1.1 Introduction

Since we are going to study many algorithms and data structures for solving similar problems, it is essential that we are able to decide between them. For example, we will study five or six sorting algorithms. We must be able to reason about their running times and memory usage under differing conditions so that we can decide which one is most suited to a particular problem. If we are lucky, one algorithm and data structure may be best for all situations, but that is not always the case.
1.2 Running Times of Algorithms

We typically need to know (and can usually most easily estimate) the worst case running time of an algorithm, often called its time complexity. For Bubble Sort, the running time is $O(n^2)$, where $n$ numbers (or keys) are to be sorted in main memory. Quick Sort also has time complexity $O(n^2)$, but is preferred over Bubble Sort because its average time complexity is $O(n \log n)$. (Note that we can use “Big O” notation for the average running time as long as we make it clear in the associated text that the average is intended.) Moreover, with Quick Sort, it is possible to make the worst case an extremely rare occurrence.

Sometimes we can also estimate the best case running time. For example, given a set of $n$ integers $\{I_1, I_2, \ldots, I_n\}$, find a subset whose sum is closest to a given value, $k$. In the best case, the sum of the first $p$ values equals $k$ and an algorithm that tries these values first will finish in $\Omega(p)$ time, $p \leq n$. Since we have no way to estimate $p$, we say the running time is $\Omega(n)$.
In the worst case, it may be necessary to sum all possible subsets before finding the best solution. That’s $2^n$ subsets, so the running time is also $O(2^n)$.

We can easily prove this last result by noting that an $n$ bit binary vector can be used to indicate which values are in the current subset. Bit $j = 1$ if value $I_j$ is included in the subset. There are $2^n$ settings of this binary vector.

Further more, for each setting of the vector, we would have to add the corresponding subset of the values. If we assume the average number of values to be added is $n/2$, then we see that the total time complexity is $O(n \times 2^n)$.

A time complexity with a factor of $2^n$ is daunting. For $n = 100$, $2^{100} \approx 10^{30}$ and the age of the universe is, according to the cosmologists, about 15 billion years, which is about $10^{28}$ nanoseconds. Our computation would last 100 times the age of the universe if we were able to try one subset every nanosecond.
We have just performed a proof by construction. We started with known facts and assembled them into a result. It is often the case that we have a good idea what the resulting equation for running time looks like, but we may not be completely sure, or we may not know the values of some constants that appear in the result. In such cases, other proof techniques are used.

For example, we could disprove an incorrect equation by finding a special case in which it doesn’t work. If someone suggests that Bubble Sort runs in worst case time $O(n)$, I can easily show this cannot be true by a simple example with only $n = 4$ values that are initially positioned in reverse order.

In a similar way, proof by contradiction is a powerful technique for proving theorems.
Consider the theorem, *There is no such thing as the largest prime number.*

Assume the theorem is false, and the $k^{th}$ prime $Q = P_k$ is the largest prime number. Form a new number $R = P_1 \times P_2 \times \cdots P_k + 1$. The only prime that is a factor of $R$ is $P_1 = 1$ and so $R$ is also prime, and clearly $R > Q$.

We tried to disprove the theorem by proving the contrary, but we failed. This is how proof by contradiction works.

The primes are *countably infinite*, which means that you can put them in one to one correspondence with the integers - there is a first, a second, etc. Physicists have calculated that there are about $10^{80}$ particles in the universe (http://www.nature.com/nsu/020527/020527-16.html). They are countable, but not infinite. Points on the real line between 0.0 and 1.0 are not countable, but there are infinitely many of them. There are two kinds of infinity. The members of a set may be finite, countably infinite or infinite but not countable.
Other examples of theorems that can easily be proved by contradiction are:

- The worst case running time of Bubble Sort is more than $O(n)$.
- The best case running time of any sorting algorithm is at least $O(n)$.

But what about:

- No human being on earth is taller than 12 ft.
- God exists.
If we try to use contradiction on the first statement, we conjecture that there is someone taller than 12 ft and we have to show this isn’t true. To do so requires a search for someone amongst potentially the entire human race. If, however, you happen to know someone who is taller than 12 ft, you can disprove the theorem immediately. If there is no proof by construction (based on concrete biological theorems), it’s hard to see how to prove that the first statement is a theorem. Similar problems arise if we try to use contradiction (or any mathematical proof technique) on the second statement.

1.3 Proof by Induction

Induction proof techniques are extremely important in the analysis of algorithms. They are used when we have a good idea of the resulting equation, and need to prove that our idea is correct, or when we need to find the values of constants in our equation.
Let’s start with something easy.

\[ \sum_{i=1}^{n} i^3 = [kn(n + 1)]^2 \]

The induction method has 2 steps. First we choose a basis, or base case, such as \( n = 2 \), and insert it into the above equation and show that it is correct for that value. Then we assume that the equation is true for values of \( n \) up to some value \( m \) and show that, if this is the case, it is also true for \( m + 1 \). This is called the inductive step.

Basis: \( n = 2 \) : \( 1^3 + 2^3 = [k \times 2 \times (2 + 1)]^2 \)

The basis is ONLY true if \( k = 1/2 \). So we can assume this in what follows.
Inductive step:

\[
\sum_{i=1}^{m+1} i^3 = \left[ \frac{m(m + 1)}{2} \right]^2 + (m + 1)^3 = \left[ \frac{(m + 1)(m + 2)}{2} \right]^2 \\
= \frac{(m + 1)^2(m + 2)^2}{4} \\
= \frac{(m + 1)^2(m^2 + 4m + 4)}{4} \\
= \left[ \frac{m(m + 1)}{2} \right]^2 + (m + 1)^3
\]

By working on the right hand side we were able to show that the inductive step was successful and the equation is a theorem.
Here is another example of a proof by induction:

\[ \sum_{i=1}^{n} 2i - 1 = n^2 \]

The sum of the first \( n \) odd numbers is \( n^2 \). Clearly we can use the general result for the sum of an arithmetic progression, but let’s try induction instead.

Basis: \( n = 1 \) : \( 1 = 1^2 \).

Inductive step:

\[
\sum_{i=1}^{m+1} 2i - 1 = m^2 + 2(m + 1) - 1 = (m + 1)^2 \\
= m^2 + 2m + 1
\]
Often much simpler tricks can be used to prove theorems, especially those involving sums of series:

\[ \sum_{i=1}^{n} \frac{1}{2^i} = 1 - \frac{1}{2^n} \]

A proof by construction can be formulated simply by examining the binary value of the sum so far:

<table>
<thead>
<tr>
<th>n</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1_2</td>
</tr>
<tr>
<td>2</td>
<td>0.11_2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td>0.11111_2</td>
</tr>
</tbody>
</table>

The sum approaches 1.0 as \( n \) tends to infinity. The sum’s value is given by \( 1 - 1/2^n \).
Consider the theorem below that, at first sight, seems more difficult to prove:

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$$

Induction is not immediately useful since we are trying to prove a result that does not involve $n$. Instead, a trick is used. We call the sum $S$, and write out the first few terms. Then we write out the sum for $2S$ and subtract the two series. Terms partially cancel in the process, leaving a simpler series that we can easily sum to get the result we need:

$$S = 1 + \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots$$

$$2S = 1 + 1 + \frac{3}{4} + \frac{4}{8} + \frac{5}{16} + \cdots$$

$$2S - S = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots$$

$$S = 2.0$$
Now you can construct the equation for the finite sum version:

\[ \sum_{i=1}^{n} \frac{i}{2^i} \]

### 1.4 Recurrence Relations

We make use of recurrence relations extensively in the analysis of recursive algorithms. In some cases the algorithm may not appear recursive - it may not contain a recursive function, but its behavior is distinctly recursive in nature.

For example, Merge Sort takes an array of \( n \) values or keys and sorts them into order. It partitions the array into two approximately equal parts, sorts them separately, and then combines the two sub-arrays. If this strategy reduces the running time (over just sorting the entire array as a single unit), then we will apply it recursively to give a faster sorting algorithm.
Each sub-array that is larger than a single value is sorted by first partitioning it into two, sorting the two parts, and combining the results. It is possible to write merge sort as an iterative or a recursive function (and, in general, any recursive function can be rewritten as an iterative function).

A simple sorting algorithm is one that sorts \( n \) values in \( T(n) = C_1n^2 \) time (such as quick sort). If \( n \) is even, we can partition the array into halves and sort each half in \( T(n/2) = C_1(n/2)^2 \) time, and we can merge the two in \( C_2n \) time, making the overall time \( 2C_1(n/2)^2 + C_2n = 2T(n/2) + C_2n \), which is an improvement over the simple sorting algorithm for large \( n \). Writing this as a recurrence relation, we have:

\[
T(n) = \begin{cases} 
C_1 & : \ n = 1 \\
2T(n/2) + C_2n & : \ n > 1
\end{cases}
\]

Since we have written this as an exact relation (an equals sign is used), there is an assumption that \( n \) is a power of 2. We shall return to this point later.
The solution of a recurrence relation is a *closed form equation* for $T(n)$. A closed form equation is one in which a value of $n$ can be inserted and a result found. It does not involve recurrences, summations, or any other indirect method of computing $T(n)$.

There are three main ways of solving recurrence relations:

- Guess the result and prove it by induction.
- Expand the recurrence by repeatedly substituting it into itself.
- Use the general solution method.

The general method is similar to the general method for solving differential equations. We shall not need it in this course. We will try the first two methods.

Experience suggests that the solution might be $T(n) = an \log n$, but this equation does not agree with the base case of the recurrence relation, $n = 1$. So let’s try $T(n) = an \log n + b$. 
Base case, $n = 1$, is correct if $C_1 = b$.

For the inductive step, we assume that the equation is true for values of $n$ up to $m/2$ and show that it is also true for $m$. Note that this application of induction matches the way in which the recurrence relation works. We do not try to prove the equation for $m + 1$ because the recurrence relation is written in terms of halving $n$ at each application.

Inductive step:

\[
T(n) = 2 \left[ \frac{a}{2} \log \left( \frac{n}{2} \right) + b \right] + C_2 n
\]

\[
= 2 \left[ \frac{an}{2} \log n - \frac{an}{2} \log 2 + b \right] + C_2 n
\]

\[
= an \log n - an + 2b + C_2 n
\]

By choosing the base of the logarithm to be 2, $\log 2 = 1$.

If $a = C_2 + b$ the guess is proved correct and $T(n) = O(n \log n)$. 

In the substitution method, we simply insert the recurrence into itself.

\[
T(n) = 2T(n/2) + C_2n \\
= 4T(n/4) + 2C_2n \\
= 8T(n/8) + 3C_2n \\
... \\
= 2^kT(n/2^k) + kC_2n
\]

When \(2^k = n\), \(k = \log n\), \(T(1) = C_1\), and \(T(n) = C_1n + C_2n\log n\)

Merge sort has the optimum running time of any general sorting algorithm. It is also very simple to write. Unfortunately it requires \(2n\) space.

Our analysis is based on the assumption that \(n\) is a power of 2. What happens when it isn’t true?
First of all, we have to work out how algorithm would cater for the general case. The most obvious way of doing so is as follows: When a sub-array is encountered of size $m$ that is not a power of 2 partition it into the largest sub-array that is a power of 2 and whatever remains. So if $m = 1023$, two sub-arrays of size 512 and 511 are formed. The technique is applied recursively.

It is fairly simple to prove that for $m \leq 2^{p+1}$ the running time is $\leq T(2^{p+1})$. An exact result would require a considerably more detailed analysis.

**A final note on merge sort**

We previously said that the algorithm sorted two sub-arrays and combined the results. If we apply the process recursively, at the lowest level, the sub-arrays comprise single values. They clearly do not need sorting. The partitioning process does not take up any time either. In merge sort, pairs of elements are merged into sub-arrays of size two (i.e. $a_0$ and $a_1$ are merged, $a_2$ and $a_3$ are merged, etc.) Then, sub-arrays of size 2 are merged to form sub-arrays of size 4, and so on.
Each merge pass takes $O(n)$ time and there are clearly $\log n$ such passes, for $n$ equals a power of 2. The running time is therefore $O(n\log n)$. This proof is by construction and appears to be a lot easier than the other techniques. Unfortunately such methods are not always simple to apply.
Here is another example:

\[
T(n) = \begin{cases} 
1 & : n = 1 \\
3T(n/2) + n & : n > 1 
\end{cases}
\]

The constants have been set equal to 1 to simplify the details.

\[
T(n) = 3^2T(n/4) + n + 3n/2 \\
= 3^3T(n/8) + n + 3n/2 + 3^2n/2^2 \\
\ldots \\
= 3^kT(n/2^k) + n \sum_{i=0}^{k} \frac{3^i}{2^i}
\]
When $n = 2^k$, $k = \log n$, $T(1) = 1$, and

\[
T(n) = 3^{\log n} + n \sum_{i=0}^{\log n} \frac{3^i}{2^i}
\]
\[
= 3^{\log n} + n \left(\frac{(3/2)^{(\log n)+1} - 1}{3/2 - 1}\right)
\]
\[
= 3^{\log n} + n(3^{\log (3/2)} - 2)
\]
\[
= 4n^{\log 3} - 2n
\]

\[
T(n) = O(n^{\log 3}) = O(n^{1.585})
\]

The following identity was used to simplify the equations:

\[
a^{\log b} = b^{\log a}
\]

Fortunately the sum was a geometric progression, which enabled a simple closed form expression.
Note that the recurrence relations for this example and for merge sort look somewhat similar, yet the results are quite different. Which equation gives the smaller running time for large $n$?

Our ability to solve recurrence relations by substitution depends on correct substitution, formulation of the general equation, and simplification of any summation and log terms into closed form expressions. It is a tedious process that requires a lot of care and attention.