IB Mathematics: Analysis & Approaches HL

Internal Assessment

Topic: Computer Programming and Algorithm Analysis.

Aim: Exploring methods for solving recurrence relations commonly occurring in algorithm analysis.

Pages: 17

Candidate code: `jky029`

Introduction

I have a strong interest in computer programming. A while back, for a small programming contest, we were asked to write a program that would display the first 100 numbers from the Fibonacci sequence. The program would be tested on different factors such as speed and length of the program. For my program, I closely followed the recursive definition of the Fibonacci sequence:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+2} &= F_n + F_{n+1} \end{aligned}$$

This is a recurrence relation (a sequence where the nth number is defined recursively) which gives the sequence $0, 1, 1, 2, 3, 5, ...$ and so on. However, I quickly noticed that my program was extremely slow, and took several minutes just to display the first 50 numbers. I had previously learned about algorithm analysis, which is a process for finding the efficiency or 'complexity' of computer programs (Cormen et al.). For example, we can analyze an algorithm to count the number of operations it will execute for a given input. If we assume each operation takes the same amount of time to execute, then we can estimate how much time the algorithm will take for an input. This is known as 'time complexity'. I became curious about the time complexity of my fibonacci algorithm, and decided to do some research on algorithm analysis and try to understand the time complexity of my algorithm.

When analyzing my fibonacci program, I found that the number of operations for any input was also given by a recurrence relation almost identical to the definition of the Fibonacci numbers. This made sense because my program closely followed that definition. However, while doing further research to learn more about this, I found that recurrence relations are a very common and natural way of expressing many different algorithms, because recursion itself appears frequently in computer algorithms. Therefore, algorithm analysis can often involve solving recurrence relations as well. There are many ways of solving recurrence relations, and some are more suitable for algorithm analysis than others. Thus, the aim of this exploration is to explore the methods which are suitable for algorithm analysis and apply them to some common algorithms, including my fibonacci numbers algorithm.

Algorithm Analysis

In general, algorithm analysis is used to find the 'computational complexity' of computer programs. Computational complexity refers to the amount of resources an algorithm needs to perform its task. For example, in this exploration, I am mainly interested in finding the time complexity of algorithms to see how much time it takes for them to execute. It is difficult to find the precise complexity for any program, because it will depend on many factors such as how powerful the computer running the program is. However, algorithms can still be analyzed by finding upper and lower bounds for the resources needed. For time complexity, this is done by counting the number of operations an algorithm will execute for a given input size, where we assume that each operation takes a constant time to execute. This gives us a function $T(n)$, where n is the input size for the program (Cormen et al.).

As an example, if we are given a list of n unique numbers and asked to find x, one algorithm to do this, called 'linear search', would be to go through each number in the list one by one from start to finish, and check if it is equal to x. We assume the time taken to check if a number is equal to x is constant, denoted by c. If x happens to be the first number, then our algorithm would be finished after one operation, so $T(n) = 1 \times c$. In other words, this is a best-case scenario since this is the smallest time that the algorithm can take. If x happens to be the last number, then we would have to go through the entire list, and so $T(n) = n \times c$. This is the worst-case scenario.

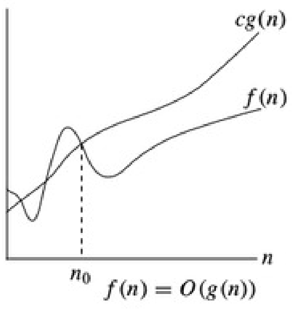
Since constants depend on factors such as the hardware, they are usually unrelated to the efficiency of the algorithm itself and so we choose to ignore them. Instead, we focus only on how fast $T(n)$ grows for larger inputs. For example, if an algorithm had the time complexity $T(n) = 4n^2 - 2n + 2$, we can ignore the constants and focus only on the n^2 term, since it dominates all other terms as the input gets larger. We say that "$T(n)$ grows at the *order* of n^2." This is expressed using **Landau notation** as $T(n) \in O(n^2)$. By using this notation, we can ignore the constants and focus only on the growth rate of the algorithm (Cormen et al.).

More specifically, the notation $f(n) \in O(g(n))$ (sometimes written with '=' instead of '$\in$') represents the asymptotic upper bound on the growth rate of $f(n)$. This means that as n becomes larger and larger, the function $g(n)$ grows faster than $f(n)$. $O(g(n))$ is defined as:

$$O(g(n)) = \{f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n), \text{ for all } n \geq n_0\}$$

In other words, $O(g(n))$ is just the set of all functions that eventually grow slower than $g(n)$, for any constants attached to $g(n)$. Since it is the upper bound for the time taken, it represents the worst-case scenario for the algorithm. For example, we can say that the linear search algorithm is in $O(n)$, because in the worst-case scenario it takes n operations (but never more than that).

Graphically, $f(n) \in O(g(n))$ means:



Source: HackerEarth

As the graph shows, for all sufficiently large n ($n > n_0$), the function $g(n)$ grows larger than $f(n)$, regardless of the constants attached to it.

In some cases an algorithm may have different behavior for inputs of the same size (for example, the linear search algorithm where we can have the best-case and worst-case scenarios). In these cases, we can use $\Omega(g(n))$ for the asymptotic lower bound (representing the best-case scenario), and $\Theta(g(n))$ for the asymptotic tight bound (representing the average case scenario):

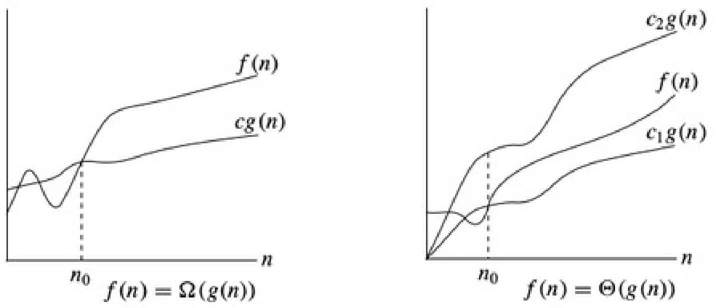
$$\Omega(g(n)) = \{f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n), \text{ for all } n \geq n_0 \}$$

$$\Theta(g(n)) = \{f(n) \mid \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ for}$$

all $n > n_0$ }

In other words, $\Omega(g(n))$ is the set of all functions that grow faster than $g(n)$. $\Theta(g(n))$ is the set of all functions that grow just as fast as $g(n)$.

Graphically:



Source: HackerEarth

Notice that when $f(n) \in \Theta(g(n))$, we also have that $f(n) \in \Omega(g(n))$ and $f(n) \in O(g(n))$. This is because $f(n)$ and $g(n)$ grow at the same rate, but we can attach a large constant c_2 to $g(n)$ so that it is larger than $f(n)$, and a small constant c_1 so that it is smaller than $f(n)$. For example, $10n \in \Theta(n)$, because we can have $100n$ as the upper bound, and $0.1n$ as the lower bound by attaching the constants 100 and 0.1, respectively, to $g(n)$. However, this is not possible for $10n \in O(n^2)$, because even for very small constants, n^2 will eventually grow larger than $10n$.

We can also use limits to express this idea:

$$\lim_{n\to\infty} \frac{T(n)}{g(n)} = \begin{cases} 0 & \text{- } T(n) \text{ has smaller growth rate than } g(n) \\ c & \text{- } T(n) \text{ has the same growth rate as } g(n) \\ \infty & \text{- } T(n) \text{ has larger growth rate than } g(n) \end{cases}$$

When we find the time complexity $T(n)$ for an algorithm, this definition can be helpful in determining a function $g(n)$ for the upper, lower or tight bounds. For example, linear search is $T(n) = cn$. If we choose $g(n) = n$, then $\lim_{n\to\infty} \frac{cn}{n}$ evaluates to c, a constant. This means $T(n)$ grows as fast as $g(n)$, so $T(n) \in \Theta(g(n))$. For $g(n) = n^2$, we have $\lim_{n\to\infty} \frac{cn}{n^2} = 0$. Thus, $T(n)$ grows slower than $g(n)$, so we say $T(n) \in O(g(n))$. For $g(n) = \log n$, we have $\lim_{n\to\infty} \frac{cn}{\log n} = \infty$. This means $T(n)$ grows faster than $g(n)$, and so $T(n) \in \Omega(g(n))$.

Since we are dealing with asymptotic bounds and do not care about constants, it is not necessary to fully solve recurrence relations; approximating the bounds can also be useful for algorithm analysis.

In this exploration, I explored three methods for solving recurrence relations and applied them to algorithm analysis: the first method uses 'generating functions', the second is known as the expansion method, and the third makes use of the 'Master Theorem' for algorithm analysis.

Method 1: Generating Functions & Series

We can think of an algorithm's time complexity as a sequence $T(n)$ indexed by n: $T(0), T(1), T(2), \ldots$

and so on. Then, we can try to find the 'generating function' for this sequence, which is a special kind of function that can be expanded into a *generating series* (Levin). A generating series is an infinite power series (a series of the form $\sum_{k=0}^{\infty} c_k x^k$) whose coefficients "display" the terms of the sequence we are interested in. For example, the generating series $1 + 2x + 3x^2 + 4x^3 + ...$ has coefficients that represent the sequence $1, 2, 3, 4, ...$ and so on. A *generating function* is a function (with a closed formula) which can be expanded into an infinite power series. This power series will also be a generating series for the sequence we are interested in. For example, the power series $1 + x + \frac{x^2}{2} + \frac{x^3}{6} + ... + \frac{x^n}{n!} + ...$ has the closed form e^x. Then, we call e^x the generating function for the sequence $\frac{1}{0!}, \frac{1}{1!}, \frac{1}{2!}, \frac{1}{3!}, ..., \frac{1}{n!}, ...$ and so on. If we can write the generating series in the form $\sum_{n=0}^{\infty} f(n)x^n$, then $f(n)$ is a closed formula for the coefficient of the nth term of the series (Wilf). Since this generating series comes from the sequence given by the algorithm's time complexity $T(n)$, these coefficients are the elements of the sequence we started with. Thus, the formula $f(n)$ is also a function for the time complexity of the algorithm.

The advantage of using generating functions for recurrence relations is that recurrence relations can be easily translated to a sequence and then solved using the method above.

In general, linear recurrence relations can usually be solved with generating functions using the following steps:

1. Make sure the the set of values of the index variable for the relation (such as n) is clearly outlined.
2. Name the generating function (say, $G(x)$) that we will look for, and write it out in terms of the unknown sequence (for example, $G(x) = \sum_{n \geq 0} a_n x^n$).
3. Multiply both sides of the recurrence by x^n, and sum over all values of n for which the recurrence holds.
4. Express both sides of the resulting equation in terms of the generating function.
5. Solve the equation for the unknown generating function.
6. Expand the generating function into a power series using any method. This will give an explicit formula for the sequence defined by the recurrence relation.

The last step will give us a closed form expression for the recurrence relation, which can be used to find the asymptotic growth rate for the algorithm from which the relation arose. We will use this method to analyze the time complexity of the same fibonacci algorithm I wrote, as mentioned in the Introduction. A simplified version of the algorithm is as follows:

```
fib(n):
   if n = 0 or n = 1:
       return 1
   else:
       return fib(n-1) + fib(n-2)
```

In the first case, when $n \leq 1$, the function simply returns 1; we will assume this takes a constant time c to perform. In the second case, we follow the definition of fibonacci numbers and return the sum of the previous two numbers. We assume the addition of these two numbers also takes constant time c; `fib(n-1)` takes time $T(n-1)$, while `fib(n-2)` takes time $T(n-2)$. Thus, the total time is $T(n) = T(n-1) + T(n-2) + c$.

The first step is to make sure the values of the index variable are clearly outlined. The index variable here is n. Since we are calculating fibonacci numbers, we know that $n \in \mathbb{Z}^+$. Our recurrence relation $T(n) = T(n-1) + T(n-2) + c$ holds true for $n \geq 2$. When n is 0 or 1, we only perform a single operation, so $T(0) = T(1) = c$.

Thus, we have our recurrence relation:

$$\begin{aligned} T(0) &= T(1) = c \\ T(n) &= T(n-1) + T(n-2) + c \end{aligned}$$

The second step is to name our generating function and write it in terms of the unknown power series:

$$G(x) = \sum_{n=0}^{\infty} T(n)x^n$$

The next step is to go back to the recurrence relation we found, and multiply both sides of it by x^n and sum over values of n for which it holds:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c \\ \implies \sum_{n=0}^{\infty} T(n)x^n &= \sum_{n=0}^{\infty} \big(T(n-1) + T(n-2) + c\big)x^n \end{aligned}$$

The next step is to express *both* sides of the equation in terms of $G(x)$. The LHS is easy, since it is equal to the expression for $G(x)$ we had above:

$$\begin{aligned} \sum_{n=0}^{\infty} T(n)x^n &= \sum_{n=0}^{\infty} \big(T(n-1) + T(n-2) + c\big)x^n \\ \therefore\ G(x) &= \sum_{n=0}^{\infty} \big(T(n-1) + T(n-2) + c\big)x^n \end{aligned}$$

However, the RHS requires some manipulation. Since we know $T(0) = T(1) = c$, we can take out the first two terms of the sum and change the index:

$$\begin{aligned} G(x) &= \sum_{n=0}^{\infty} \big(T(n-1) + T(n-2) + c\big)x^n \\ &= 2c + \sum_{n=2}^{\infty} \big(T(n-1) + T(n-2) + c\big)x^n \end{aligned}$$

Then we separate the sums:

$$= 2c + \sum_{n=2}^{\infty} T(n-1)x^n + \sum_{n=2}^{\infty} T(n-2)x^n + \sum_{n=2}^{\infty} cx^n$$

Then, factor x from the first sum and x^2 from the second:

$$= 2c + x\sum_{n=2}^{\infty} T(n-1)x^{n-1} + x^2\sum_{n=2}^{\infty} T(n-2)x^{n-2} + \sum_{n=2}^{\infty} cx^n$$

Notice that the second sum is now the same as $\sum_{n=0}^{\infty} T(n)x^n$, which is equal to $G(x)$ and so we can replace it:

$$= 2c + x\sum_{n=2}^{\infty} T(n-1)x^{n-1} + x^2G(x) + \sum_{n=2}^{\infty} cx^n$$

To do the same with the first sum, we need to change the index so that $n = 1$. The $n = 1$ term is $xT(1) = cx$, so we add and subtract this term to bring it inside the equation:

$$\begin{aligned}
&= 2c + x\sum_{n=2}^{\infty} T(n-1)x^{n-1} + (xT(1) - xT(1)) + x^2G(x) + \sum_{n=2}^{\infty} cx^n \\
&= 2c + x\sum_{n=2}^{\infty} T(n-1)x^{n-1} + xT(1) - cx + x^2G(x) + \sum_{n=2}^{\infty} cx^n
\end{aligned}$$

We can move $xT(1)$ into the first sum, and change its index to $n = 1$, allowing us to replace it with $G(x)$:

$$\begin{aligned}
&= 2c + x\sum_{n=1}^{\infty} T(n-1)x^{n-1} - cx + x^2G(x) + \sum_{n=2}^{\infty} cx^n \\
&= 2c - cx + xG(x) + x^2G(x) + \sum_{n=2}^{\infty} cx^n
\end{aligned}$$

Now, we solve the equation for $G(x)$:

$$\begin{aligned}
G(x) &= 2c - cx + xG(x) + x^2G(x) + \sum_{n=2}^{\infty} cx^n \\
G(x) - xG(x) - x^2G(x) &= 2c - cx + \sum_{n=2}^{\infty} cx^n \\
G(x)\left[1 - x - x^2\right] &= 2c - cx + \sum_{n=2}^{\infty} cx^n
\end{aligned}$$

Before isolating $G(x)$, we can simplify the last summation. We know that the infinite sum of a converging geometric series is $\frac{a}{1-r}$, where a is the first term and r is the common ratio. A series of the form $\sum_{k=0}^{\infty} cx^k$ is also a geometric series, which converges for $|x| < 1$. In the context of generating functions, we are only interested in the coefficients of the series, and not the values of x, so we can choose any convenient value for x and assume that the series will always converge (Levin). Since $\sum_{n=2}^{\infty} cx^n$ is also a geometric series, with first term cx^2, we get:

$$\sum_{n=2}^{\infty} cx^n = \frac{cx^2}{1-x}$$

We can then simplify the RHS of our equation:

$$\begin{aligned}
G(x)\left[1 - x - x^2\right] &= 2c - xc + \frac{cx^2}{1-x} \\
&= \frac{2c(1-x)}{1-x} - \frac{xc(1-x)}{1-x} + \frac{cx^2}{1-x} \\
&= \frac{2c - 3xc + 2x^2c}{1-x}
\end{aligned}$$

Now we can solve for $G(x)$:

$$\begin{aligned}
G(x)\left[1 - x - x^2\right] &= \frac{2c - 3xc + 2x^2c}{1 - x} \\
G(x) &= \frac{2c - 3xc + 2x^2c}{(1 - x)(1 - x - x^2)} \\
&= \frac{2c - 3xc + 2x^2c}{1 - 2x + x^3} \\
&= c \times \frac{2 - 3x + 2x^2}{1 - 2x + x^3}
\end{aligned}$$

The last step is to expand this into a power series using any suitable method. Since we have a rational function, we can try to decompose it into partial fractions. We can then use the identity

$$\frac{1}{1 - x} = \sum_{n=0}^{\infty} x^n$$

to expand the fractions into a power series (again, we assume this identity holds and is valid because we never actually use the value of x; we are only interested in the coefficients of the series).

For convenience, we will ignore c for now and work only with the fraction, naming it $g(x)$:

$$\begin{aligned}
G(x) &= c \times g(x) \\
g(x) &= \frac{2 - 3x + 2x^2}{1 - 2x + x^3}
\end{aligned}$$

To decompose $g(x)$, we first factorize the denominator:

$$1 - 2x + x^3 = (x - 1)(x + \phi)(x + \psi)$$

Where:

$$\phi = \frac{1 + \sqrt{5}}{2}, \quad \psi = \frac{1 - \sqrt{5}}{2}$$

Thus:

$$\begin{aligned}
g(x) &= \frac{2 - 3x + 2x^2}{(x - 1)(x + \phi)(x + \psi)} \\
&= \frac{A}{(x - 1)} + \frac{B}{(x + \phi)} + \frac{C}{(x + \psi)}
\end{aligned}$$

This is equivalent to:

$$2 - 3x + 2x^2 = A(x + \psi)(x + \phi) + B(x - 1)(x + \psi) + C(x - 1)(x + \phi)$$

Setting $x = -\psi$ gives us C:

$$\begin{aligned}
2 - 3(-\psi) + 2(-\psi^2) &= C(-\psi - 1)(-\psi + \phi) \\
C &= \frac{2 + 3\psi + 2\psi^2}{\psi^2 - \psi\phi + \psi - \phi}
\end{aligned}$$

Setting $x = -\phi$ gives us B:

$$\begin{aligned}
2 - 3(-\phi) + 2(-\phi^2) &= B(-\phi - 1)(-\phi + \psi) \\
B &= \frac{2 + 3\phi + 2\phi^2}{\phi^2 - \phi\psi + \phi - \psi}
\end{aligned}$$

Setting $x = 1$ gives us A:

$$\begin{aligned}
2 - 3(1) + 2(1^2) &= A(1 + \psi)(1 + \phi) \\
A &= \frac{1}{1 + \phi + \psi + \psi\phi}
\end{aligned}$$

Luckily, the numbers ϕ and ψ have some nice properties (see **Appendix A**) that will let us simplify our answers:

$$\begin{aligned}
\phi \times \psi &= -1 & \phi + \psi &= 1 \\
\phi^2 - \phi &= 1 & \psi^2 - \psi &= 1 \\
\phi - \psi &= \sqrt{5} & \psi - \phi &= -\sqrt{5}
\end{aligned}$$

Simplifying A:

$$A = \frac{1}{1 + (\phi + \psi) + (\psi\phi)} = 1$$

Simplifying B:

$$\begin{aligned}
&(\mathit{numerator}) & &(\mathit{denominator}) \\
2 + 3\phi + 2\phi^2 &= 2 + 3\phi + 2(\phi + 1) & \phi^2 - \phi\psi + \phi - \psi &= (\phi + 1) - (-1) + (\sqrt{5}) \\
&= 4 + 5\phi & &= 2 + \phi + \sqrt{5}
\end{aligned}$$

Similarly for C:

$$\begin{aligned}
&(\mathit{numerator}) & &(\mathit{denominator}) \\
2 + 3\psi + 2\psi^2 &= 2 + 3\psi + 2(\psi + 1) & \psi^2 - \psi\phi + \psi - \phi &= (\psi + 1) - (-1) - (\sqrt{5}) \\
&= 4 + 5\psi & &= 2 + \psi - \sqrt{5}
\end{aligned}$$

Putting A, B, C back into $g(x)$:

$$g(x) = \frac{1}{x - 1} + \frac{\Phi}{x + \phi} + \frac{\Psi}{x + \psi}$$

Where:

$$\Phi = B = \frac{4 + 5\phi}{2 + \phi + \sqrt{5}}, \quad \Psi = C = \frac{4 + 5\psi}{2 + \psi - \sqrt{5}}$$

Now that we have decomposed the fraction, we need to bring it to the form $\frac{a}{1-r}$ in order to expand it into a power series. The first term is easy to rewrite:

$$\frac{1}{x - 1} = \frac{-1}{1 - x}$$

For the second fraction, we divide each term inside it by ϕ:

$$\frac{\frac{\Phi}{\phi}}{\frac{x}{\phi}+\frac{\phi}{\phi}} = \frac{\frac{\Phi}{\phi}}{\frac{x}{\phi}+1}$$

Since we know $\phi\psi = -1$, we have that $-\psi = \frac{1}{\phi}$:

$$\frac{\frac{\Phi}{\phi}}{\frac{x}{\phi}+1} = \frac{\frac{\Phi}{\phi}}{1-\psi x}$$

We do the same with the third fraction, dividing each term inside it with ψ this time to obtain:

$$\frac{\Psi}{x+\psi} = \frac{\frac{\Psi}{\psi}}{1-\phi x}$$

We now have:

$$g(x) = \frac{-1}{1-x} + \frac{\frac{\Phi}{\phi}}{1-\psi x} + \frac{\frac{\Psi}{\psi}}{1-\phi x}$$

We can rewrite each term as a power series:

$$\begin{aligned}
&= \sum_{n=0}^{\infty} -x^n + \sum_{n=0}^{\infty} \frac{\Phi}{\phi}\psi^n x^n + \sum_{n=0}^{\infty} \frac{\Psi}{\psi}\phi^n x^n \\
&= \sum_{n=0}^{\infty} \left(\frac{\Phi}{\phi}\psi^n + \frac{\Psi}{\psi}\phi^n - 1\right) x^n
\end{aligned}$$

Returning to our original generating function $G(x)$:

$$\begin{aligned}
G(x) &= c \times g(x) \\
&= c \times \sum_{n=0}^{\infty} \left(\frac{\Phi}{\phi}\psi^n + \frac{\Psi}{\psi}\phi^n - 1\right) x^n \\
&= \sum_{n=0}^{\infty} \left(\frac{\Phi}{\phi}\psi^n + \frac{\Psi}{\psi}\phi^n - 1\right) cx^n
\end{aligned}$$

We can now equate this with the original expression for $G(x)$ which we started out with:

$$\begin{aligned}
G(x) &= \sum_{n=0}^{\infty} T(n)x^n \\
\sum_{n=0}^{\infty} T(n)x^n &= \sum_{n=0}^{\infty} \left(\frac{\Phi}{\phi}\psi^n + \frac{\Psi}{\psi}\phi^n - 1\right) cx^n \\
\therefore\ T(n) &= c \times \left(\frac{\Phi}{\phi}\psi^n + \frac{\Psi}{\psi}\phi^n - 1\right)
\end{aligned}$$

We now have a closed form expression for the recurrence relation we started with. Asymptotically, we can see that $T(n)$ is of the form a^n. We can now also use the limit definition for Landau notation to find the asymptotic bounds for $T(n)$. Since $T(n)$ is an exponential function, we can let $g(n) = a^n$

for now, and analyze the limit for different values of a:

$$\begin{aligned}\lim_{n\to\infty}\frac{T(n)}{g(n)} &= \lim_{n\to\infty}\frac{c\times\left(\frac{\Phi}{\phi}\psi^n+\frac{\Psi}{\psi}\phi^n-1\right)}{a^n}\\ &= c\times\lim_{n\to\infty}\left(\frac{\frac{\Phi}{\phi}\psi^n}{a^n}+\frac{\frac{\Psi}{\psi}\phi^n}{a^n}-\frac{1}{a^n}\right)\\ &= c\times\left(\frac{\Phi}{\phi}\lim_{n\to\infty}\frac{\psi^n}{a^n}+\frac{\Psi}{\psi}\lim_{n\to\infty}\frac{\phi^n}{a^n}-\lim_{n\to\infty}\frac{1}{a^n}\right)\end{aligned}$$

Since we are considering time complexity, we know a cannot be negative because time (and hence the time complexity function) is always positive. We also know $a \neq 1$, since that would be a constant function. Furthermore, since $-1 < \psi < 0$, we have that ψ^n approaches 0 as n approaches infinity, so the first limit evaluates to 0 regardless of the value of a. We then have:

$$= c\times\left(\frac{\Psi}{\psi}\lim_{n\to\infty}\frac{\phi^n}{a^n}-\lim_{n\to\infty}\frac{1}{a^n}\right)$$

For $0 < a < 1$, we can write the limit as follows:

$$= c\times\left(\lim_{n\to\infty}\frac{\frac{\Psi}{\psi}\phi^n-1}{a^n}\right)$$

Here, we can see that the limit evaluates to infinity since the numerator gets larger and larger while the denominator gets smaller and smaller as $n\to\infty$. If we consider $1 < a < \phi$, we notice the same behavior:

$$= c\times\left(\frac{\Psi}{\psi}\lim_{n\to\infty}\frac{\phi^n}{a^n}-\lim_{n\to\infty}\frac{1}{a^n}\right)$$

The second limit evaluates to 0, while the first evaluates to infinity, since $\phi^n > a^n$. So we have the lower bound for the time complexity:

$$T(n)\in\Omega(a^n),\quad 0<a<\phi,\quad a\neq 1$$

When $a=\phi$, the last limit again evaluates to 0. In the second one, the numerator and denominator are the same, and so it evaluates to 1. We thus have:

$$\begin{aligned}&a=\phi\\ &\Longrightarrow c\times\left(\frac{\Psi}{\psi}\lim_{n\to\infty}\frac{\phi^n}{\phi^n}-\lim_{n\to\infty}\frac{1}{\phi^n}\right)\\ &\quad= c\times\left(\frac{\Psi}{\psi}-0\right)\\ &\quad= c\times\frac{\Psi}{\psi}\end{aligned}$$

Since this is a constant, we have the tight bound for $T(n)$:

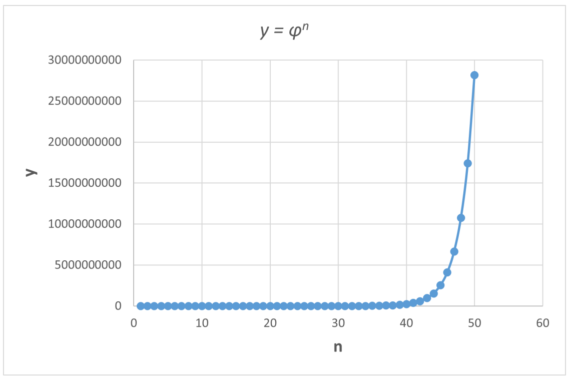
$$T(n)\in\Theta(\phi^n)$$

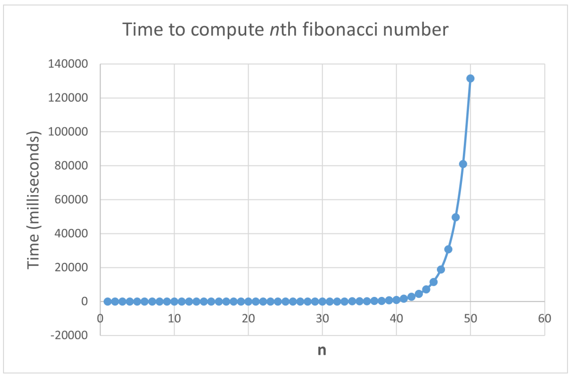
For $a > \phi$, it is easy to notice that the limits evaluate to 0, giving us the upper bound:

$$T(n) \in O(a^n), \quad \phi < a$$

This process also shows how useful the limit definition can be for better understanding the time complexity of an algorithm.

To test if my result was correct or not, I wrote my original algorithm once again and this time recorded the time taken for the program (see **Appendix B** for the program used) to compute each fibonacci number from 0 to 50 (see **Appendix C** for raw data). I plotted the data on a graph, and compared it to the graph of ϕ^n for $0 \leq n \leq 50$:





Clearly, the two functions have the same growth rate. This means that $T(n) \in \Theta(\phi^n)$ is accurate.

We can also test the accuracy of $T(n)$ itself. We do not know the constant term c, however, because it is an exponential function, the ratio of the nth and $(n-1)$th term should be constant. I used this as a test for the accuracy of my solution by computing $T(n)$ for some n and $(n-1)$, and then comparing their ratio to the ratio of the same n and $n-1$ values I found in my data collection. I used $n = 50$, and wrote a program (see **Appendix D**) to compute $T(50)$ and $T(49)$:

$$T(50) = 48508764196.00005$$
$$T(49) = 29980065025.00004$$
$$\frac{T(50)}{T(49)} = 1.618033988770509$$

Interestingly, the ratio came out to be the golden ratio which makes sense considering the recurrence relation was essentially the same as the fibonacci sequence. From the data collection (let $t(n)$ = time taken to compute nth fibonacci number):

$$t(50) = 131459.9375$$
$$t(49) = 80926.4219$$
$$\frac{t(50)}{t(49)} = 1.624437784515467$$

This is quite close the previous ratio, and indicates that our time complexity function $T(n)$ is accurate. The growth rate of exponential functions is quite fast, which also explains why my fibonacci algorithm was so slow for larger inputs.

The advantage of using generating functions to solve recurrence relations is that the process is quite methodical. Following the steps will usually lead to a solution for the relation. In fact, the process can even be performed by computer algebra software to solve recurrences for us.

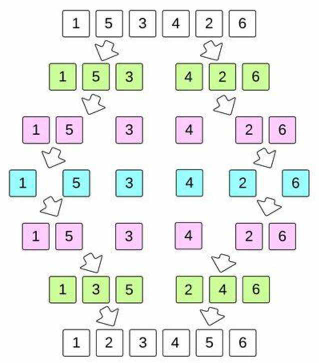
Method 2: Expansion Method

While generating functions are powerful and can be used for non-linear recurrence relations, the process is not as straightforward compared to linear relations. A common class of algorithms is the 'divide-and-conquer' algorithms, which divide a problem into smaller sub-problems which are easier and faster to solve. These algorithms also involve recursion and so their time complexity function is a recurrence relation as well. However, it is not a linear recurrence because when the problem is divided into sub-problems, the input size for each sub-problem is reduced by a factor of, for example, 2 or more (depending on how the problem is divided up) (Levitin). This means the time complexity recurrence relation is non-linear, and so I wanted to explore methods of solving these next.

A different technique which is usually simpler is the expansion method (also called iterative or substitution method). It consists of three simple steps (Cormen et al.):

1. Expand the recurrence relation.
2. Guess the pattern for the relation.
3. Prove the pattern, usually by induction.

I used this method to find the time complexity of the mergesort algorithm. *Mergesort* is an efficient and popular algorithm for sorting lists of, for example, numbers. It is also a divide-and-conquer algorithm, which means its time complexity is a non-linear recurrence relation. Essentially, it divides the given list into smaller lists, sorts the smaller lists and then merges them back together. Below is a visualization of the algorithm (see **Appendix E** for the pseudocode):



Source: Sehgal

In this case, our input size n is the size of the list we need to sort. First, if the list is of size 0 or 1, we simply return the same list, so $T(0) = T(1) = c$. For $n > 1$, we divide the list into two smaller lists and recursively call the algorithm on the two sub-lists, and then merge them. For simplicity, we assume that $n = 2^m, m \in \mathbb{Z}^+$, so that the list can be divided evenly at each iteration. Then, the recursive calls to the two sub-lists take time $T(\frac{n}{2}) + T(\frac{n}{2}) = 2T(\frac{n}{2})$. We divide the list into half at each iteration, which takes $\frac{n}{2}$ operations. Afterwards, we merge the lists back together. This again takes $\frac{n}{2}$ operations. If each operation takes time c, then we have an additional $c\left(\frac{n}{2}+\frac{n}{2}\right)=cn$ time taken for the dividing and merging of the list. Putting all this together, we have the following recurrence relation for the time complexity:

$$\begin{aligned} T(0) &= T(1) = c \\ T(n) &= 2T(\frac{n}{2}) + cn \end{aligned}$$

Now we can try to solve this relation. The first step in the expansion method is, of course, to expand the equation:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + cn \\ &= 2(2T(\frac{n}{4}) + \frac{cn}{2}) + cn \\ &= 4T(\frac{n}{4}) + cn + cn \\ &= 4(2T(\frac{n}{8}) + \frac{cn}{4}) + cn + cn \\ &= 8T(\frac{n}{8}) + cn + cn + cn \\ &= nT(\frac{n}{n}) + cn + ... + cn + cn \\ &= nT(1) + cn + ... + cn + cn \\ &= cn + cn + ... + cn + cn \end{aligned}$$

We start with $T(n)$, and at each iteration the input to $T(n)$ is halved, while cn is added to the total sum. Since we are dividing by half, this process repeats $\log_2 n$ times, until we have $nT(1)$. This is again equal to cn, so in total we have $\log_2 n + 1$ terms, where each term is cn. Therefore, we can *guess* that:

$$\begin{aligned} T(n) &= cn(\log_2 n + 1) \\ &= cn\log_2 n + cn \end{aligned}$$

Now that we have our guess, we can try to prove it using strong induction:

$$\mathbf{P(n):}\ T(n) = 2T(\frac{n}{2}) + cn = cn\log_2 n + cn,\ n \geq 2$$

(Again, we assume $n = 2^m$, $m \in \mathbb{Z}^+$).

Base case $n = 2$:

$$\begin{aligned} T(2) &= 2T(\frac{2}{2}) + 2c \\ &= 2T(1) + 2c \\ &= 2c + 2c \\ &= 2c\log_2 2 + 2c \end{aligned}$$

Assume the statement is true for all $k < n$ (inductive hypothesis):

$$\mathbf{P(k):}\ T(k) = 2T(\frac{k}{2}) + ck = ck\log_2 k + ck$$

When $n = 2k$:

$$\begin{aligned}
T(2k) &= 2T(\frac{2k}{2}) + 2kc \\
&= 2T(k) + 2kc \\
&= 2kc\log_2 k + kc + 2kc && \text{(by inductive hypothesis)} \\
&= kc(2\log_2 k + 1) + 2kc \\
&= kc(2\log_2 k + \log_2 2) + 2kc \\
&= kc(2\log_2 2k) + 2kc \\
&= 2kc\log_2 2k + 2kc
\end{aligned}$$

Since $\mathbf{P(n)}$ was true for base case $n = 2$, and it has been shown that when $n = k$, $\mathbf{P(k)} \implies \mathbf{P(2k)}$, it follows by the principle of mathematical induction that $\mathbf{P(n)}$ is true for all $n = 2^m$, $m \in \mathbb{Z}^+$.

Thus, we have solved the recurrence relation and found that $T(n) = cn\log_2 n + cn$. Again, we can express this growth rate in Landau notation using the same method as before. This time it is much simpler, because simply choosing $g(n) = n\log_2 n$ gives us the tight bound:

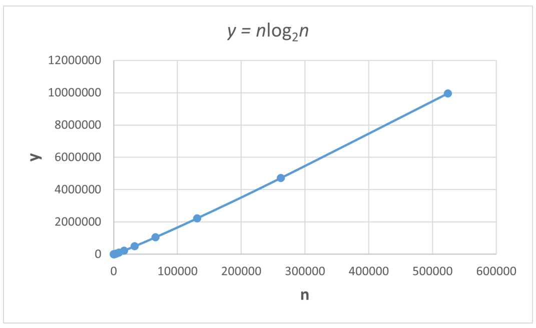
$$\begin{aligned}
\lim_{n\to\infty} \frac{T(n)}{g(n)} &= \lim_{n\to\infty} \frac{cn\log_2 n + cn}{n\log_2 n} \\
&= \lim_{n\to\infty} \frac{cn\log_2 n}{n\log_2 n} + \lim_{n\to\infty} \frac{cn}{n\log_2 n} \\
&= c
\end{aligned}$$

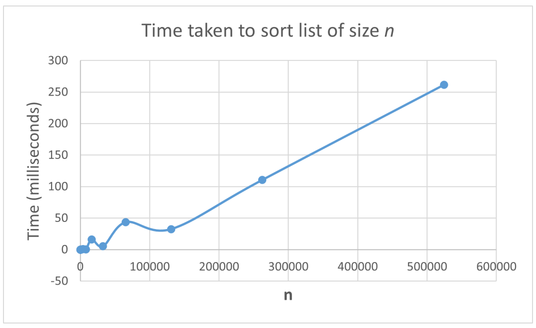
Thus:

$$T(n) \in \Theta(n\log_2 n)$$

Although it is beyond the scope of this essay, it can also be shown that the time complexity for all $n \in \mathbb{Z}^+$ is approximately the same, even without the restriction $n = 2^m$, $m \in \mathbb{Z}^+$.

To test the result, I again ran the program (see **Appendix F**) and recorded the time taken (**Appendix G**):





Again, we can see that the growth rate of the two graphs is quite similar for sufficiently large n ($n > 130000$ in this case), indicating that our solution is accurate. Notice that for smaller values of n, the graph from the data collected fluctuates. This is likely because of external factors related to my computer running the program, causing differences in the time recorded. This shows why mathematically analyzing the growth rate of an algorithm as the input gets larger, while ignoring external constant factors, is much more useful instead of simply collecting raw data and using that to measure the efficiency of an algorithm.

One advantage of the expansion method is that it is quite simple and elegant, as long as we can find the pattern. However, it is much less methodical than the previous method, and requires some creativity when trying to guess the pattern for the recurrence relation, which could be much more difficult for other algorithms.

Method 3: Master Theorem

The expansion method is simple, but not always suitable since it can sometimes be difficult to notice a pattern when expanding the recurrence relation. Luckily, for divide-and-conquer algorithms such as the mergesort algorithm from the previous section, the Master Theorem gives an easy method for finding the asymptotic growth (Levitin).

The Master Theorem can be applied to recurrences of the form:

$$T(n) = aT(\frac{n}{b}) + f(n), \text{ where } a \geq 1, b > 1$$

To understand the idea behind this theorem, we ignore the $f(n)$ term for now, and instead try to solve the recurrence $T(n) = aT(\frac{n}{b})$. We can do this using the expansion method discussed previously:

$$\begin{aligned} T(n) &= aT(\frac{n}{b}) \\ &= aaT(\frac{\frac{n}{b}}{b}) \\ &= a^2T(\frac{n}{b^2}) \\ &= a^3T(\frac{n}{b^3}) \\ &= a^iT(\frac{n}{b^i}) \end{aligned}$$

We again simplify this by assuming that $n = b^k$, so that the recurrence stops at $T(\frac{n}{n}) = T(1)$. This means it stops when $b^i = n$, which is equivalent to $\log_b n = i$. Thus:

$$\begin{aligned} T(n) &= a^iT(\frac{n}{b^i}) \\ &= a^{\log_b n}T(1) \end{aligned}$$

$a^{\log_b n}$ can be rewritten as $n^{\log_b a}$. Asymptotically, we have that $T(n) \in \Theta(n^{\log_b a})$. This is the basis of the Master Theorem, which essentially compares $n^{\log_b a}$ and $f(n)$ to find which term dominates over the other. To apply the theorem, we consider the following three cases:

$$\begin{aligned} \textbf{Case 1: } & f(n) \in O(n^{\log_b(a)-\epsilon}), \text{ for some } \epsilon > 0 \\ \textbf{Then: } & T(n) \in \Theta(n^{\log_b a}) \end{aligned}$$

$$\begin{aligned} \textbf{Case 2: } & f(n) \in \Theta(n^{\log_b a}) \\ \textbf{Then: } & T(n) \in \Theta(n^{\log_b a} \log_2 n) \end{aligned}$$

$$\begin{aligned} \textbf{Case 3: } & f(n) \in \Omega(n^{\log_b(a)+\epsilon}), \text{ for some } \epsilon > 0 \\ & \quad \textbf{and } af(\frac{n}{b}) \leq cf(n) \text{ for some } c < 1, \text{ for all sufficiently large } n \\ \textbf{Then: } & T(n) \in \Theta(f(n)) \end{aligned}$$

Case 1 checks whether $f(n)$ is dominated by $n^{\log_b a}$ (recall that $O(g(n))$ represents the asymptotic upper bound). If there exists some ϵ such that the asymptotic upper bound of $f(n)$ is $n^{\log_b(a)-\epsilon}$, the theorem states that $T(n)$ is in $\Theta(n^{\log_b a})$. This is because $n^{\log_b(a)-\epsilon} < n^{\log_b a}$, and so $n^{\log_b a}$ dominates. Case 3 is the opposite, where we have that $f(n)$ dominates over $n^{\log_b a}$. The extra condition in case 3 is known as the *regularity condition* (Cormen et al.), which is needed for the proof of the Master Theorem. The proof is beyond the scope of this essay; we will assume the condition is required for the case to apply. In Case 2, we have that if both $f(n)$ and $n^{\log_b a}$ are asymptotically equivalent, then $T(n) \in \Theta(n^{\log_b a} \log_2 n)$.

The recurrence relation for the time complexity of the mergesort algorithm explored in the previous section is of the form $T(n) = aT(\frac{n}{b}) + f(n)$, so we can also try to find its asymptotic growth using the Master Theorem. From the previous section, we found the recurrence relation for mergesort to be:

$$\begin{aligned} T(0) = T(1) &= c \\ T(n) &= 2T(\frac{n}{2}) + cn \end{aligned}$$

We have $a = 2, b = 2, f(n) = cn$. We check case 1 first:

$$\begin{aligned} f(n) &\in O(n^{\log_b(a)-\epsilon}), \text{ for some } \epsilon > 0 \\ n^{\log_b(a)-\epsilon} &= n^{\log_2(2)-\epsilon} \\ &= n^{1-\epsilon} \end{aligned}$$

Notice that for any $\epsilon > 0$, we have $1 - \epsilon < 1$. Since we know that $n^k < n$ for any $k < 1$, we can see that $f(n) = cn$ grows faster than $n^{1-\epsilon}$. Thus, $f(n) \notin O(n^{\log_b(a)-\epsilon})$ and so case 1 does not apply.

We check case 2 next:

$$\begin{aligned} f(n) &\in \Theta(n^{\log_b a}) \\ n^{\log_b a} &= n^{\log_2 2} \\ &= n \end{aligned}$$

Since $f(n) = cn$ and $n^{\log_b a} = n$, we can see that $f(n) \in \Theta(n^{\log_b a})$ holds. This means we can apply case 2 to this relation:

$$\begin{aligned}\textbf{Case 2: } & f \in \Theta(n^{\log_b a}) \\ \textbf{Then: } & T(n) \in \Theta(n^{\log_b a} \log_2 n)\end{aligned}$$

$$\begin{aligned}n^{\log_b a} \log_2 n &= n^{\log_2 2} \log_2 n \\ &= n \log_2 n\end{aligned}$$

Thus, by the master theorem, $T(n) \in \Theta(n \log_2 n)$. This is, of course, the same result we found in the previous section and tested using experimental data as well.

It can be seen above that the Master Theorem is quite useful because of how easy it is to apply on many divide-and-conquer algorithms. It is, however, important to note that the Master Theorem method only tells us about the asymptotic growth of divide-and-conquer recurrence relations, but it does not actually provide a proper *solution* to the relation. While it is usually sufficient for analyzing algorithms, other methods would be more suitable if a proper solution is needed. Moreover, there may be relations where none of the three cases of the theorem apply, in which case the theorem cannot be used for finding the asymptotic bounds of the relation.

Conclusion

In this exploration, I explored and applied the methods for solving or approximating recurrence relations to algorithm analysis, and was able to verify the accuracy of the results using experimental data. Analyzing algorithms in this manner can help us understand why an algorithm may be running slow (like in my case) or using too many resources. This can then help in improving those algorithms to make them more efficient.

There are, of course, other methods for solving recurrence relations as well which are suitable for different cases. Even for the methods explored in this exploration, there are extensions of them (such as the Akra-Bazzi method, an extension of the Master Theorem) or variations (such as backward/forward substitution, which is similar to the expansion method), which can apply to more algorithms and have their own advantages when dealing with certain types of algorithms. For example the expansion method and its variations would be useful for graph theory algorithms because they make use of induction, which is suitable for recursive structures like graphs (where, for example, the root node or the starting point on the graph can represent the base case, and the other connected nodes can represent the induction cases). It would be interesting to explore these methods and their use in other algorithms as an extension to this exploration. The methods I explored above, however, tend to be the most commonly used in algorithm analysis. They cover many different types of algorithms we usually deal with, which makes them very useful in this area.

Appendix A: ϕ and ψ

$$\phi = \frac{1+\sqrt{5}}{2}, \quad \psi = \frac{1-\sqrt{5}}{2}$$

$$\begin{aligned}
\phi \times \psi &= \frac{1+\sqrt{5}}{2} \times \frac{1-\sqrt{5}}{2} \\
&= \frac{(1+\sqrt{5})(1-\sqrt{5})}{4} \\
&= \frac{1-\sqrt{5}+\sqrt{5}-5}{5} \\
&= \frac{1-5}{4} \\
&= \frac{-4}{4} \\
&= -1
\end{aligned}$$

$$\begin{aligned}
\phi + \psi &= \frac{1+\sqrt{5}}{2} + \frac{1-\sqrt{5}}{2} \\
&= \frac{1+\sqrt{5}+1-\sqrt{5}}{2} \\
&= \frac{2}{2} \\
&= 1
\end{aligned}$$

$$\begin{aligned}
\phi^2 - \phi &= \left(\frac{1+\sqrt{5}}{2}\right)^2 - \frac{1+\sqrt{5}}{2} \\
&= \frac{(1+\sqrt{5})^2}{4} - \frac{1+\sqrt{5}}{2} \\
&= \frac{1+2\sqrt{5}+5}{4} - \frac{2(1+\sqrt{5})}{4} \\
&= \frac{1+2\sqrt{5}+5-2-2\sqrt{5}}{4} \\
&= \frac{1+5-2+2\sqrt{5}-2\sqrt{5}}{4} \\
&= \frac{4}{4} \\
&= 1
\end{aligned}$$

$$\begin{aligned}
\psi^2 - \psi &= \left(\frac{1-\sqrt{5}}{2}\right)^2 - \frac{1-\sqrt{5}}{2} \\
&= \frac{(1-\sqrt{5})^2}{4} - \frac{1-\sqrt{5}}{2} \\
&= \frac{1-2\sqrt{5}+5}{4} - \frac{2(1-\sqrt{5})}{4} \\
&= \frac{1-2\sqrt{5}+5-2+2\sqrt{5}}{4} \\
&= \frac{1+5-2-2\sqrt{5}+2\sqrt{5}}{4} \\
&= \frac{4}{4} \\
&= 1
\end{aligned}$$

$$\begin{aligned}
\phi - \psi &= \frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2} \\
&= \frac{1+\sqrt{5}-1+\sqrt{5}}{2} \\
&= \frac{2\sqrt{5}}{2} \\
&= \sqrt{5}
\end{aligned}$$

$$\begin{aligned}
\psi - \phi &= \frac{1-\sqrt{5}}{2} - \frac{1+\sqrt{5}}{2} \\
&= \frac{1-\sqrt{5}-1-\sqrt{5}}{2} \\
&= \frac{-2\sqrt{5}}{2} \\
&= -\sqrt{5}
\end{aligned}$$

Appendix B: Fibonacci Program

The following C program was used to run the fibonacci algorithm for inputs 1 to 50 and record the time taken for each input:

```c
#include <sys/time.h>
#include <stdio.h>

int fib(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}

float timedifference_msec(struct timeval t0, struct timeval t1) {
    return (t1.tv_sec - t0.tv_sec) * 1000.0f + (t1.tv_usec - t0.tv_usec) / 1000.0f;
}

int main(void) {
    struct timeval t0;
    struct timeval t1;
    float elapsed;

    for (int n = 0; ; n++) {
        gettimeofday(&t0, 0);
        int result = fib(n);
        gettimeofday(&t1, 0);

        elapsed = timedifference_msec(t0, t1);
        printf("Fib(%d): %d | ", n, result);
        printf("Took %f milliseconds.\n", elapsed);
    }
    return 0;
}
```

Appendix C: Raw Data from Fibonacci Program

Below is the raw data that was collected from the fibonacci program above (note that from $n = 1$ to $n = 28$ the time taken was too small to be recorded by the program and so some of the values have been omitted for brevity):

n	`fib(n)` [milliseconds]
1	0.000
2	0.000
3	0.000
4	0.000
5	0.000
...	...
25	0.000
26	0.000
27	0.000
28	0.000
29	9.980
30	9.615
31	10.284
32	20.237
33	35.064
34	59.703
35	100.095
36	150.171
37	249.664
38	400.094
39	655.067
40	1049.894
41	1730.158
42	2749.491
43	4499.489
44	7210.479
45	11739.970
46	18899.736
47	30669.816
48	49720.008
49	80926.422
50	131459.938

Appendix D: Computing $T(n)$ for the Fibonacci Algorithm

The following Python program was used to compute the time complexity found for the fibonacci algorithm (the constant c was assumed to be equal to 1, for simplicity):

$$T(n) = c \times \left(\frac{\Phi}{\phi}\psi^n + \frac{\Psi}{\psi}\phi^n - 1 \right)$$

```python
from math import sqrt

phi = (1 + sqrt(5)) / 2
psi = (1 - sqrt(5)) / 2

PHI = (4 + (5*phi)) / (2 + phi + sqrt(5))
PSI = (4 + (5*psi)) / (2 + psi - sqrt(5))

def rel(n):
    phiphi = PHI / phi
    psipsi = PSI / psi

    phipow = phi ** n
    psipow = psi ** n

    result = phiphi * psipow + psipsi * phipow - 1

    return result

def loop():
    x = input("Enter number:")

    if x == "psi":
        print(psi)
    elif x == "phi":
        print(phi)
    elif x == "PSI":
        print(PSI)
    elif x == "PHI":
        print(PHI)
    elif x == "q":
        return
    else:
        n = int(x)
        print(rel(n))

    loop()

if __name__ == "__main__":
    loop()
```

Appendix E: Mergesort Pseudocode

Source: Algorithmist

```
func mergesort( var a as array )
    if ( n == 1 ) return a

    var l1 as array = a[0] ... a[n/2]
    var l2 as array = a[n/2+1] ... a[n]

    l1 = mergesort( l1 )
    l2 = mergesort( l2 )

    return merge( l1, l2 )
end func

func merge( var a as array, var b as array )
    var c as array

    while ( a and b have elements )
        if ( a[0] > b[0] )
            add b[0] to the end of c
            remove b[0] from b
        else
            add a[0] to the end of c
            remove a[0] from a
    while ( a has elements )
        add a[0] to the end of c
        remove a[0] from a
    while ( b has elements )
        add b[0] to the end of c
        remove b[0] from b
    return c
end func
```

Appendix F: Mergesort Program

The following C program was used to run the fibonacci algorithm for inputs 1 to 50 and record the time taken for each input:

```c
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

float timedifference_msec(struct timeval t0, struct timeval t1) {
    return (t1.tv_sec - t0.tv_sec) * 1000.0f + (t1.tv_usec - t0.tv_usec) / 1000.0f;
}

int main() {
    struct timeval t0;
    struct timeval t1;
    float elapsed;

    for (int n = 1; ; n = n * 2 ) {
        int a[n];
        int i = n;

        for (int j = 0; j == n; j++) {
            a[j] = i;
            i--;
        }

        gettimeofday(&t0, 0);
        mergeSort(a, 0, n - 1);
        gettimeofday(&t1, 0);

        elapsed = timedifference_msec(t0, t1);
        printf("n = %d | Took: %f milliseconds\n", n, elapsed);
    }
    return 0;
}
```

Appendix G: Raw Data from Mergesort Program

Below is the raw data that was collected from the fibonacci program above:

n	`mergesort(list[n])` [milliseconds]
1	0.000
2	0.000
4	0.000
8	0.001
16	0.002
32	0.004
64	0.009
128	0.021
256	0.044
512	0.086
1024	0.198
2048	0.329
4096	0.857
8192	0.260
16384	15.950
32768	5.251
65536	43.429
131072	32.466
262144	110.398
524288	261.462

Works Cited

Algorithmist, Editors of. “Merge sort: Pseudocode”. 2019. Web. 9 Jan. 2021.

Cormen, Thomas H., et al. *Introduction to Algorithms*. Mcgraw-Hill College, 1990. Print.

HackerEarth, Editors of. “Time and Space Complexity”. Web. 9 Jan. 2021.

Levin, Oscar. *Discrete Mathematics - An Open Introduction*. 2013. Print.

Levitin, Anany. *Introduction to the Design and Analysis of Algorithms*. Pearson, 2011. Print.

Sehgal, Karuna. “A Simplified Explanation of Merge Sort”. 2018. Web. 9 Jan. 2021.

Wilf, Herbert. *generatingfunctionology*. Academic Press, Inc., 1994. Print.