1. Introduction

This course is about algorithms and data structures; roughly speaking, an algorithm is a clearly defined list of manipulations which can be applied to data to achieve an objective and a data structure is a way to store information. The word algorithm comes from the name of the great ninth century Persian astronomer and mathematician al-Khwarizmi.

We've always had algorithms, counting beads for tallying, the Babylonian algorithm for solving quadaratic equations, the Chinese remainder theorem for solving congruence equations; for much of history the idea of an algorithm has been a part of applied mathematics, a method for working something out. Now, with computers the study of algorithms has become a seperate discipline, with abstract questions about computibility and practical ones about how to evaluate algorithms and chose between available algorithms for a particular application.

Algorithms are the part of a programming that don't generally depend on the hardware or programming language. This isn't to say that some programming lanaguages don't express some algorithms for naturally or more easily and that some compilers aren't better or worse at optimizing certain algorithms and there is lots of variability, for example, in how well recursion runs from language to language, compiler to compiler and chip to chip. However, the aim in examining algorithms is to do so in a platform independent way. This isn't to say we won't use a platform, examples will be given in C and you will be encouraged to try out the algorithms using your knowledge of C and Haskell programming.

1.1 Sorting - Insertion Sort

Discussions of algorithms tend to involve lots of discussions of sorting. This is partly because sorting is a big issue, search and sort algorithms are important in the analysis of data, be it scientific data or web addresses and data analysis is at the heart of contemporary technological revolution. The other part is that sorting is a particularly good illustrative example.

Consider sorting a pack of card, using bridge ordering, clubs, diamonds, hearts, spades, the goal is to order the cards so the ace of clubs is at the bottom and the king of spades at the other. To make it more straight forward, imagine they are numbered one, for the ace of clubs, through to 52 for the king of spades, that makes it less confusing to talk about one card having a value less than another.

The obvious way to sort is probably the algorithm known as *insertion sort*. Take the unsorted deck, the cards will be added one by one to another deck, the sorted deck, with, for the sake of definiteness, the lowest card at the bottom, the highest on top and all the others in order in between. First, take the first card in the pack and put it in the other deck, the sorted deck now has exactly one card in it. Now take the second card in the unsorted deck and put it above or below the first according to whether it has a higher or lower value. After that, keep going, take a card from the unsorted deck, search through the sorted deck from the top until the next card has a smaller value than it and put it in above that card. When all the cards are gone from the unsorted deck, the algorithm is complete. An example of insertion sort is given in Table **??**.

0.0.1 Implementation and run speed

A C function for sorting is given in Table **??**; this is a function, to see it in action download `insert_sort.c`. Now, the question is, how good an algorithm is this. Of course, 'how good'

1		4	2	0	1	3
2	4		2	0	1	3
3	4	2		0	1	3
4	2	4		0	1	3
5	2	4	0		1	3
6	2	0	4		1	3
7	0	2	4		1	3
8	0	2	4	1		3
9	0	2	1	4		3
10	0	1	2	4		3
11	0	1	2	4	3	
12	0	1	2	3	4	

Table 1: An insertion sort example. A space divides the sorted deck from the unsorted deck; in practice this isn't needed, the sorted and unsorted elements can be stored in the same array with an index or pointer used to remember where one stops and the other starts. Consider, for example, steps **7** to **10** where the value 1 is moved to the sorted deck and then compared with each of the sorted values until it reaches the 0, since the 0 is a lower value than 1 the 1 isn't swapped with it and the algorithm goes on to move the 3 into the sorted deck.

could mean lots of different things, depending on what constraints there are on the use of the program, it could mean the size of the program itself, or the amount of memory the program needs for data storage, but most frequently, it means run time.

Lets examine the run time for the function in Table **??**. Let n be the number of elements in the list. The amount of time each line takes to run is platform dependent, so lets say the run time for the ith line is t_i:

$$t_i = \text{time it takes line } i \text{ to run once} \tag{1}$$

Lets call the total run time $T(n)$ since it depends on n, the number of elements that need sorting and the contribution to $T(n)$ of the ith line T_i,

$$T_i = \text{the total time the program spends on line } i \tag{2}$$

Now $T_i \neq t_i$ in general because some lines run more than once, here, for example, $T_4 = t_4$ since the fourth line only runs once, but $T_7 = (n-1)t_7$, once for each value of i. T_6 is a bit more complicated in an uninteresting way since the end condition $i < n$ is checked n times, the initial condition $i = 1$ only happens once and the increment happens $n - 2$ times, but lets approximate and say $T_6 = nt_6$.

Lines **11** and **12** are complicated in a more interesting way, we don't know how many times `j` needs to be reduced until the while condition returns false, it depends on the value of `a[i]` and the values of the elements already sorted. Obviously, if the list is already sorted then the while condition fails each time and these lines are never run; this is an exceptional case however. Conversely, in the worst case the sequence starts of in reverse order, in which case lines **11** and **12** run i times for each value of $i = 1$ to $i = n - 1$, thus

$$T_{11} = \sum_{i=1}^{n-1} it_{11} = \frac{1}{2}n(n-1)t_{11} \tag{3}$$

```
1  void sort(int a[],int n)
2  {
3
4      int i,j,this_a;
5
6      for(i=1;i<n;i++){
7          this_a=a[i];
8          j=i-1;
9
10         while(j>=0&&this_a<a[j]){
11             a[j+1]=a[j];
12             j=j-1;
13         }
14
15         a[j+1]=this_a;
16     }
17 }
```

Table 2: A function for sorting an array. This function can be found included in the program `insert_sort.c`. It takes as an argument an array of ints called `a[]` and `n`, the size of the array and sorts it using insert sort. The array is sorted from `a[0]` and the sorted elements are stored in the same array, with i marking at each iteration the last element in the sorted deck, the while loop moves the element that was `a[i]` into the correct place.

So there is formula for summing all the numbers from one to n:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{5}$$

This can be proved by induction. If $n = 1$ it holds with one on each side of the equation. Now assume it holds for $n - 1$ and consider

$$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i = n + \frac{1}{2}n(n-1) = \frac{n(n+1)}{2} \tag{6}$$

proving the formula by induction.

Table 3: Mathematical aside: the formula for adding numbers.

The formula used to calculate this is explained in Table **??**. Line **10** is a bit like line **6**, it is run one extra time so

$$T_{10} = \sum_{i=1}^{n-1}(i+1)t_{10} = \frac{1}{2}(n+1)nt_{10} \tag{4}$$

All these precise sums are being calculated this time for illustrative reasons, in fact, the point we will make is that the only thing that is really of interest is the power of the highest power of n and detailed calculations like this won't be done again.

Now we can calculate $T(n)$ for the worst case.

$$\begin{aligned} T(n) &= t_1 + t_4 + nt_6 + (n-1)(t_7 + t_8 + t_{15}) \\ &\quad + \frac{1}{2}(n+1)nt_{10} + \frac{1}{2}n(n-1)(t_{11} + t_{12}) \end{aligned} \tag{7}$$

Gathering everything together this has the form

$$T(n) = an^2 + bn + c \tag{8}$$

where, for example $a = (t_{10} + t_{11} + t_{12})/2$. What we see is that the coefficients, a, b and c are complicated, the depend on the run times of individual lines of code, the balance between the n^2 and n terms is similarly hard to evaluate. In short, the most interesting and definite platform independent statement we can make about $T(n)$ is that it grows like n^2 as n becomes large. We say

$$T(n) \in O(n^2) \tag{9}$$

The n^2, the fastest growing term in the expression for T_n is sometimes called the leading term. This will be discussed further, but first have a look at Fig. **??** where the run time of the program is plotted against n for $n \in (1, 1000)$. We see that the curve is extremely well matched by n^2, the sub-leading order terms, the n term and the constant, aren't important.

Figure 1: Run time for sort. The program was run 500 times for each value of n and the y-axis gives the total run time in seconds on my laptop, while I was also doing other things like typing this document and playing music videos on youtube. The code for doing the timing test is `insert_sort_timing.c`. The n^2 behavior is clearly visible and the curve $0.88(n/1000)^2$ is plotted too, this was fitted by eye but is so good a fit it is hard to see it behind the run time data.