```
1  int search(int a[],int n, int val)
2  {
3
4    int i;
5
6    for(i=0;i<n;i++)
7      {
8        if(a[i]==val)
9          return i;
10     }
11
12   return -1;
13 }
```

Table 1: Linear search. This function searches the entries in the array a and returns the index when it finds val, if it doesn't find val it returns -1. The program `linear_search.c` implements this..

4 Search: linear and binary search example

Lets do a quick example of working out big oh: searching for the index of an element in a sorted list. A completely terrible way to do this is linear search, this is terrible because it doesn't make use of the fact that the list is sorted. A code listing is given in Table 1. We can see straight away that the code between lines 6 and 10 is run n times in the worst case, everything else is run once and so this algorithm is $O(n)$.

A much better way to search a sorted array is binary search. This is an example of a 'divide and conquer' algorithm, many of the fastest algorithms use divide and conquer. It will be clear to you that this algorithm would be better written using recursion, this is typical of divide and conquer, but we haven't looked at analysing recursion yet. The idea is to divide the array in half and check which half, the half with bigger numbers or the half with smaller numbers, the value we are searching for belongs to and to keep doing this, dividing the remaining part of the array into two parts again and again until the remaining part of the array that is being searched has only one element. A code listing is given in Table 2.

This search is extremely fast. There is a chance that a[mid]==val, after a small number of iterations, indeed, if the middle value of the array is the search value it will halt after only one iteration. However, as usual, we assume the worst case, in which case the algorithm runs to end, dividing the number of elements in half each time. Ignoring the integer rounding effects, it goes like Table 3. Starting with n states each subsequent iteration halves the number of states until the last one when there is one state left. Thus

$$1 = \frac{n}{2^{T(n)-1}} \tag{1}$$

and taking the log of both sides

$$0 = \log_2 n - (T(n) - 1) \tag{2}$$

using $\log_2 1 = 1$, $\log_2 a^b = b \log_2 a$ and $\log_2 2 = 1$. Hence

$$T(n) = \log_2 n + 1 \tag{3}$$

```
1  int search(int a[],int n, int val)
2  {
3    int mid, low=0, high=n-1;
4
5    while(low<=high){
6         mid=(low+high)/2;
7         if(a[mid]==val)
8           return mid;
9         else if(val>a[mid])
10          low=mid+1;
11        else
12          high=mid -1;
13      }
14
15   return -1;
16 }
```

Table 2: Binary search. This function starts in the middle of the array and checks if the value there is bigger or smaller than val, if it is bigger then it does the same in the top half of the array, if it is smaller, in the bottom half and then repeats until there are no elements left. The program `binary_search.c` implements this.

1	2	3	4	...	k	...	T(n)
n	$\frac{n}{2}$	$\frac{n}{4}$	$\frac{n}{8}$	...	$\frac{n}{2^{k-1}}$	...	1

Table 3: The number of elements left for binary search.

and this algorithm is $O(\log_2 n)$.

So, to reiterate; the usual way to examine the behavior of an algorithm is to look at the worst case run time. This is because the best case run time is often exceptional, like the one for binary search if the first guess happens to be correct. The average run time is often hard to calculate, both because it is often difficult to do the mathematics and because it would often mean having some description of how the initial data is distributed. Typically the worst run time is also 'of the same order' as the average run time. We will see an exception to this later on in the case of quick sort in which the worst case behavior is unusual. The big-Oh notation is used for describing an algorithm, if the algorithm is said to be $O(g(n))$ we mean $T(n) \in O(g(n))$ no matter what the initial condition. Since $O(g(n))$ involves an upper bound $T(n) < cg(n)$ this makes sense.