is a good sort to use.

Standard C libraries contain a function `qsort` that implements **quicksort**. It has prototype

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

This is a function that will sort an array of arbitrary data items. The arguments are

- `base` the start of the array of items
- `nmemb` the number of items
- `size` the size of each data item (for example, an array of integers might have `size == 4`)
- `compar` a function that takes pointers to two items and returns a value < 0 or $= 0$ or > 0 acording to whether the first data item should be ordered before, the same or after the second data item.

To sort an array of integers:

```
int compint(int *a, int *b)
{
  if (*a < *b) return -1;
  if (*a > *b) return 1;
  return 0;
}
...
int vals[10];
...
qsort(vals, 10, sizeof(int), compint);
```

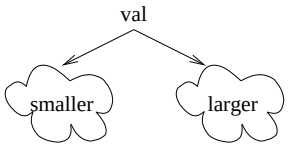

Figure 22: Tree for In-Order Traversal

The GNU glibc implementation of `qsort` contains these comments

```
/* Discontinue quicksort algorithm when partition gets below this size.
   This particular magic number was chosen to work best on a Sun 4/260. */
#define MAX_THRESH 4

...

/* Order size using quicksort.  This implementation incorporates
   four optimizations discussed in Sedgewick:

   1. Non-recursive, using an explicit stack of pointer that store the
      next array partition to sort.  To save time, this maximum amount
      of space required to store an array of SIZE_MAX is allocated on the
      stack.  Assuming a 32-bit (64 bit) integer for size_t, this needs
      only 32 * sizeof(stack_node) == 256 bytes (for 64 bit: 1024 bytes).
      Pretty cheap, actually.

   2. Chose the pivot element using a median-of-three decision tree.
      This reduces the probability of selecting a bad pivot value and
      eliminates certain extraneous comparisons.

   3. Only quicksorts TOTAL_ELEMS / MAX_THRESH partitions, leaving
      insertion sort to order the MAX_THRESH items within each partition.
      This is a big win, since insertion sort is faster for small, mostly
      sorted array segments.

   4. The larger of the two sub-partitions is always pushed onto the
      stack first, with the algorithm then concentrating on the
      smaller partition.  This *guarantees* no more than log (total_elems)
      stack size is needed (actually O(1) in this case)!  */
```

4.7 Tree

Unbalanced trees are easy: start with an empty tree. Insert each object into the tree such that the left children are all less than the node value, and all right children are greater than the node value. Then do a in-order depth-first traversal to get the sorted values.

Function **inserttree**. Insert a value in a tree.

- if the tree is empty set the tree to be this node and return it
- if the value is less than the root value then:

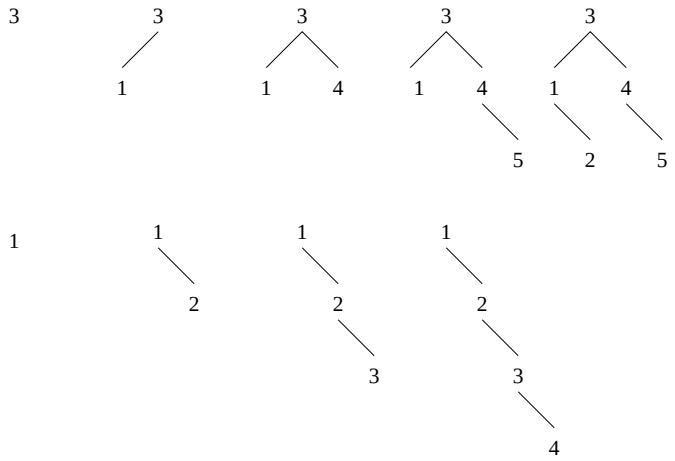

Figure 23: Tree Sorts: balanced and unbalanced

- if the left subtree is empty then set the left subtree to be a new node containing the value
- else insert, using **inserttree**, the value in the left subtree
- else if the right subtree is empty then set the right subtree to be a new node containing the value
- else insert, using **inserttree**, the value in the right subtree

Function **treesort**. Print values in a tree in increasing order.

- For each value
- insert the value in the tree using **inserttree**
- do an in-order traversal of the tree to get the sorted data

Or a reversed in-order traversal to get the reversed data!

Example. Sort 3 1 4 5 2. See figure 23.

If the tree remains roughly balanced, it has depth about $O(\log n)$. Inserting a value takes about $O(\log n)$ steps, giving a total complexity of $O(n \log n)$.

Unfortunately, trees get unbalanced very easily. Starting with sorted data the tree grows deep in one direction, losing the $\log$ depth. This means that the sort becomes an insertion sort with order $O(n^2)$.

- Time: best $O(n \log n)$, average $O(n \log n)$, worst $O(n^2)$.
- Space $O(n)$.
- Other criteria: good for when we don't know how much data we have in advance (streamed data). Good for random data, bad for nearly sorted and reverse nearly sorted data. Needs good memory management building the tree, or else the time taken to manipulate memory will dominate the time taken to do the sort.

We can try to keep the tree balanced. Start with an empty tree. Insert each object into the tree such that the left children are all less than the node value, and all right children are greater than the node value, *rebalancing* when necessary.

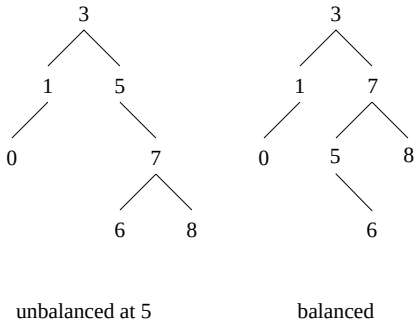

Figure 24: Balancing Trees

This is a complex manoeuvre that move subtrees and values around. Several rebalancing methods exist, all of them pretty complicated.

For example, *AVL trees*, by Adelson-Velskii and Landis (1962). This is a balanced tree where the heights of the two subtrees at every node differ by at most one. In figure 24, left, the two subtrees of 3 have heights 2 and 3, so the tree is balanced at 3. On the other hand, the subtrees at 5 have heights 0 and 2, so the tree is unbalanced here. In the right figure the subtrees at 7 have heights 1 and 2 and the tree is balanced everywhere. Note the tree has two elements in the left subtree at 3 and four elements in the right subtree, so this definition of "balanced" may differ slightly from the intuitive meaning of the word, but it is what we need for AVL as it keeps the ***heights*** of the subtrees in control.

We add a new value to to tree just as in the unbalanced case. If the tree is now unbalanced according to our definition, we rebalance it by using *rotations*:

- in a *left* rotation a node N is moved down and to the left. N's right child replaces N, and the right child's left child becomes N's right child.
- in a *right* rotation a node N is moved down and to the right. N's left child replaces N, and the left child's right child becomes N's left child

When we say "child" in the above we mean "child and all of its subtree".

We do one, occasionally two, appropriate rotations to make the tree balanced. We use a left rotation when the right subtree has greater height than the left, and vice versa. Inserting an element requires looking at all the nodes above the new node in order to check the balance. Thus insertion takes $O(\log n)$ steps, as this is the height of a balanced tree of n items. So inserting n elements takes $O(n \log n)$ time.

Then do a in-order traversal to get the sorted values, as before. The trick in making balanced trees is to make the rebalancing fast so that we don't push the overall complexity above the $O(n \log n)$ we want.

- Time: best $O(n \log n)$, average $O(n \log n)$, worst $O(n \log n)$.
- Space $O(n)$.
- Other criteria: good for when we don't know how much data we have in advance. Hard to get right. Needs good memory management.

AVL is not suitable for tree-in-an-array implementations as the rotation operations require shuffling subtrees, which is not cheap in an array.

There are many other kinds of balanced tree.

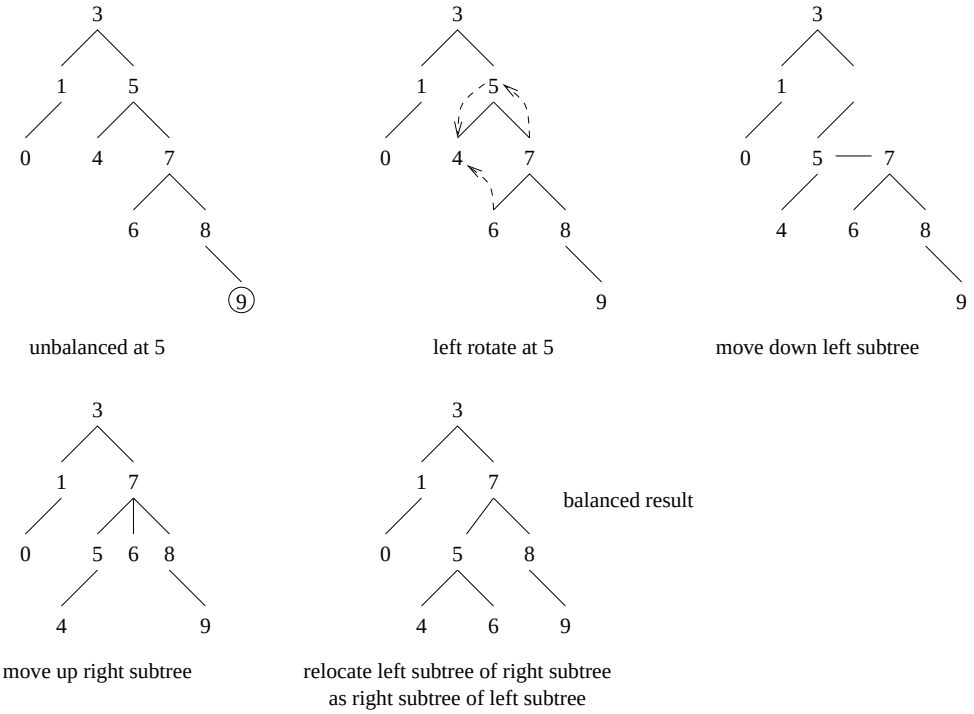

Figure 25: AVL rotation

- Red-Black trees. These colour nodes red or black and there is a set of rules governing how red and black nodes may be arranged. There is another notion of "balanced" that is looser then AVL's, but it still retains the $O(\log n)$ property.
- B-trees. These have a variable number of children in a given range, e.g., a 2-3 tree has 2 or 3 children. All leaf nodes are kept at the same depth. Used in databases, where the nodes tend to live on disk, rather than in memory.
- Splay trees. Really for searching problems. As we search a splay tree for an item the tree is rearranged so to make commonly requested items near the top.
- And so on.

4.8 Heap

This is a kind of a merger of tree and bubble sorts, usually implemented in an array, but can be done using a tree.

This algorithm works by entering values into a tree at the leaves, then bubbling them into the right place. At every stage the value at the root will be the smallest so far. In fact, at every stage each node's value is less than all the values in the subtrees below that node.

Function **heapsort**.

Phase 1:

- For each value

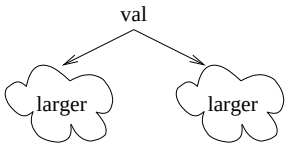

Figure 26: Heap: smallest value is always at root

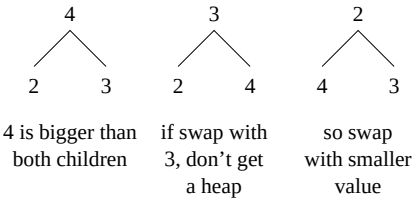

Figure 27: Swap with smaller child

- insert value at next free leaf (N.B. we simply insert values at each successive leaf working left to right in rows and don't search for a place like in **treesort**)
- bubble up the tree, i.e., while value is less than its parent, swap it with its parent. This is called *heapifying*.

We now have our data in a *heap*. How should we get our sorted data? Traversals don't work on heaps, they are not ordered that way. Instead look a the root: it contains the smallest value. We are going to take our heap apart by successively removing the root value, which at each stage the is next smallest value.

Phase 2:

- Repeat
- output value at root
- remove value at last leaf and place at root
- heapify: bubble down the tree, i.e., while value is greater than a child, swap it with that child. If value is greater than both children, swap it with the *smaller* child.

Example. Sort 3 1 4 5 2.

In an array this is:

```
Phase 1:
3              3 1            1 3            1 3 4
insert         insert         bubble         insert

1 3 4 5        1 3 4 5 2      1 2 4 5 3
insert         insert         bubble

Phase 2:
1 2 4 5 3      3 2 4 5 1      2 3 4 5 1      2 3 4 5 1
output 1       leaf to root   bubble         output 2

5 3 4 2 1      3 5 4 2 1      3 5 4 2 1      4 5 3 2 1
leaf to root   bubble         output 3       leaf to root
```

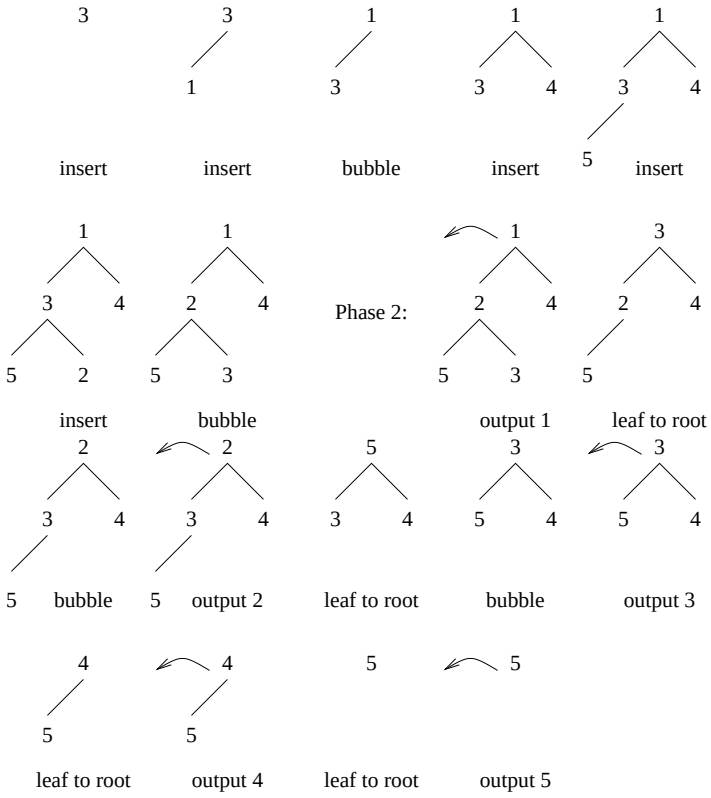

Figure 28: Heap Sort

```
4 5 3 2 1       5 4 3 2 1       5 4 3 2 1
output 4        leaf to root    output 5
```

In real implementations we simply put the root value where the next left comes from. This leaves the data sorted in decreasing order in the array. In practice we would heap sort for the *largest* values to leave the data in decreasing order. The insertion step is a no-op, and we don't "output" anything in phase 2, but swap the root element with the next leaf. When bubbling down we take care not to swap with the sorted part of the array, i.e., the values that have been swapped from the root.

We are left with the data in increasing order at the end.

```
Phase 1:
3 (1 4 5 2)     3 1 (4 5 2)     3 1 (4 5 2)      3 1 4 (5 2)
^                 ^                                  ^
next             next           no bubble         next

4 1 3 (5 2)     4 1 3 5 (2)     4 1 3 5 (2)      4 5 3 1 (2)
^   ^                 ^             ^   ^            ^ ^
bubble          next            bubble           bubble

5 4 3 1 2       5 4 3 1 2
        ^
next            no bubble

Phase 2:
5 4 3 1 2       2 4 3 1 (5)     4 2 3 1 (5)      1 2 3 (4 5)
^       ^       ^ ^             ^     ^          ^   ^
swap root       bubble          swap root        bubble

3 2 1 (4 5)     1 2 (3 4 5)     2 1 (3 4 5)      1 (2 3 4 5)
^   ^           ^ ^             ^ ^              ^
swap root       bubble          swap root        swap root

(1 2 3 4 5)

done
```

Parentheses indicate the part of the heap that should not participate in the bubbling.

Heapsort has a tendency to do swaps when they are not needed, but it is quite hard to avoid them.

Phase 1: there are n elements each taking $O(\log n)$ steps to enter, total $O(n \log n)$.

Phase 2: removing n elements, each taking $O(\log n)$ steps, total $O(n \log n)$.

So $O(n \log n)$ in total.

- Time: best $O(n \log n)$, average $O(n \log n)$, worst $O(n \log n)$.
- Space $O(n)$.

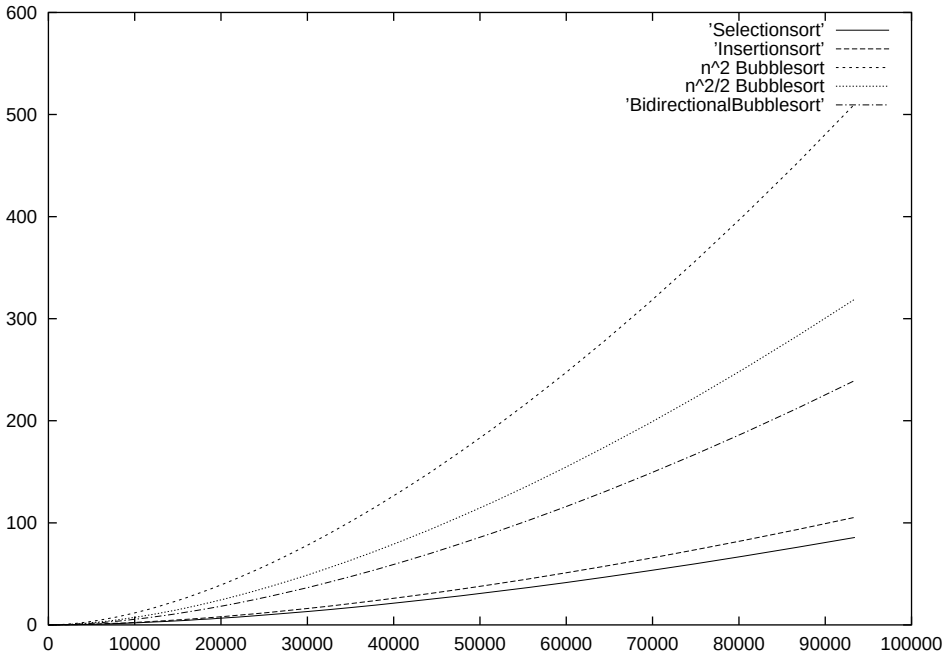

Figure 29: Quadratic Sorts

- Other criteria: reliable behaviour in all cases. In-place. Can't take advantage of nearly sorted data. Sorted/Reverse sorted is a worst case for Phase 1 as every insertion must be bubbled all the way to the root, but worst case still has a good complexity. Reverse sorted/Sorted is best case for Phase 1, but worst case for Phase 2 as the leaf that is swapped to the root must be bubbled all the way down again. Fiddly to program, but not so bad as some other sorts. **Heapsort** is usually slower than (enhanced) **quicksort**, but no clever tricks are needed to avoid bad cases: there are none!

4.9 Graphs

See figure 29 and following. These we made from data from actual implementations of the various sorts. No particular optimisations or clever tricks were used: these are the raw sorts.

4.10 Others

Radix/Bucket (sorting CDs). "Malgorithms": n^3 sort (favoured by students in exams!), throw-a-pack-of-cards-in-the-air sort (best $O(n)$, average $O(n!)$, worst infinite). Not-in-memory sorts. Splay tree: the tree is re-arranged every time it is traversed to keep commonly wanted stuff near the top. Red-Black tree: a more loose idea of balanced than AVL, but still $O(\log n)$. B-tree: a variable number of children, e.g., 2-3 has 2 or 3. All leaf nodes are kept at the same depth. Used in databases: the nodes are stored on disk and there can be 1000s of childrens for each node (to keep the tree shallow, so few lookups required to find a leaf).

Stable sorts.

It is possible to prove that *any* "comparison based" sorting algorithm *must* take at least $\Omega(n \log n)$ steps. By "comparison based" we mean that the only information used is gained by comparing elements against each other. If other information is available, it might be possible to go faster: the Spaghetti sort is a good example.

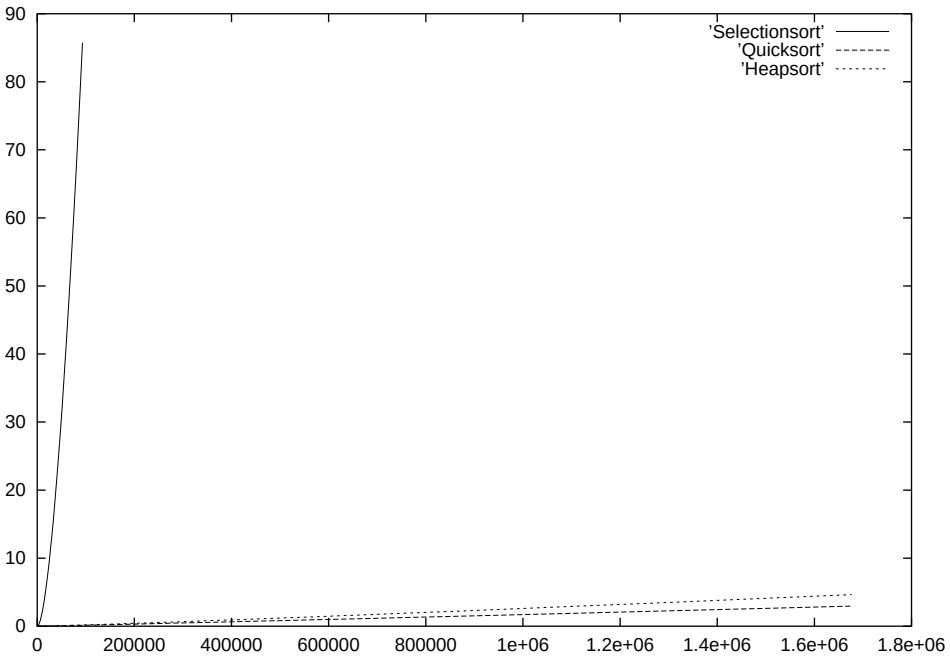

Figure 30: Quadratic and $n \log n$ Sorts

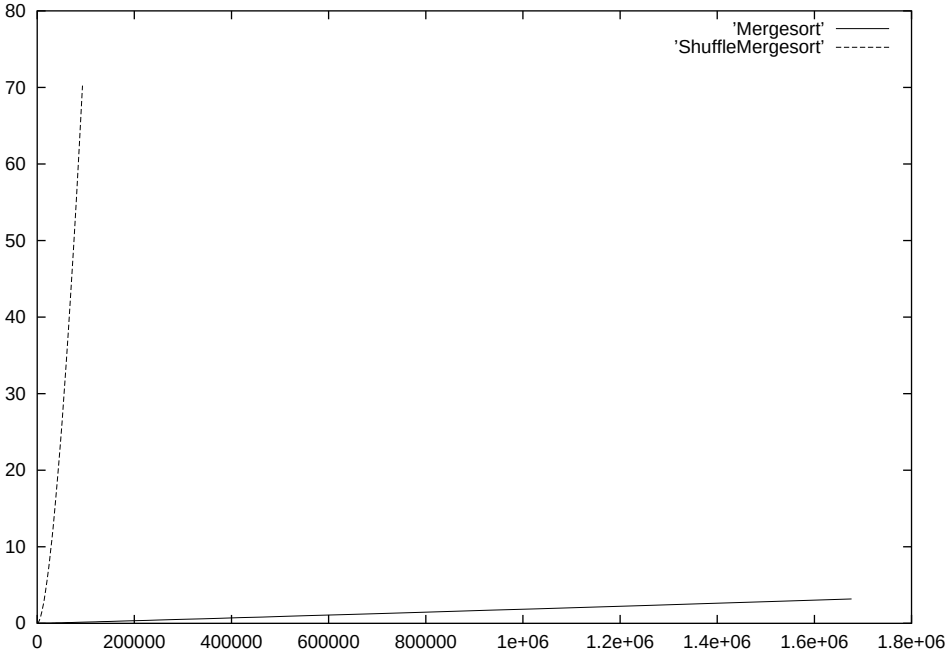

Figure 31: Merge Sorts

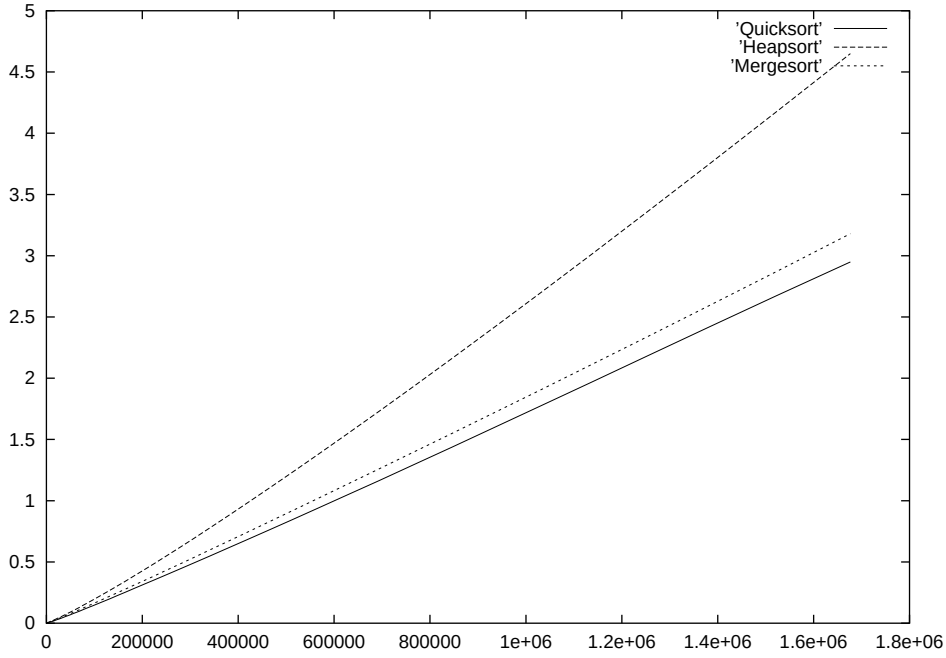

Figure 32: $n \log n$ Sorts

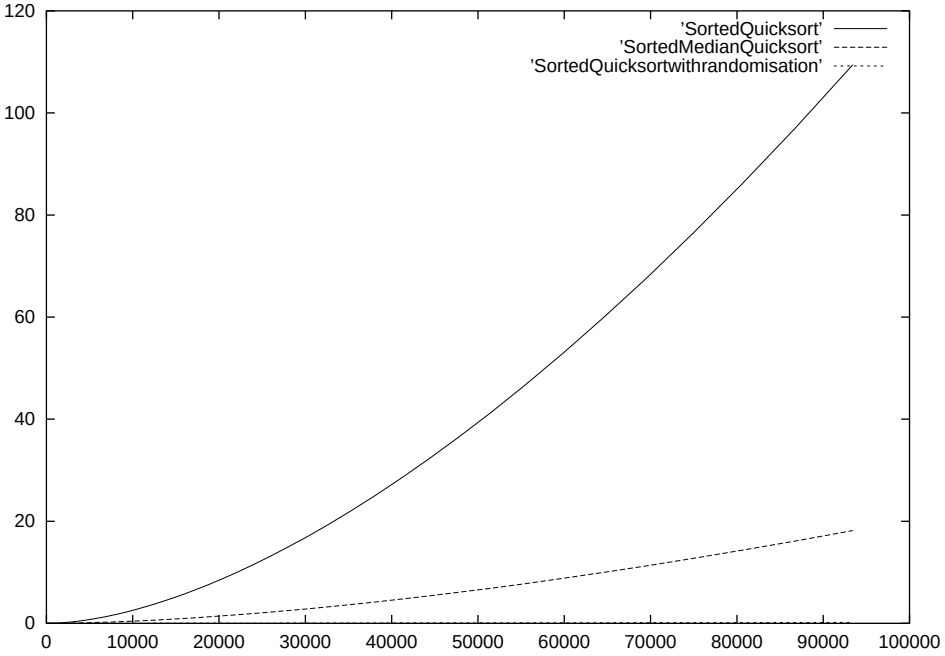

Figure 33: Quicksort with Sorted Data

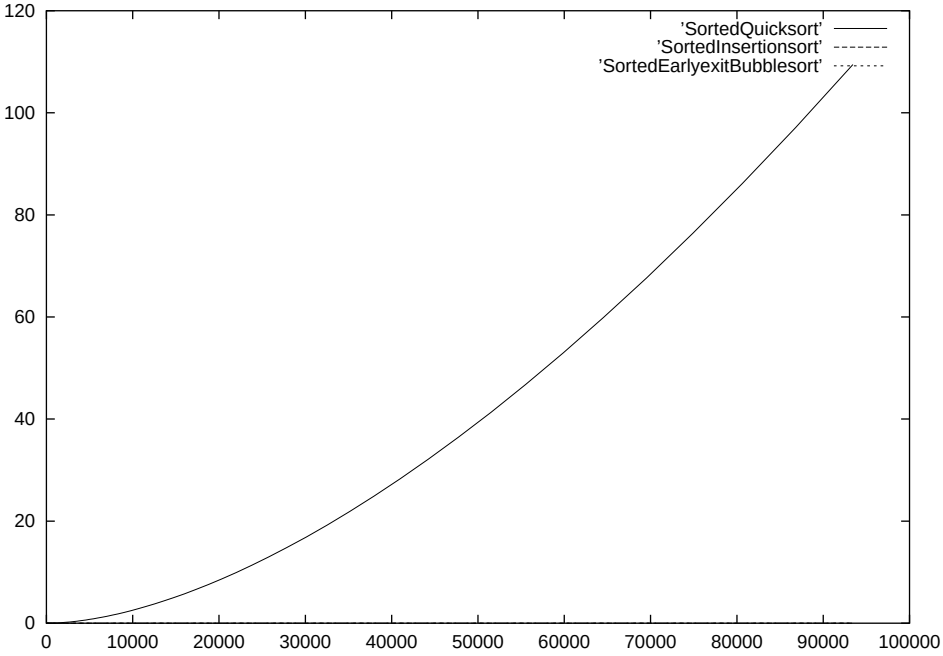

Figure 34: Sorted Data

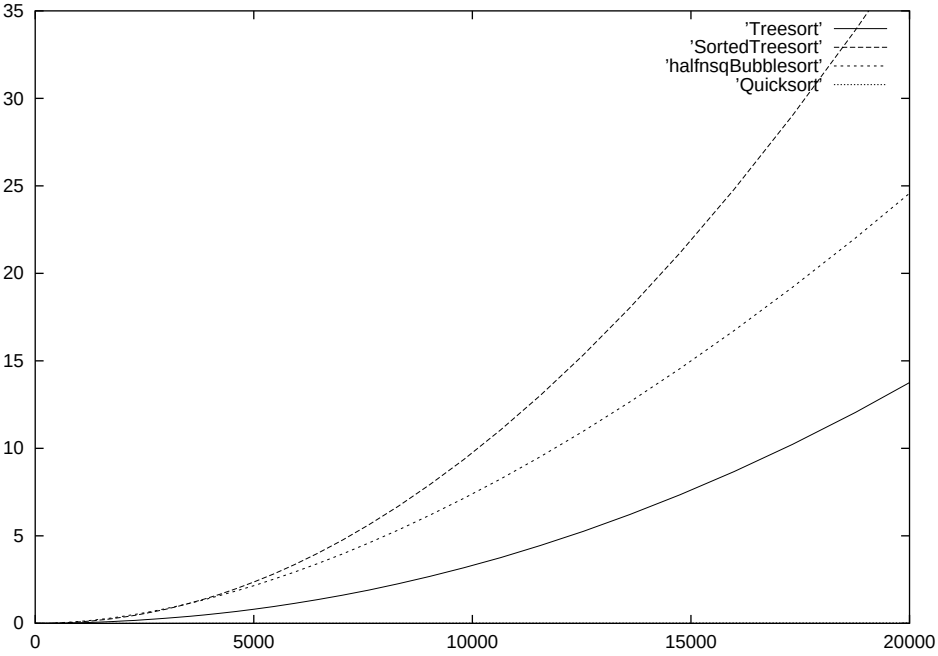

Figure 35: Trees

Given n objects we must pick one order from $n!$ possible orders. In doing comparisons of elements against each other we are building a binary tree ("go left for those orders with $a < b$, go right for the others"). Thus we have a tree of $n!$ leaves. This has depth $\log(n!) \approx n \log n$.

4.11 Counting Sort

More examples of sorts that are very fast, but dependent on the form of the data.

This counts the number of repetitions of a value, and uses that to sort the values.

Example. Sort 5 8 3 8 10 7. We are told that the data is guaranteed to lie in the range 1 to 10.

Go through the data and count the number of occurences of each value. Set count array to all 0s.

- count: 0 0 0 0 0 0 0 0 0 0
- found a 5: 0 0 0 0 1 0 0 0 0 0
- found an 8: 0 0 0 0 1 0 0 1 0 0
- found a 3: 0 0 1 0 1 0 0 1 0 0
- found an 8: 0 0 1 0 1 0 0 2 0 0
- found a 10: 0 0 1 0 1 0 0 2 0 1
- found a 7: 0 0 1 0 1 0 1 2 0 1

Now go through the count array and modify it so each value is the sum of all the values less than it, i.e., we count the number of numbers that are less than or equal to each value.

We get: 0 0 1 1 2 2 3 5 5 6.

- Look at the last value 7: count[7] = 3, so there are 3 values less than equal to our 7, so 7 goes into the third slot. * * 7 * * * Also decrement count[7] to 2, in case we find another 7, which we would place in the second slot. count: 0 0 1 1 2 2 2 5 5 6
- Next value, 10, has count 6. Put into sixth slot * * 7 * * 10, decrement count for 10 0 0 1 1 2 2 2 5 5 5
- Next value 8, count 5: * * 7 * 5 10, decrement count for 8 0 0 1 1 2 2 2 4 5 5
- Next value 3, count 1: 3 * 7 * 5 10, decrement count for 3 0 0 1 1 2 2 2 4 5 5
- Next value 8, count 4: 3 * 7 8 5 10, decrement count for 8 0 0 1 1 2 2 2 3 5 5
- Last value 5, count 2: 3 5 7 8 5 10, decrement count for 5 0 0 1 1 1 2 2 3 5 5

We get the sorted 3 5 7 8 5 10.

This requires an array the size of the range of the data, but takes $O(m + n)$ steps, where m is the range of the data.

4.12 Radix Sort

Again we sort numbers, but this time they are too large to do a counting sort.

Example. Sort 4321, 6544, 1233, 5436, 42.

We proceed by picking off a digit at a time:

- Sort the numbers by their last digit: 1, 4, 3, 6, 2. Use any sort, perhaps counting sort. We get 4321, 42, 1233, 6544, 5436.
- Sort by the 10s digit: 2, 4, 3, 4, 3. We get 4321, 1233, 5436, 42, 6544.
- Sort by the 100s digit: 3, 2, 4, 0, 5. We get 42, 1233, 4321, 5436, 6544.
- Sort by the 1000s digit: 0, 1, 4, 5, 6. We get 42, 1233, 4321, 5436, 6544.

This requires a subsort that is *stable*. It takes $O(kn)$ time, where k is the length of the numbers. If there are many numbers that are relatively short, this is about $O(n)$.

4.13 Which to Choose?

Depends. Size of n. What you want to optimise: space or time or something else. How the data are presented or accessed.

More specific complexity measures, e.g., number of comparisons between values, number of moves of values. For example, comparing strings is more expensive than comparing numbers.

- Bubble: comparisons $O(n^2)$, moves $O(n^2)$ (average)
- Selection: comparisons $O(n^2)$, moves $O(n)$.

4.14 Searching

Suppose we have a large array of data (or linked list, or tree or whatever). We wish to find a certain element within it. For example, find the smallest value; or largest; or second smallest; or middle value; or whatever. Find a particular word in a index of a book; find a word in a dictionary.

Just as with sorting, some datastructures lend themselves very well to some searching algorithms, though there are never hard and fast rules.

Requirements. Search: return if found, say if not. If a value is not there, we want to know that, too. Membership: sometimes we just need to know if the value is there, for example "is this person a member of the library?"

The ultimate goal of searching is to support databases. These are "simply" large collections of data, sometimes terabytes or more, that must be searched as fast as possible for data that fit possibly complicated criteria: find all people who live in Essex, are over the age of 21, don't have blue eyes and have income less than twice their credit card bill.

In a database the *record* contains useful information, such as credit limits and the like, not just the *key*, the value we search for. The key is used to determine which record to retrieve. The clever thing about databases is they allow efficient searching on several possible keys (age, name, eye colour) not just one.

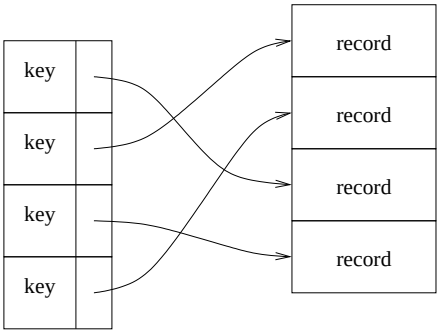

Figure 36: Database Records

4.15 Linear

This is when we go through each element in the datastructure one-by-one until we find the one we want. If we get to the end, we know it's not there.

Sometimes called *brute force*, *enumeration* or even *British Museum* search.

- Time: $O(n)$.
- Space: $O(n)$.

Best case for time is $O(1)$ when we find the key we are looking for immediately. However, best case happens with vanishingly small probability, so this is not a particularly useful measure.

Linear search is more-or-less forced upon us when we are searching a simple linked list.

This sounds bad, but can be good in two cases:

- when the table (or whatever) is small. If we have 5 things to search it is usually fastest to look at each directly rather than employ some more sophisticated algorithm.
- When the data are sorted somehow.

If, say, we have a list of integers sorted in increasing order it is easy to find the smallest value. If we wish to determine whether a value v is in the list we can stop searching as soon as we get to a value larger than v. This is faster, on average, than searching unsorted data, but is still $O(n)$.

- Time: $O(n)$. Perhaps more if we have to sort the data every time before a search. Generally, though we sort just once and search many times.
- Space: $O(n)$. Ditto above.
- Poor for large datasets, good for very small datasets.
- Inserting new elements can either be very easy ($O(1)$) if there is a gap in the table, or hard ($O(n)$) if not and we have to shuffle up a load of data to make room.
- Deletion from a table is always easy ($O(1)$).

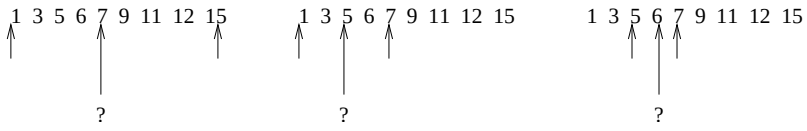

Figure 37: Binary Search

4.16 Binary

A method quicker than linear search, but only applicable when we have quick access to any element in a sorted datastructure. Sometimes called *binary chop*.

Function **binarysearch**. Find a value v in an array of length n.

- Do **binarysearchrange** for v in the range 0 to $n-1$

Function **binarysearchrange**. Find a value v in an array between indices l and r

- let $m = (l+r)/2$ and look at the value $A[m]$
- if $A[m] = v$ return it
- if $l = r$ return "not found"
- if $v < A[m]$ return **binarysearchrange** for v in the range l to $m-1$
- return **binarysearchrange** for v in the range $m+1$ to r

Example. Determine if 6 is in the array 1 3 5 6 7 9 11 12 15. Our array is indexed 0 to 8.

- Our number is somewhere between index 0 and index 8
- Look at the value at index $(0+8)/2 = 4$, namely 7.
- 7 is greater than 6, so our number is in the lower range 0-4.
- Look at the value at index $(0+4)/2 = 2$, namely 5.
- 5 is less than 6, so our number is in the lower range 2-4.
- Look at the value at index $(2+4)/2 = 3$, namely 6.
- Found it!

Each comparison halves the region where our number can lie. After m steps we have narrowed down to $n/2^m$ values. So, when we have narrowed down to one value $n/2^m = 1$, which is to say $2^m = n$, or $m = \log n$. Thus the algorithm takes time $O(\log n)$.

- Time: $O(\log n)$.
- Space $O(n)$.
- Requires data to be sorted and therefore data that *can be* sorted.
- Can exit early with "not found" if $v < A[l]$ or $v > A[r]$, i.e., v is outside the range of consideration.

- Insertion of a new value needs to be sorted into the right place, but we can use bubble to get $O(n)$ at worst.
- Deletion is easy: $O(1)$.

This has best case $O(1)$, when we hit on the right value first try, but that isn't a very common case. Best cases tend to be less important to searching than they are to sorting.

This is good for finding a word in a dictionary.

A simple extension of this is the *bucket sort*. This divides (for example) names into 26 buckets (A-Z) and searches within each bucket.

4.17 Tree

Binary tree is like a binary search but we use the tree structure to do the work for us.

The tree should be such that values in the left subtree are less than the root value, which is itself less then the values in the right subtree (and recursively for subtrees). See figure 22.

Function **treesearch**. Look for a value v in a tree.

- if the tree is empty, return "not found"
- if the value at the root is v, return it.
- if the value at the root is bigger than v, return the result of searching the left subtree using **treesearch**
- else return the result of searching the right subtree using **treesearch**

Example. Determine if 5 is in the collection 1 3 5 6 7 9 11 12 15. See figure 38.

Example. Determine if 8 is in the same collection.

The tree must be balanced to get a good complexity. As a balanced tree has depth $O(\log n)$ the search can take no more than $O(\log n)$ time. An unbalanced tree degenerates to a linear $O(n)$ time.

- Time: $O(\log n)$ for a balanced tree (e.g., AVL). $O(n)$ for an unbalanced tree.
- We implicitly get an early exit by reaching an empty subtree if we have a value bigger or smaller than every value in the current tree.
- Requires data to be sorted in a balanced binary tree.
- Insertion and deletion of items is $O(\log n)$ as we need to rebalance the tree in each case.

It could be argued that we need to include the time taken to create the tree in the first place. This could be $O(n \log n)$, say, taking the total search time up to $O(n \log n)$. In real applications we tend to create the tree just once and then search it very many times. This mean that the time to create the tree is less significant than the time taken to do the searches, so the important measure is the $O(\log n)$. Think of a telephone directory: it takes a while to compile, but we cumulatively spend much more time searching through it.

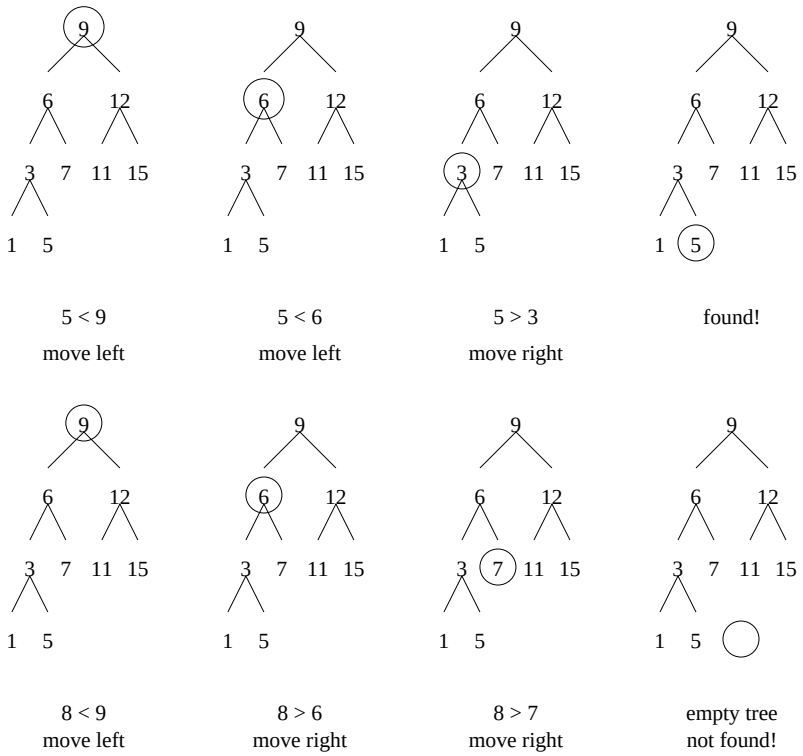

Figure 38: Tree Search

Figure 39: Indexing into an array

4.18 Hash

This is a generalisation of an array where we can index by arbitrary object, not just an integer. Suppose we wish to use people's names as the keys to a datafile. Thus we could have names like "Alice", "Bob", "Charlie", "Dave", "Eve". The file might contain the ages of these people. The problem is to get from a name to a value from the file.

One way to do this would be to put ("Alice",0), ("Bob",1), ("Charlie",2), and so on into a database (perhaps a tree). If we need to look up "Charlie" we can search the tree and get the index 2. Then we can use 2 to index into the ages array. This could take time $O(\log n)$ which is relatively fast. But we can do much better.

The idea of a *hash function* is it is a function that takes some arbitrary or complex form of data, the *key* (such as a name), and returns a number. The number can then be used to index into some datastructure, such as an array.

$$\text{key} \xrightarrow{\text{hash}} \text{integer} \xrightarrow{\text{index}} \text{record}$$

Given the names above we might take the function `name[0] - 'A'` which returns 0 for Alice, 1 for Bob, and so on. `'A'` is the value of the character "A" which is 65 in ASCII. Real hash functions are never so simple.

More realistic might be

```
hash = 0;
for (i = 0; name[i]; i++) hash += name[i];
```

which adds all the characters in the name (typically, their ASCII values). This will distinguish "Bill" (66 + 105 + 108 + 108 = 387) and "Ben" (66 + 101 + 110 = 277) . There are two problems:

- the array will be of a limited size, and the values we get from this hash may well be much larger. We might just have a table of 100 people, but this hash function returns much larger values. The simplest solution is to take the hash modulo the table size:

  ```
  int hashname(char name[], int tablesize)
  {
    int hash = 0;
    int i;

    for (i = 0; name[i]; i++) hash += name[i];
    return hash % tablesize;
  }
  ```

  That is, the remainder after dividing by the table size. This ensures the hash value will be in range. Many other solutions are possible.

- The second problem is *collisions*. The hash function may produce the same value for different names: "tabitha" and "habitat". We might respond by making the function more complex

```
hash = 0;
for (i = 0; name[i]; i++) hash += i*name[i];
return hash % tablesize;
```

(probably a bad hash, as it ignores the first character in the name). This will fix anagrams, but there will always be names that map onto the same values: there are simply more possibilities for names than can fit into a given fixed-size array, so we must almost always get a collision. We can make the hash function more complicated, but we are then sacrificing the simplicity and speed of the hash concept. The balance is a hash that is fast, but produces few collisions.

A *perfect hash* is a function that never produces a collision for our given set of index keys. Perfect hashes exist in certain circumstances, but are very rare and hard to find. The Alice and Bob example above is such a case.

Collisions are what make hash tables interesting theoretically. Here are two popular approaches, but there are hundreds of others.

4.18.1 Hash Chaining

Also called *open hashing*.

In the array we keep a linked list of records whose keys hashed to that index. If "Bill" and "Ben" both hash to 2, say, then the array will have both records in a linked list off index 2. To find the age of Ben we compute the hash 2, and search down the chain until we get to Ben's record.

Note we must keep the key ("Bill", "Ben", etc.) in the record as well as the data we are looking for so we can check against it in this search.

Ideally we want to keep the chains short so that we don't start doing long linear searching down them. This is achieved by having a good hash function that spreads the keys well across the array.

This method is easy to implement, is a little forgiving of a poor hash function (one that produces more collisions than we would want), and can accommodate large amounts of data relatively compactly. If a record is not present this is easy to determine: it will be absent from the chain. Deleting a record is easy, too: take it out of the chain.

This method degrades gracefully as the table fills and is fairly tolerant of a nearly-full table. It is best suited to languages that support linked lists and so was not popular in the early days of Fortran and Cobol! Instead, implementations in these languages tended to use a different technique, as follows.

4.18.2 Hash with search

Also called *linear probing*.

In this approach we keep the records in the array directly. If "Bill" and "Ben" collide at index 2 we put one of them at index 2, and the other at the next free space. When searching for Ben we start at the hash 2 and keep looking until we find Ben's record. We might wrap around at the end of the array. Again, we hope for a short search to find our record.

A record can be displaced from its natural position if something is already filling the slot. Suppose "Eve" hashes to 3, but "Bill" and "Ben" have previously been entered into the table. Then "Eve" will be moved along. On the other hand, if "Eve" was entered first, it would get position 3 and "Ben" would be pushed along. This is very order-dependent.

If the record is absent that is indicated by finding a gap in the table during the linear search. If the record were in the table, we would hit it before a gap. Removing records from the table is problematic as it would leave a gap which would break the above rule. Some people use a mark on the record which indicates "not present but this index is

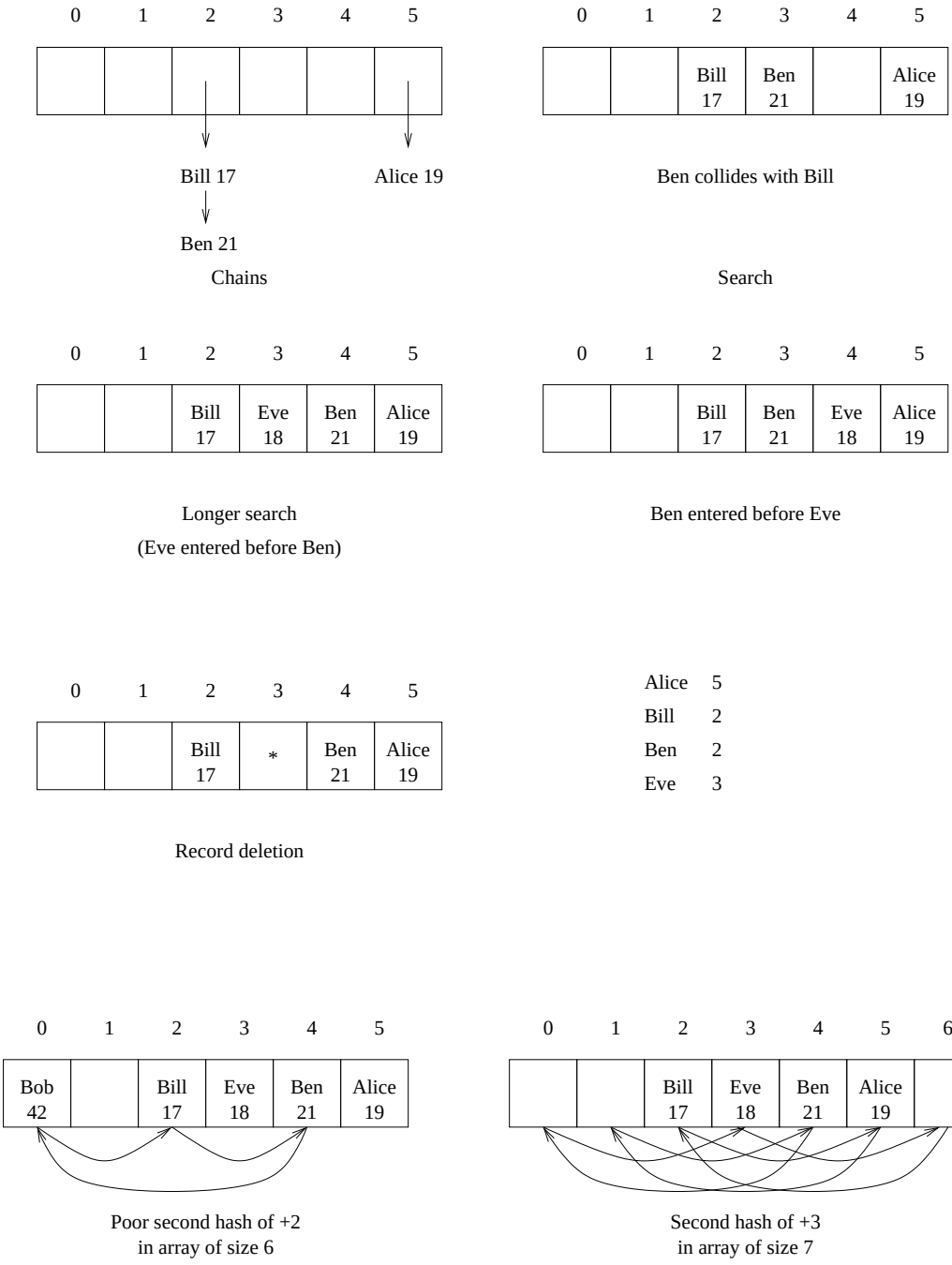

Figure 40: Hashing

occupied: keep searching". Such a record must be passed over in a search: it can be overwritten when adding a new record.

Again, ideally, we don't want to do the linear searches very far. This usually requires a table with lots of space: much more than we need to store the records. This helps against collision of the hash function. Some people recommend *double hashing*. With this, if there is a collision on the usual hash we use a second, different, hash function and try that instead. We revert to linear search if that collides, too. The idea (hope?) is that it is unlikely that two unrelated functions would both collide.

Another variant on double hashing is to use the second hash as the step size in the linear search. Thus, rather that stepping one-by-one we make big steps whose size is regulated by the second hash function. With this method we need to be more careful of our second hash function as it must return a value coprime to the array size. Otherwise we might loop back to where we started without passing through all the array values first. The easiest way to ensure this is to make the array size a prime number.

4.18.3 Generalities

If a table gets too full we spend most of our time in the linear search. Thus the behaviour with a nearly-full table can be quite poor (i.e., slow). In this case we can *rehash*. This means making a new table, perhaps twice the size or more of the old one, and re-renter our records in the new table, recomputing indices from the new hash function as we go. This table will have more gaps and a lower chance of collisions (until it fills, too!). On average we will have fast access, but once in a while it will be slow to insert a new record when we rehash.

Is chaining or searching better? There's not much to choose in practice. Perhaps chains are better if records are deleted often; on the other hand searching is easier to implement and is just as good at lookup as chaining if removals don't happen a lot.

Given a perfect hash:

- Time: best $O(1)$, average $O(1)$, worst $O(n)$, but we can avoid the worst case by rehashing when appropriate. Thus hashing is generally regarded as $O(1)$. This means the size of the table is irrelevant: it always takes the same amount of time to find something be there 10 or 10 million items of data!
- Space $O(n)$. But that can be several multiples of n.
- Other criteria: requires a good hash function, which can be hard and depends on the statistics of the keys. E.g., store 1000 integers all in the range 1 to 1000000 in an array not much larger than 1000. If the integers are random, we might choose remainder mod 1000. If the numbers are like 10001, 20001, 30001, and so on, this would be poor. Perhaps use remainder mod 999?
- Insertion of a new value is generally $O(1)$ but can be $O(n)$ if we need to rehash.
- Deletion is $O(1)$.

Problems: collisions.

Hash tables are used everywhere in programs where there are irregular data. They can improve performance of programs by many orders of magnitude when used correctly.

4.19 Space and Time Tradeoff

A common occurrence in algorithms. We often find space (memory) and time can be traded off against each other. We might be able to solve a problem using a certain amount of space and time. By choosing variations on the algorithm or using different algorithms we might be able to get a faster solution, but at the cost of using more space. Similarly, we might be able to use less space, but at a cost of more time.

This seems to be a strange property of this universe that is widely applicable.

Example. Hashing. If we have a lot of space and use a big array, we get few collisions and so a fast search. If we have limited space and a small array, we get more collisions and so a slower search.

Example. Merging two arrays of sorted integers. We can either use a fresh array to merge into (more space, less time), or merge in-place by shuffling up the array for every insertion (more time, less space).

Example. Doing repeated computations, say weather forecasting. We can save and reuse certain computations (space) to make the forecast faster (time). Or we can redo computations (time) to save some space.

Maple: criterion for choice of algorithms is to minimise space^2 $\times$ time.

4.20 String Matching

This is another kind of searching: finding where or whether some string is a subset of another. String matching is another topic where the "obvious way" to do things is not the best.

Suppose we want to look for `sat` in the string `the sad cat sat on the mat`. The `sat` is called a *pattern*, and the other string is the *text* or the *target*.

Note real life examples are *much* larger than this. If we are looking for genomes in DNA then the text can be millions of symbols long. Even simple searches for words within an email or a document are much larger.

The way most people try to so this is by the brute force method of looking through the text one character at a time:

Function **stringsearch**. Look for a pattern P in a text T. Let P have length m and T length n.

- for $i = 0$, $j = 0$ while $i < m$ and $j < n$ (i says how far along the pattern, j says how far along the text)
- if $P[i] = T[j]$ then
- $i = i + 1$, $j = j + 1$ (move along both pattern and text)
- else
- $j = j - i + 1$, $i = 0$ (reset in text and reset pattern)
- if $i = m$ then return "found at $j - m$" else return "not found"

We return the position in the string if the pattern is found.

```
the sad cat sat on the mat   the sad cat sat on the mat   the sad cat sat on the mat
sat                           sat                           sat
^                             ^                             ^

the sad cat sat on the mat   the sad cat sat on the mat   the sad cat sat on the mat
   sat                           sat                           sat
```

```
   ^                              ^                              ^

the sad cat sat on the mat   the sad cat sat on the mat   the sad cat sat on the mat
    sat                           sat                           sat
      ^                             ^                             ^

the sad cat sat on the mat   the sad cat sat on the mat   the sad cat sat on the mat
       sat                           sat                           sat
        ^                             ^                             ^

the sad cat sat on the mat   the sad cat sat on the mat   the sad cat sat on the mat
         sat                           sat                           sat
          ^                             ^                             ^

the sad cat sat on the mat   the sad cat sat on the mat
           sat                           sat
            ^                             ^
```

This algorithm has complexity $O(nm)$: consider the pattern `aaaaaab` (length m) matching the text `aaaa...aaaab` (length (n) where we have to go through every character in the pattern for every character in the text.

It turns out there are *much* better methods with complexity $O(n + m)$ or better.

4.21 Boyer-Moore

When we are matching `sat` against `sad` we get as far as the `a` before noting the `d` is wrong. We then move on and try the `s` in `sat` against the `a` in the text. Boyer-Moore recognised that *we already know that there is an* `a` *in the text next*, as we had already successfully matched it against the `a` in `sat`, so there's no point trying to match the `s` of `sat` against it. Thus we can skip more than one place before resuming testing against the `s`.

```
the sad cat sat on the mat   the sad cat sat on the mat
    sat                           sat
      ^                             ^
```

We can do even better if we match the pattern right-to-left. Consider the pattern `yabbadabbado`. If we match `do` then fail against the `a`, we can skip 12 characters in the text as we know there is an `o` in the text, but no other `o` in the pattern.

```
How can I do this strange thing   How can I do this strange thing
yabbadabbado                      yabbadabbado
          ^                                 ^

How can I do this strange thing   How can I do this strange thing
yabbadabbado                                yabbadabbado
         ^                                            ^

How can I do this strange thing
            yabbadabbado
                      ^
```

A further optimisation computes a skip table for the characters in the text, too: as `g` is not in the pattern we can skip the remainder of the pattern when we see one.

```
How can I do this strange thing   How can I do this strange thing
            yabbadabbado                                  yabbadabbado
                        ^                                             ^
```

So we often end up not even looking at some of the characters in the text!

In general we must preprocess the pattern to find out how many characters we can skip at each mismatch. Having done this we can proceed with the matching.

For the pattern `yoghourt`:

a	skip	8
b	skip	8
	...	
o	skip	3
	...	
y	skip	7
z	skip	8

BM has worst case complexity $O(n+m)$, but in real cases it tends to be about $O(n/m)$. So as the pattern gets longer, the algorithm runs *faster*! This is another space-time tradeoff. By use of a bit more space (the tables), we can make searching a lot faster (time).

String matching is just a small part of a much larger problem that of *pattern matching*. Here the pattern is more general and we are not merely matching character-for-character. A popular pattern system is called *regular expressions* where:

- a dot `.` matches any single character: thus the pattern `c.t` matches `cat`, `cut`, `cot` (and `cxt`, `c9t`, `c t` and so on)
- a bar `|` means *or*: thus `c|mat` matches both `cat` and `mat`
- a star `*` is a modifier: `<pattern>*` matches zero or more occurrences of `<pattern>`. Thus `c*` matches zero or more `c`s in a row; `c.*t` matches zero or more of any characters starting with a `c` and ending with a `t`
- a plus `+`: as `*` but one or more
- parentheses `()` for grouping: `(cat|mat)+ty` matches `catty`, `catcatty`, `matty`, `matcatty` and so on
- and several more.

Regular expressions are very powerful, very useful and very easy to get wrong! The Unix function `egrep` will match for patterns in a file.

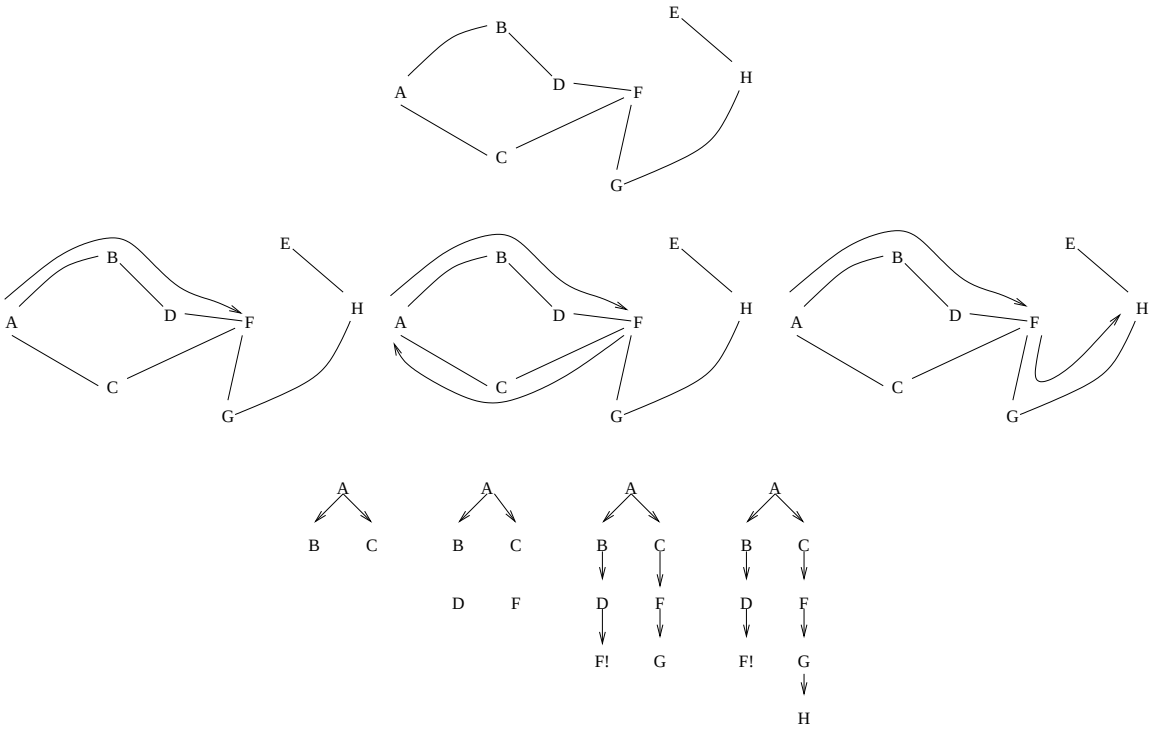

Figure 41: Searching for a Path

4.22 Depth and Breadth First Searches

There are other kinds of search. If we are playing chess we need to find the next best sequence of moves. Typically this involves searching through the many possible sequences of moves and choosing the best, where "best" is judged according to quite complex criteria.

More simply, suppose need to discover if there is a path from A to H in figure 41.

Just as for traversing trees, we can look *depth first* and *breadth first*. The added complication of this not being a tree but a *graph* is not too great: we just note if we have been to a node before and don't follow it again if we have.

4.22.1 Depth First

At each stage follow on a sequence of nodes. We get

A, B, D, F, C, A (done A, backtrack to F, where we last made a choice), G, H.

Thus we get the path A, B, D, F, G, H.

Depth first will quickly find us a path, if one exists.

4.22.2 Breadth First

We move down "layers" at a time, building up several paths at once. This does nodes of depth 1; then depth of 2; then depth of 3, and so on. We get

A, B; A, C; A, B, D; A, C, F; A, B, D, F (done F already); A, C, F, G; A, C, F, G, H.

Thus we get the path A, C, F, G, H.

Note that the path we get from breadth first is shorter, but we had to work harder to find it. This is generally true. The second time we come across a node *must* be from a path that is at best equal in length to the first one, so breadth first will always find a shortest path.

- Depth first is simple and will find a path quickly
- Breadth first is harder, but will find the *shortest* path.

5 Greedy Algorithms

To finish, we are going to look at another class of algorithm. We have seen

- brute force
- divide and conquer

being applied to searching and sorting. Greedy algorithms are best applied to *optimisation* problems: these are problems that try to find the best or cheapest or most-est whatever way of doing something.

Finding the shortest route from start to finish is an optimisation problem. Finding the fewest number of coins to make change is an optimisation problem.

The philosophy behind greedy algorithms is to build a solution step by step and

> at each step pick the best currently available option

meaning the biggest, shortest, fastest, cheapest or whatever you are trying to optimise.

Greedy algorithms do not (in general) care about future stages, so a greedy algorithm would not be too good to play chess, say. Indeed, there are many situations that greedy algorithms are not good at all: on the other hand there are many where greed is good.

5.1 Coin Changing

A simple example is making change: suppose we wish to make up an amount A using coins of values 1, 2, 5, 10, 20, 50, 100 and 200. But we want to use the fewest coins possible. What approach should we take?

A greedy algorithm says make up the change one coin at a time and at each stage pick the biggest coin. For example, if we want to make 123 we choose

- 100 leaving 23
- 20 leaving 3
- 2 leaving 1
- 1 leaving 0

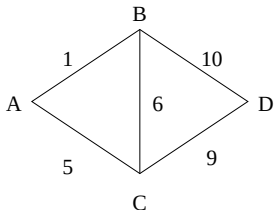

Figure 42: Shortest Paths

Thus we get $100 + 20 + 2 + 1 = 123$.

In certain circumstances, this gives the optimal (fewest) number of coins, but not always, depending on the denominations we have. Suppose we had coins 1, 6 and 10 and wanted to make change of 12. Greedily we choose

- 10 leaving 2
- 1 leaving 1
- 1 leaving 0

and so $10 + 1 + 1 = 12$. But $6 + 6 = 12$ uses fewer coins.

If we have coins $\{d_1 = 1, d_2, \ldots, d_n\}$ then the greedy algorithm is optimal when $d_i | d_{i+1}$ for all i.

This is not necessary, though. The UK coinage *is* optimal. This is because for each case where $d_i \not| d_{i+1}$ we can patch things up with a small number of lesser coins. For example $2 \not| 5$ but $2 + 2 + 2 = 5 + 1$ and two coins is better than three. Similarly for 20 and 50. If we had coins 1, 6, 10 this fails: $6 + 6 = 10 + 1 + 1$ and three coins in *worse* than two.

This algorithm takes time (approximately) $O(n)$ where n is the amount we are trying to find change for.

A brute force algorithm to do the same would take $O(c^n)$ time, where c is the number of types of coins we have, by choosing all possible combinations of coins that add up to the right amount and picking the best solution.

Pre-decimal coinage was 1 (penny), 3 (thrupenny), 6 (tanner), 12 (shilling or bob), 24 (florin), 30 (half a crown), 60 (crown), and 240 (sovereign). This is ignoring the fractions 1/4 (farthing) and 1/2 (ha'penny). There were 240 pennies to the pound.

Greedy fails for $48 = 30 + 12 + 6 = 24 + 24$.

5.2 Others

Another example is the problem of shortest routes between cities. Suppose we want to get from town A to town D and the distances between towns are as shown.

A simple greedy algorithm would pick the shortest road at each stage. We also avoid a road if it makes a loop. Start at A.

- take AB, length 1
- take BC, length 6
- take CD, length 9 (taking the shorter road CA would make a loop)

Total length $1 + 6 + 9 = 16$. However, choosing AB then BD has length $1 + 10 = 11$.

Dijkstra's algorithm to find shortest paths is more subtle. It keeps a list of *all* shortest paths and adds the road that makes the *overall* new shortest path.

Start with the shortest path from start to start, which is of length 0.

- for all the roads that extend a shortest path, pick the one that gives the shortest path.

For our example:

- shortest path from A to A is of length 0

  A 0

- Extend the path

  AB 1
  AC 5

  shortest is AB. We now have shortest paths

  A 0
  AB 1

- Extend the paths

  AC 5
  ABC 7
  ABD 11

  Shortest AC. We now have

  A 0
  AB 1
  AC 5

- Extend the paths

  ABD 11
  ACD 14

  No need to consider ABC and ACB as we already have the shortest paths to these two. Shortest is ABD.

  We get

  A 0
  AB 1
  AC 5
  ABD 11

  So the shortest path A to D is ABD.

This grows the shortest paths and at each stage has computed the shortest path to each city.

Dijstra's algorithm will always find the shortest path. Sometimes greedy algorithms give a non-optimal solution, but a good approximation to it.

For example, when packing a case a good strategy is to put the largest things in first and fit smaller things in afterwards. This may not give you the best possible packing, but it will (apart from certain specially constructed pathological cases (pun intended)) give you a good fit that is probably quite close to the optimum.