What are Algorithms

This section is a very gentle introduction to our first algorithm and it's analysis.

1 What are Algorithms?

You may have heard the word *algorithm* used in other courses, earlier in your education, perhaps used in everyday language (often incorrectly), and even referenced in TV shows when one wants to allude to the fact that something is highly complex. There is nothing particularly mysterious about this word "*algorithm*". In computer science, an **algorithm** is simply a *procedure* that takes some **input** and then produces some required **output**. Algorithms existed before computers did, in fact for thousands of years we have had procedures that carry out tasks or computations. In the middle ages, we used a procedure using jugs of two different capacities to measure any given quantity of water. In ancient times, we had quite complex algorithms for carrying out multiplication and fraction representation. The only difference between these ancient algorithms and those of today, is that now we use computers to carry them out. Today, algorithms lie at the heart of modern computing: they are behind your apps that drive you to work, they secure out your on-line banking, they are what allow you to pay online safely with your credit card, and even how you find friends on Facebook.

If you want to properly describe an algorithm, you need to first know what its *job* is: what is the problem that needs to be solved. This involves defining some relationship between the input and the desired output. For example, we could define a very simple problem of finding the average of n numbers. Such a problem has the following input and output:

The average-finding problem:

Input:. A set of n numbers, $a_1, a_2, \ldots, a_n$

Output: The average of the numbers, $\frac{1}{n}(a_1 + a_2 + \ldots + a_n)$

An *algorithm* for the problem above would then be a well-defined procedure for producing the required output. The algorithm is **correct** if it produces the correct output for *every possible input* (which also implies that eventually the algorithm must terminate and can't simply carry out computations forever). This means that whatever set of n numbers are provided as input, the algorithm correctly computes the average and returns it as output.

Describing an average-finding algorithm seems rather trivial - even if you know nothing about algorithms. The procedure should simply sum the numbers together and then divide by n. We write this basic algorithm below in *psuedo-code*:

The average-finding algorithm:

$sum = 0$
for $i = 1$ to n **do**
  $sum = sum + a_i$
end for
return $\frac{sum}{n}$

This algorithm simply sums up the n numbers using a for loop, and then divides by n. Notice however that we have assumed that whoever or whatever is carrying out this algorithm knows how to *add* and how to *divide*. Do we also have to describe those procedures? And if not, how is it that we can assume "it" knows how to add and divide but not how to find an average? Certainly if we are going to describe algorithms, we need to know what operations are possible for whatever "machine" is going to carry them

out. This leads us to the next section, where we define a model for the machine that will implement these algorithms.

2 Model of Technology

In this course, we assume that all our algorithms are carried out by a **random-access machine**. You may have heard of such a description of modern-day computers. The RAM model represents a *simplification* of real computers. The reason we use this model is so that we can properly analyze our algorithms - a task that can be quite difficult mathematically, even with this simplification. This model represents a very clear definition of what our "*machine*" can and cannot due. Once this definition is laid out, we then know what operations are allowed by our algorithms. For example, if the model allows addition, then we can assume our machine knows how to add and our algorithms can use addition. If the model doesn't know how to sort, then we need to write an algorithm to sort.

The RAM model assumes that instructions are executed one after the other, and that **the following elementary operations can be carried out in a constant amount of time**:

- Computations such as: add, subtract, multiply, divide, remainder, floor, ceiling, compare, etc.
- Access memory: Load, store, swap, look-up, copy, etc.
- Control: call a function, return output, etc.

This is certainly not an exhaustive list, and in further sections we may refer specifically to operations we consider as constant-time operations. Elementary operations such as those above are considered as a "*step*" in the RAM model, and typically we analyze algorithms by counting how many steps they take. Real computers don't work exactly like this. Memory for example is quite complex: with several layers of caches and hierarchy, and the RAM model ignores all this. Furthermore, not all mathematical operations are done in constant time: exponentiation for example is not. In the RAM model however, for small values of y, we assume we can compute x^y in constant time.

Although the RAM model is not a perfect representation of real computers, by using this simplification to analyze our algorithms we are able to get an excellent predictor of how they will behave on real machines.

3 What we analyze and why

When we talk about "*analyzing algorithms*" we are usually referring to some prediction of how that algorithm behaves: what resources it uses, and most importantly, how much time it takes. We will begin with the notion of the the **running time** of an algorithm: that is the *number of elementary (or primitive) steps* the algorithm takes. Recall that all basic or primitive operations in the RAM machine are considered to be a "step"'. Thus when we refer to the running time of an algorithm, we ignore the exact time of each basic step, and simply focus on the **number of steps**.

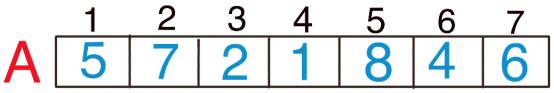
You may notice immediately in our simple example above on finding the average, that the number of steps the algorithm takes is dependent on how many numbers are given in the input. In general, the number of steps an algorithm takes *grows* with the size of the input. Therefore, we describe running times in terms of the **input size**.

By analyzing algorithms, we are able to predict how they perform on real computers. Why would this be important? If they seem to be producing the correct output, why would we not just put the algorithm to work? The first reason is quite obvious - often we have several candidate algorithms for solving a particular problem, and by analyzing each one, we are then able to select that which is the most efficient. Secondly, we shall see instances where certain algorithms perform extremely well on some specific *types* of input. This gives us the choice of preferring certain algorithms when the input is known to be of a certain type. Thirdly, we shall work through cases where some algorithms perform well *most* of the time, but sometimes

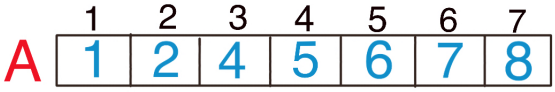
(depending on the input) perform extremely poorly - something called the **worst-case running time**. Again, this gives us the option of choosing something that is either fast *most of the time* and slow very rarely, or something else that has a more stable overall performance. These are all examples of things that need to be considered in real life applications.

4 Your first Algorithm: Insertion sort

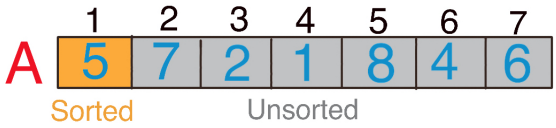
Our first algorithm aims at solving the **sorting problem**: given a list of n numbers, the goal is to sort them in increasing order. We assume that the numbers we wish to sort are listed in an array A which is indexed from 1 to n:



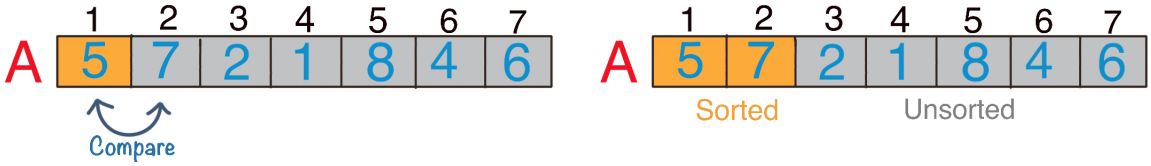
During this course we shall study many different algorithms that tackle this problem. The first one we look at here is called **insertion-sort**. It works much like we sort a hand of cards, where we typically start with a single card and as each new card arrives, we *insert* it into its proper position. Insertion-sort is similar in that it uses the notion of adding a number one at a time to the sorted section. It does this **in-place**, meaning that it uses the original array that contains the numbers and when the algorithm terminates the numbers are sorted in the original array:



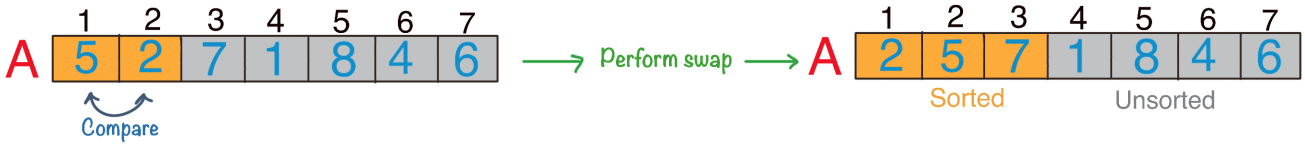
The analogy with card-sorting works as follows: suppose that the numbers on the left of the array are those that are "*already sorted*" and the numbers on the right are those that still need to be inserted into the sorted list. Before we do any work at all, we can consider the first number in the array to be "already sorted" (much like when you receive the first card, you don't need to do any work):



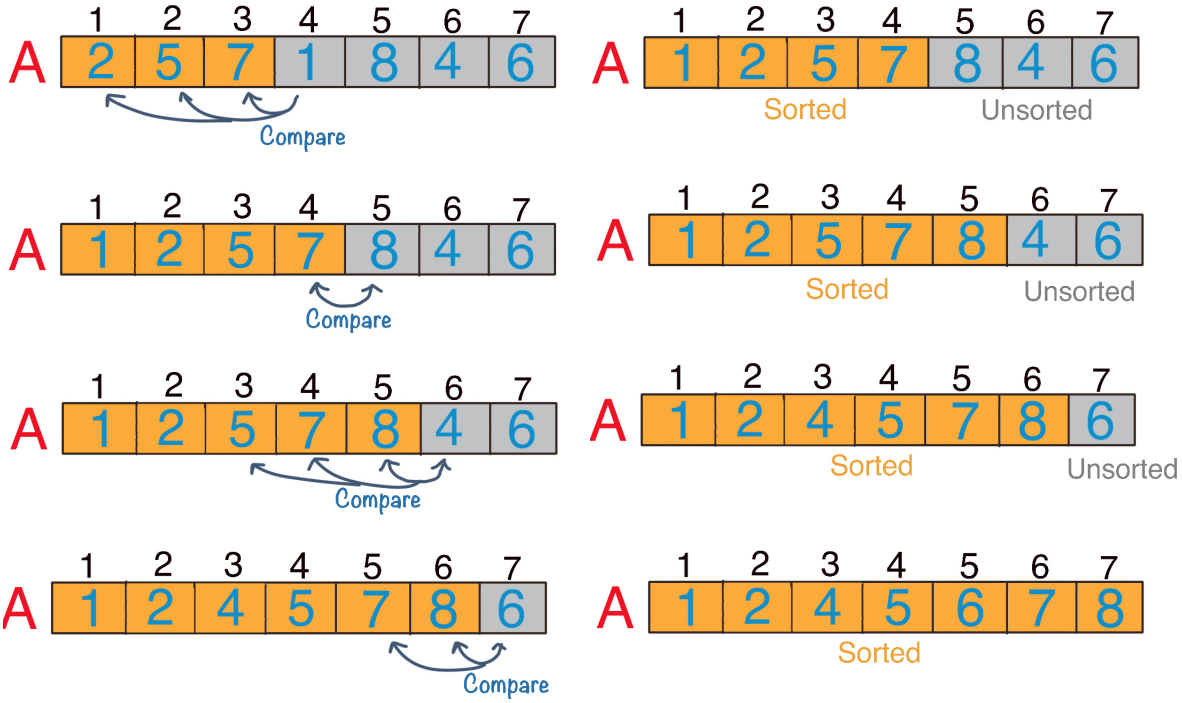
Now we need compare the second number to the first in order to determine if they need to be swapped or not. We swap them if necessary, and now the first two numbers are sorted.



The next number to be "*inserted*" is in the third position. Starting with the *right-most number* in our sorted section, we compare each number to the one that it is to be inserted, shifting any larger number to the right to give space for the new insertion. Once we find where the insertion position, we leave our new number in place. Now our sorted section is size 3.



Continuing in this way we summarize the next four steps:



Below is the basic pseudo-code for this algorithm. Note that this is slightly different than the exact version in CLRS. The execution of the pseudocode is animated in the corresponding class video. Here we assume (as in CLRS) that the input is an array of length n, indexed from 1 to n. The index i is used to partition the sorted section from the unsorted section, and index j is used to loop through the sorted section performing swaps when necessary.

```
InsertionSort(A[1...n])
    for i = 2 to n :
        for j = i down to 2:
            if A[j] < A[j − 1]
                Swap A[j] and A[j − 1]
            else break
```

4.1 Counting the number of operations

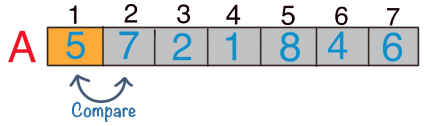
As we discussed in the previous section, the running time of this algorithm depends on the *input size*. In the above example, our array had a specific length of seven. For run-time analysis however, we assume that our general array has length n. The goal now is to determine how many steps the algorithm takes in relation to this array size, n

What kind of operations are carried out in the above description of insertion-sort? Our pseudocode above uses the following types of operations:

- Comparison of two numbers
- Swapping the position of elements in the array
- Operations for the loops, read/write/access etc.

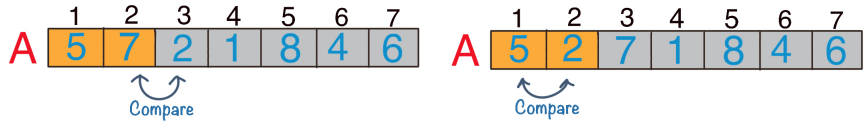
In our RAM model, comparisons and swaps can be done in a constant amount of time. The same is true for the initialization of variables and the updates necessary at each iteration of a loop. Thus we simply need to determine *how many* of each of these basic steps are executed during the algorithm. Let's start off by just focusing on counting how many comparisons are made throughout the algorithm.

- **Step 1:** In the very first step, the second number was compared to the first, accounting for a single comparison.



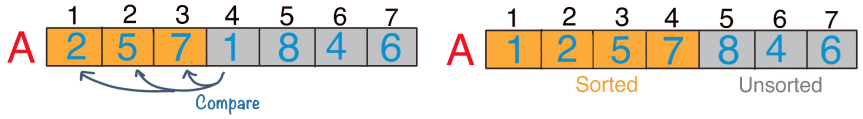
One comparison

- **Step 2:** In the second step, the third number was compared with those in the sorted list in order to find its insertion position. How many comparisons is this? It seems like this would depend on the final position of the new number. In some cases, we may have to compare the new number to *both* numbers in the sorted list - this would be the case if the third number was the *smallest* of the three, as shown in our particular example below. Or, it may be that only one comparison was needed, which would be the case if the new number was the *largest* of the three. In this analysis, we consider how many steps are used in the **worst-case** scenario, which then gives us a run-time that represents the number of steps the algorithm takes in the worst possible case. We call this an **upper bound** for the run-time and discuss this more in the next section.



Two comparisons

- The following steps work in the same way. When the fourth number needs to be inserted, we compare it to those in the sorted list until we find its insertion point. Again, in the worst case, it may be that we have to compare it to *everything* in the sorted list - meaning a total of three comparisons.



Three comparisons

- This continues until the very last step, when the nth item is inserted into the sorted list. The size of the sorted list is now $n-1$, which means that the worst case for the number of comparisons needed is $n-1$.

We are now ready to total up all these comparisons. The total is:

$$1+2+3+\ldots+(n-1)=\frac{n(n-1)}{2}=\frac{1}{2}n^2-\frac{n}{2}$$

This value gives us a direct relationship between the size of the input and the worst-case number of comparisons needed by the algorithm.

What about the number of swaps? Note that when the input is sorted in reverse order, a swap is performed after each comparison. Therefore the worst-case number of swaps is also $\frac{1}{2}n^2-\frac{n}{2}$. And the other types of operations? Similarly, the worst-case scenario is when the data is sorted in reverse order, meaning that the above loops are executed a total of $1+2+3+\ldots+(n-1)$ times, so any additional loop operations are also executed at most $\frac{1}{2}n^2-\frac{n}{2}$ times.

4.2 The Run-time of Insertion Sort

The goal now is to put all of this information together to describe the overall run-time of the algorithm. We must total the run-time of each of the operation types. Suppose a comparison takes time k, a swap takes time s, and the loop operations take time x in the worst case. We also assume some constant amount

of time, say c, for any other constant work performed during the algorithm. The actual amount of time to complete the above algorithm is :

$$k(\frac{1}{2}n^2 - \frac{n}{2}) + s(\frac{1}{2}n^2 - \frac{n}{2}) + x(\frac{1}{2}n^2 - \frac{n}{2}) + c$$

We are interested in what *type* of function this is, not the exact constants. In the above expression, we have several n^2 terms, and some n terms, and a constant term c. By collecting like terms, the resulting expression can be simplified to :

$$an^2 + bn + c$$

for some constants a, b, c. This is a **quadratic function** of n. When studying runtimes, we are interested in the overall *form* of the function, and we do not evaluate the exact values of a, b, c.

The **worst-case** runtime of Insertion Sort on input of size n is of the form

$$an^2 + bn + c$$

for some constants a, b, c.

Again, we emphasize that we *do not* care about the exact *values* of a, b, c, nor do we care about the values of k, s and x. This may seem strange, but let's take a moment to see why. Suppose we evaluate the exact speed of each operation, and determine the specific values of a, b, c on a particular computer. As an example, suppose the result is a run-time of $3n^2 + 2n + 5$. Now imagine that someone else makes the same measurements on a different computer, and found the run-time to be $6n^2 + 4n + 10$! Notice that the only difference in the above two run-times is the *constants* in front of each term. Clearly the first run-time is faster than the second. However, the exact speed of each operation is influenced by things that are out of our control, such as the speed of the particular processor, or other factors related to things like the programming language or the person who programmed the algorithm. If we spend time nailing down the exact time of a single comparison, and then determining the exact running time, then in a year or two this value will be out of date, and our answer would vary from one computer to another. What *does* remain the same, however, is that the running time is **quadratic**: a function of the form $an^2 + bn + c$ for some numbers a, b, c. And this fundamental fact is independent of external factors. Thus in algorithm analysis, we focus on the *type* of function that bounds the running time - in this case, a quadratic function of the form $an^2 + bn + c$. The particular constants in front are not relevant, since they could depend on external factors.

5 Summary

In summary, we have discussed the following important elements related to algorithm analysis under the RAM model:

- Count the *number* of constant-time operations when determining the running time of an algorithm.
- Focus on the *type* of function that describes the running time, and not the exact value of the multiplicative constants.

In our next lesson we shall develop this concept further and formally introduce our running time notation.