Algorithms

.

1 Sequential Search (Linear Search)

Sequential Search is an algorithm that searches a list of n elements to find a specified key within the list. We will see some pseudo code of Sequential Search below that will complete one of two things.

```
1. Return the index with the key is located in the list.
2. Return a negative number is the key is not located in the list.
```

So for example if we had a list containing numbers 1-5 and we wanted to find number 1 in the list our algorithm would return index 0. If we wanted to find number 6 in the list our algorithm would return a negative number that we specified as 6 is not within the list. Below is our pseudo code for Sequential Search.

```
linearSearch(L, key):
    for i from 0 to len(L) -1:
        if L[i] == key
            return i
    return - 1
```

As you can see above the algorithm will walk through each element of the list and compare each element to the key. If the element specified at that index matches the key we return i which is the index number. After the for loop if the key is not found we return -1.

2 Binary Search

Binary Search is an algorithm that takes a sorted list of n elements and a key and checks the middle of the list to find the specified key. If the key is not found it the algorithm checks if the key is ¡ or ¿ than the middle element and will repeat the algorithm with either the lower half of the list or the upper half until the key is found or not found. Binary Search can only be done if the following are met.

```
1. List must be sorted.
```

Now let us take a look at the Binary Search pseudo code below.

```
binarySearch(L, key):
    low = 0
    high = len(L) -1
    while low <= high:
        mid = (low + high) // 2
        if key == L[mid]
```

```
         return mid
   else if key < L[mid]
         high = mid - 1
   else
         low = mid + 1
return - low - 1
```

In the above pseudo code you can see what we described on how Binary Search works with a list and the key. However, why do we return (- low - 1) and not just return - 1 like in Sequential Search? Reason that we return that value instead is that we want to figure out where to insert our key in the list. We can do that by taking the return value and using (- return - 1) and by that we have our index to insert!

3 Comparisons of Sequential Search and Binary Search

Which one of these algorithms is faster? We can check that by how many max comparisons each of these algorithms make. There can be many variables involved that can determine how fast each algorithm runs so actually implementing algorithms isn't the most efficient way to test which was one is faster. This is where checking how many comparisons each algorithms do comes in! Lets take a look at Sequential Search.

Sequential Search Comparisons

To find the number of comparisons for Sequential Search we have to see the max number of elements in the list that we are using. If n = 5 then we will at most compare 5 elements. If n = 1000 then there will be at most 1000 elements to check. below are some more examples.

```
1. n = 10, max number of comparisons = 10
2. n = 100, max number of comparisons = 100
3. n = 1000000, max number of comparisons = 1000000
```

Binary Search Comparisons

When using Binary Search the number of comparisons are signifcantly reduced. If you recall when we are finding a number in a list using Binary Search we knock off half of the possibilities evey time we run it due to how the algorithm works. How we can calculate the number of comparisons is by just using the following formula!

```
Formula: Log Base 2(n) + 1
Examples:
    1. n = 32, Log Base 2(32) = 5 + 1 = 6 comparisons
    2. n = 64, Log Base 2(64) = 6 + 1 = 7 comparisons
    3. n = 128, Log Base 2(128) = 7 + 1 = 8 comparisons
```

Taking a Look at Both

Now lets do a full comparison of both algorithms.

```
1. n = 128
    Sequential Search Comparisons = 128
    Binary Search Comparisons = 8
2. n = 1,024
    Sequential Search Comparisons = 1,024
    Binary Search Comparisons = 11
3. n = 1,048,576
    Sequential Search Comparisons = 1,048,576
    Binary Search Comparisons = 21
```

As you can see with the above examples, Binary Search takes considerably less max comparisons compared to Sequential Search. Always keep in mind though that just because one algorithm is faster doesn't necessarily mean it's the best option!

4 The Fibonacci Sequence

The Fibonacci Sequence is an algorithm that take the two preceding numbers of a sequence and adds them together. Example below shows how it works.

n	0	1	2	3	4	5	6
fib(n)	0	1	1	2	3	5	8

So our main problem that we are trying to solve using this algoriithm is we would want to find the nth fibonacci number. There are two different ways we can express the Fibonacci Sequence in algorithms.

```
1. Recursive
2. Dynamic (Iterative)
```

Below is the code for the recursive algorithm for the Fibonacci Sequence.

```
def fibRecursive(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibRecursive(n - 1) + fibRecursive(n - 2)
```

Overall this recursive algorithm is very inefficient. Now why is this algorith inefficient? It is ineffiecient because you are doing the same kind of work over and over again. Below is the number comparisons that would be made for n amount of elements.

```
n = 0, number of terms compaired = 1
n = 1, number of terms comapared = 1
n = 2, number of terms compared = 3
n = 3, number of terms compared = 5
n = 4, number of terms compared = 9
n = 5, number of terms comapared = 15
n = 6, number of terms compared = 25
```

As you can see there is no tell tale pattern, but you can also see if we increase n by 2 then the number of terms more than doubles making it inefficient. Now lets look at the dynamic algorithm for the Fibonacci Sequence.

```
def fibDynamic(n):
    if n == 0:
       return 0
    if n == 1:
       return 1
   fib = [0, 1]
   for i in range(1, n + 1):
       fib.append(fib[i -1] + fib[i - 2]
   return fib[n]
```

With this version of the Fibonacci Sequence algorithm it builds dynamically and will return your value once you reach it taking less comparisons as well.

Proof by Induction

As we stated in the recursive form of the algorithm lets see how the number of terms computed more than doubles when we double the amount of elements.

```
Let T(n) terms computed for fib(n)
    1. T(n) > 2(T((n-2))
    2. T(n) > 2 x 2 x T(n-4)           #when n goes down by two we more than double
    3. T(n) > 2 x 2 x 2 x T(n-6)
    .
    .
    .  T(n) > 2 x 2 x 2 ..... 2 x T(0) #the amount of 2's we have would be (n/2) as
                                        we're going down by 2 everytime
       = 2^(n/2) x T(0)
       = 2^(n/2)
```

What all this means basically is that to calculate T(n), again the number of terms needed to compute the nth fibonacci number, is greater than $2^{n/2}$. Now lets prove this by proof of induction.

Claim: For n >= 2, T(n) $>2^n/2$

Base Case: n = 2, T(2) = 3, $2^2/2 = 2^1 = 2$

3 >2

Our base case is true!

Base Case 2: n = 3, T(3) = 5, $2^3/2 = 2^3/2 = 2.8$

5 >2.8

Our base case 2 is also true!

Inductive Hypothesis: Assume for all m, 2 <= m <n, T(m) $>2^m/2$

Induction Steps:

To show that T(n) $>2^{(n/2)}$

T(n) = T(n - 1) + T(n - 2) + 1

$>2^{(n-1)/2} + 2^{(n-2)/2} + 1$

$>2^{(n-1)/2} + 2^{(n-2)/2}$

$>2^{(n-2)/2} + 2^{(n-2)/2}$

$= 2(\ 2^{((n-2)/2)})$

$= 2^{((n-2)/2)+1}$

$= 2^{((n-2)/2)+(2/2)}$

$= 2^{((n-2+2)/2)}$

$= 2^{n/2}$

As you see we were able to get $2^{n/2}$.