TEMPLETON SCHOLARSHIP PROGRAM 2022

COMPUTER SCIENCE

Evaluating the efficiency of different searching algorithms

U.Arjun

4 July 2022

Contents

1 Algorithms

1.1 What are algorithms?

Whilst the word algorithms sounds like some sort of complex computer related jargon, it simply means a series of steps or processes that are executed in order to fulfil a task. For example, your morning routine is also an algorithm. It's a set of steps that you complete in order to get ready. Let's say that you brush your teeth, wash your face, have breakfast and then get changed for school or work. You wouldn't complete those steps in a different order such as getting changed and then eating your breakfast because it would be inefficient or cause further delays, perhaps if you spill some milk on your shirt or if there's a fruit stain on your blazer. Thus, we ourselves have developed an algorithm to get ready in the morning. Another example may be using an algorithm that someone else created, such as baking a cake. You, the user, will be executing the set of steps in order to achieve an output which would be the cake in this case.

Computer algorithms operate in the same way as us humans, they fulfil a series of tasks to achieve an output that are set out by us humans.

```python
def main():
    x = 4
    y = 6
    x = x + y
    answer = x / 2

    return answer
```

Listing 1: Basic algorithm

If we look at this algorithm, the computer starts by assigning the variables x, y with 4, 6 respectively. It then adds the two variables and stores the value to the variable x. Finally, it divides the variable x by 2 and stores this value in the variable answer and returns this. Although this is a very basic, useless algorithm, it helps us understand how the computer executes the algorithm. However, if the computer hadn't executed each line in order, then we wouldn't get the right output. For instance, if it divided x by 2 before adding it to y, then we would get 8 whereas we expect the value of 5 to be returned.

1.2 Why do we use algorithms?

There are a countless number of algorithms and they all have various purposes and we can modify or create algorithms to suit our purposes but ultimately, we use algorithms to save time. By following the same set of steps, we are almost guaranteed to achieve our output given that there are no external factors that affect the output. A route optimisation algorithm finds the quickest way home for us, while considering numerous factors such as traffic, road closures and speed limits. This is much faster than us taking a certain route home, to find that the road may be closed, or if there is traffic, or perhaps if there's a faster route. Algorithms also ensure that we are maximising our resources because the algorithm is optimised to be efficient. For example, certain algorithms may be used minimise CPU usage which makes the computer more efficient. We can also reduce cost by using an algorithm. Continuing with the use of route optimisation algorithms, we can choose the shortest route determined by the algorithm to reduce fuel costs. More often than before, algorithms are being used in machine-learning and AI to allow computers to optimise the algorithm to maximise efficiency.

On the face of it, algorithms seem quite niche to the computer science industry but when you delve in further, you quickly realise that there's more to it than some lines of code. I read an article from The Times written in 2012, and Charles Duhigg described the uses of algorithms in targeted marketing. Target, the example looked at by the writer, used algorithms to notice patterns and habits in people and specifically, pregnant women. Target generates a guest ID for all customers and all transactions, enquiries, emails and everything else related to that customer is then linked to that guest ID. This will also include details such as age, gender, marital status and more. They then used an algorithm to track the items that pregnant women bought a couple of months after they are pregnant such as lotion and supplements like zinc, magnesium and calcium. This would indicate that the women is a few months into her pregnancy and when they start buying towels, cotton balls and hand sanitizer, the women would be in her final few months of pregnancy according to this study. The statistician, Andrew Poole, recognized this pattern and used it to implement an algorithm that would anticipate when a women was going to have a baby and assign each women a "pregnancy prediction". This data would later be used to influence their targeted marketing to certain people such as advertising baby strollers, clothes and food amongst the other products. But they did so in a discrete way, by placing the baby clothes opposite the lawnmower or putting an offer on diapers opposite a wineglass to mask their targeted advertising. Soon enough the algorithm was implemented and sales started to grow by manipulating people to think that they need something when in reality, they may not need it now. And so, this shows us how algorithms are being used nowadays, and it's for much more than finding the quickest way home.

There are many different types of algorithms and the most common ones include searching algorithms, path finding algorithm, sorting algorithms, compression algorithms, tree and graph-based algorithms, pattern matching algorithm among many others. Today, I'm going to be focusing on searching algorithms and finding the most efficient one while considering the time and space complexity of each algorithm.

2 Searching Algorithms

2.1 What do they do?

Searching algorithms are used to find items in a list or file as fast as possible. Over the years, more and more searching algorithms have been devised in an effort to maximise efficiency. For example, if you were asked to find a word in a book, then most likely, you would read the book until that specific word is found. This is known as a linear search and one of the searching algorithms that I'll be looking at today. However, when using a dictionary, we don't read each word until we find the word we want. Instead we use an algorithm. We understand that the dictionary is alphabetically sorted, and therefore we can use this knowledge to skip through the book and go to the initial letter of the word we're looking at. From there, we would repeat the process until we find the word we want. This is an example of a searching algorithm because we are using a series of steps to find an item in a list.

2.1.1 Big O Notation

While I am describing the efficiency of a algorithms, I will be using the Big O notation which is used to reflect the time and space complexity of the algorithms. It's important to be familiar with this notation to understand the key differences between programs.

«Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. It is a member of a family of notations invented by Paul Bachmann, Edmund Landau, and others, collectively called Bachmann–Landau notation or asymptotic notation.»

-The definition on Wikipedia's page for Big O Notation. Big O shows how the time taken for an algorithm changes with the number of items in that list and it is also used to represent how much extra space is taken by the algorithm.

$$O(n) = Linear$$

where n is the number of items in the list. For instance, if the list has 100 items, then there will be 100 iterations in the worst case scenario of a linear search.

$$O(n^2)$$

similarly, n represents the number of items but this time the time taken for the algorithm can be up to n^2 of the number of items in that list. So if the list has 100 items, then the algorithm could take up to 100^2 * the time taken for a single item or 10,000 * the time taken for a single item.

Ultimately, the Big O notation is simply a tool used to depict the bounds of how long an algorithm takes with respect to the number of items that are in the item.

2.2 List of searching algorithms

Searching algorithms are divided into sub-categories which are sequential search and interval search. Sequential search describes algorithms that search each value in a linear method whereas

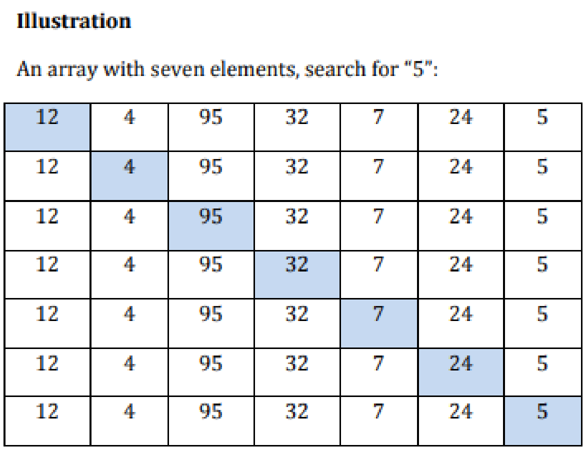
interval search refers to searching the list at certain intervals such as binary search.
However, it's important to remember that different algorithms will have different pre-requisites and cannot function without those conditions being met. For example, for most interval search algorithms, the list must be ordered for the list to function which can cause hassle and take time so in fact the time complexity would include the time taken to sort the list.

2.2.1 Linear Search

Linear search is the most basic searching algorithm and does nothing more than just checking each item one-by-one until the item is found or if there are no more items in the list. It's similar to someone looking through a page for a specific word. They would read each word until they have found the word, or if there are no more words on that page. It's a very easy algorithm to implement and although it's quite inefficient, the simplicity of it makes it much quicker to code and implement than other more efficient algorithms.It works on unordered lists as well which boasts the versatility of this algorithm.

Algorithm 1 Linear Search Pseudocode

```
$pointer \leftarrow 0$
while $pointer \leq len(list) - 1$ do:
    if $list[pointer] == desiredItem$ then:
        return index($desiredItem$)
    else
        $pointer = pointer + 1$
    end if
        return -1
end while
```



Illustration

An array with seven elements, search for "5":

12	4	95	32	7	24	5
12	4	95	32	7	24	5
12	4	95	32	7	24	5
12	4	95	32	7	24	5
12	4	95	32	7	24	5
12	4	95	32	7	24	5
12	4	95	32	7	24	5

Figure 1: A visual representation of the linear search algorithm

As we can see in the pseudo-code above, this algorithm will keep checking each item in the list until the index number is greater than the length of the list - 1 which means that there are no items left in the list to check and the program will return -1 to inform the user that the item has not been found.

Best time complexity: O(1)
Worst time complexity: O(n)
Space complexity: O(1)

2.2.2 Maximal Search

Maximal search, or also known as maximum search, is a subsidiary of linear search but its scope is very limited. It uses a linear search algorithm to return the smallest or largest value in a list.

Algorithm 2 Maximal Search Pseudocode

```
$largestValue \leftarrow 0$
for $value$ in list do
    if value > largest value then
      $largestValue \leftarrow value$
    end if
end for
      return largestValue
```

The algorithm is very similar to a linear search, and can only be used to find the largest or smallest value in a list. This makes it very impractical and limited in scope.

Best time complexity: O(n)
Worst time complexity: O(n)
Space complexity: O(2)

It's worth noting here that both the best time complexity and the worst time complexity are the same. This is because even if the largest/smallest value is found, the algorithm has to check all the remaining value to verify this. In addition, the space complexity is now 2 rather than just one because the algorithm must keep two values in memory - the current largest value and the value being compared.

2.2.3 Recursive program to linearly search an element in a given array

This algorithm is also a variation of the linear search algorithm however, the difference is that this algorithm carries out a linear search from both ends of the list simultaneously in each iteration. For example, in the first iteration, the algorithm will compare the first and last item to the desired item. If the desired item is found, then the index is returned. Else, the second and second to last item are compared. The algorithm will iterate through until the index of the desired item is returned or if the pointers cross over in the sense that the back pointer is smaller than the front pointer.

Algorithm 3 Recursive program to linearly search

```
$frontPointer \leftarrow 0$
$backPointer \leftarrow len(list) - 1$
for x in $len(list)/2$ do
    if backPointer > frontPointer then
      return -1
    end if
    if array[frontPointer] $==$ desiredItem then
      return frontPointer
    end if
    if array[backPointer] $==$ desiredItem then
      return backPointer
    end if
    $frontPointer \leftarrow frontPointer + 1$
    $backPointer \leftarrow backPointer - 1$
end for
```

The same concept of the linear search is used here, except it's carried out from two fronts so the probability of getting a best case scenario is doubled and it would take half the iterations to check all the values. But it's possible that the desired value is right in the middle, so all values would have to be checked.

Best time complexity: O(1)
Worst time complexity: O(n)
Space complexity: O(3)

The space complexity is now 3 because the algorithm now has to store the two values being compared to the desired value. When dealing with larger numbers that take up more bits to store, this will result in slower performance because the memory needed to run the algorithm will be tripled.

2.2.4 Binary Search

Binary Search is the first of the algorithms that require a sorted list to work. However, this overhead can be justified with the reduced run time. The basis of the algorithm is halving the number of values that have to be searched with each iteration. During an iteration, the middle value is compared to the desired value and half of the list is removed based on whether the desired value is larger or smaller than the middle value.

Algorithm 4 Binary Search

```
$frontPointer \leftarrow 0$
$backPointer \leftarrow len(list) - 1$
while $frontPointer! = backPointer$ do
    $midPointer = (frontPointer + backPointer/2)$
  if midPointer == desiredItem then
    return midPointer
  end if
  if desiredItem > midPointer then
    backPointer = midPointer + 1
  else
    frontPointer = midPointer + 1
  end if
end while
    return -1
```

Best time complexity: O(1)
Worst time complexity: O(log n)
Space complexity: O(1)

We now see the time complexity begin to differ and the binary searching algorithm has a time complexity of log n because with each iteration, the list size is halved and so ultimately, the time taken will increase as the number of items increases, but at the same time, the effectiveness of the algorithm to eliminate half the possible values also increases. This means that the binary search algorithm isn't particularly effective for small data sets, because only a limited number of numbers will be omitted and fundamentally the efficiency of the algorithm comes from the ability to reduce the number of comparisons. A list with 200 elements will take 8 comparisons to find the desiredItem whereas a list with 10 elements will take 4 comparisons.

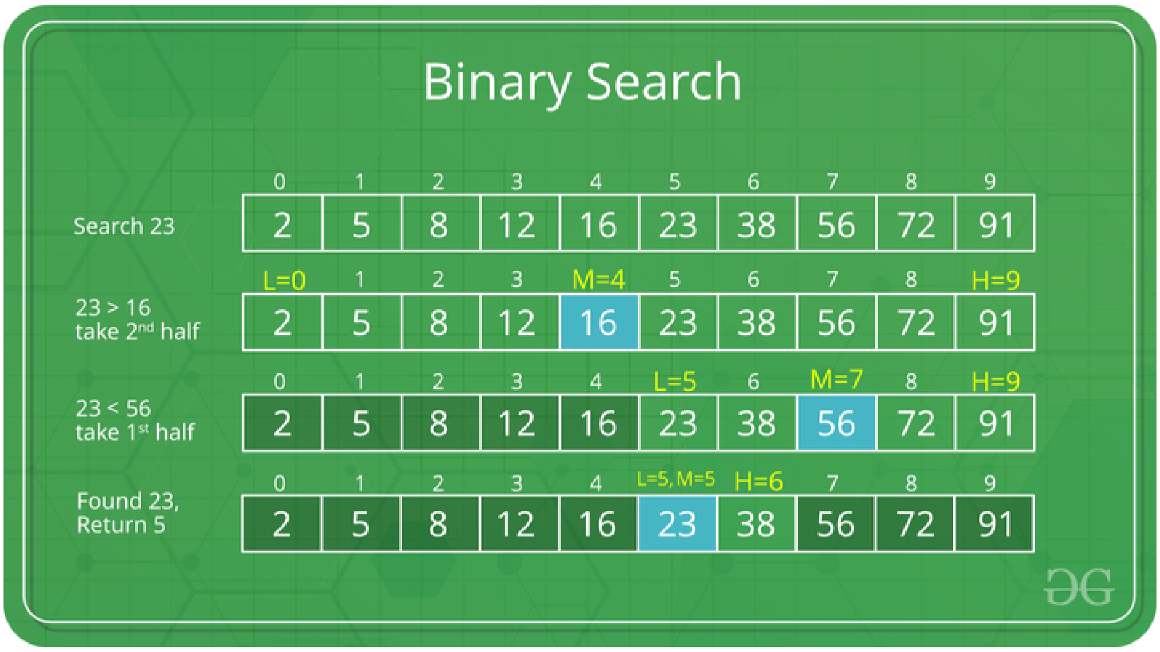


Figure 2: Visualisation of binary search algorithm

2.2.5 Hybrid Search

The hybrid search is a less commonly known searching algorithm and it incorporates both the linear search algorithm and the binary search algorithm to provide an improved version of the linear search algorithm and this search will work on unordered lists unlike the binary search algorithm. It begins by dividing the array into 2 and then comparing the first,middle and last element against the desiredItem and if there's no match, then using a linear search in between the starting and middle element and repeating this process for the other side as well. There's a small difference compared to the normal linear search because the searching takes place in a different order but ultimately, there's no difference in time or space. It's similar to the functions of maths where both multiplication and division have no priority over each other although in various cases, it may be easier to use one oppose to the other. In the case of our two algorithms, it's a matter of where the desiredItem is located but there's no way that the user could know this prior to the search.

Best time complexity: O(1)
Worst time complexity: O(n)
Space complexity: O(4)

The space complexity has further increased for this algorithm because it has to store 3 values - low, mid and high alongside the desiredItem value. This algorithm also has a lot of condition statements which will further increase the time taken for each iteration, making it quite an inefficient algorithm in terms of practicality.

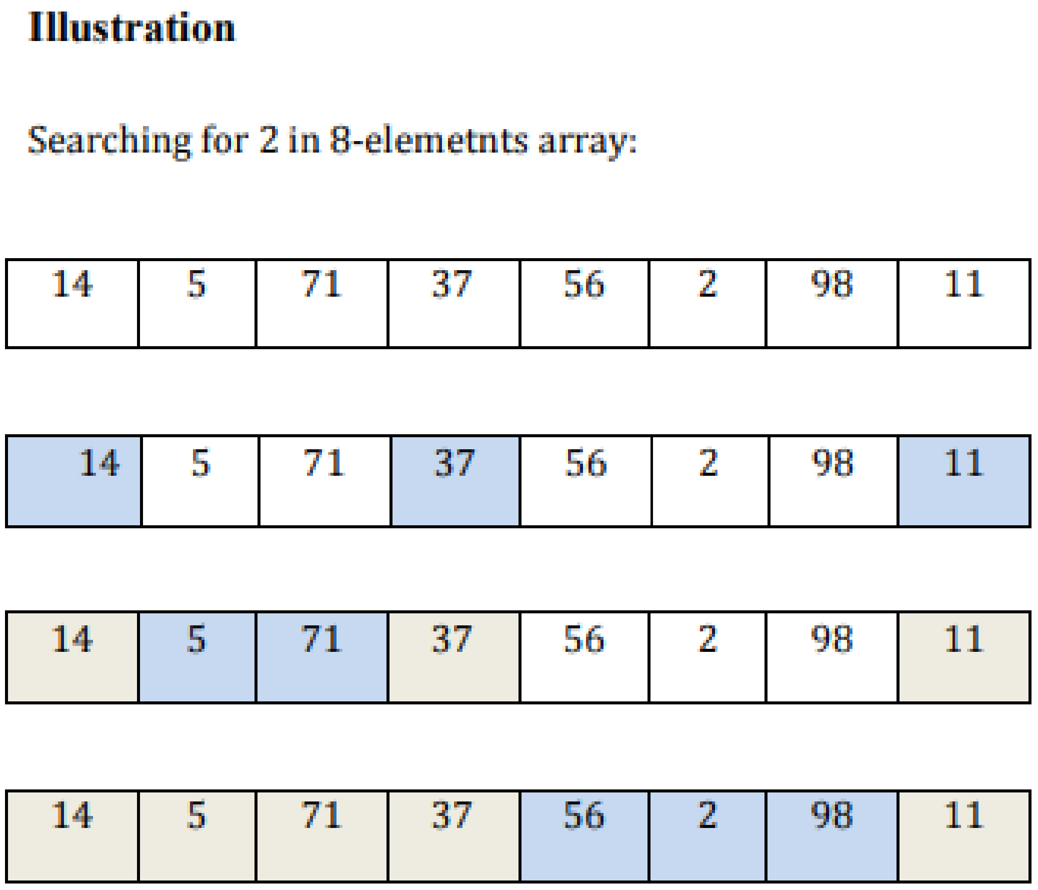


Figure 3: A visual representation of the hybrid search algorithm

Algorithm 5 Hybrid Search

```
$low \leftarrow 0$
$high \leftarrow len(list) - 1$
procedure HybridSearch (list, low, high, desiredItem)(
    )$mid = (low + high)/2$
   if mid == desiredItem then
    return mid
   end if
   if low == desiredItem then
    return low
   end if
   if high == desiredItem then
    return high
   end if
   if $low \geq high - 2$ then
    return -1
   else
    p = procedure HybridSearch(list,low + 1, mid - 1, desiredItem)
      if $p == -1$ then
    p = procedure HybridSearch(list, mid + 1, high - 1, desiredItem)
    return p
      end if
   end if
    return -1
end procedure
```

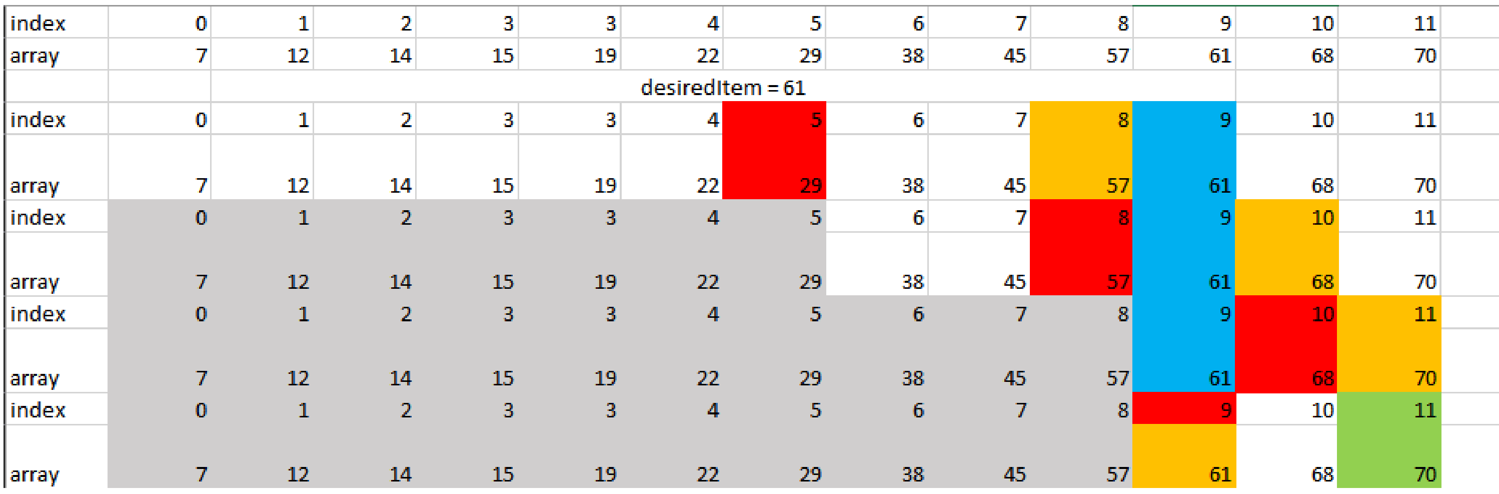
2.2.6 Fibonacci Search

As the name suggests, this is a searching algorithm that makes use of the Fibonacci sequence and it is an interval search based algorithm. Although the basis of the algorithm is quite similar to binary search, there are some similarities and differences.
Similarities:

- Only works on sorted lists
- The average time complexity is Log n time which is the same as the binary search.
- Fibonacci Search is also a Divide and Conquer algorithm which means that the list is broken down into smaller lists and "Conquered" or searched in other words.

Differences:

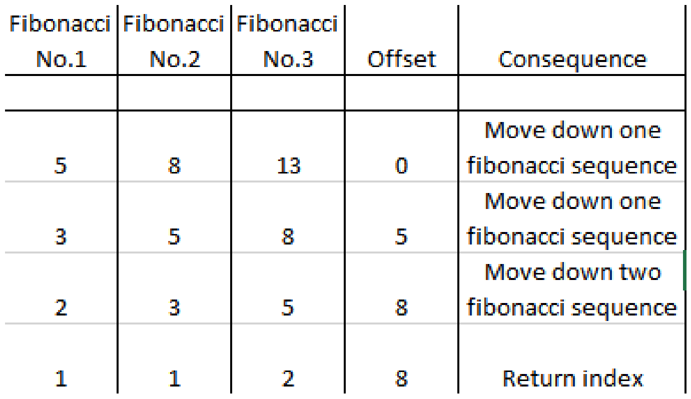
- Fibonacci Search breaks down the list into unequal sizes unlike the Binary Search that divides the list into two equal parts each time.
- While the Binary Search uses a division operator to create the divisions in the lists, the Fibonacci Search only uses the + and - operators which can result in different time complexities for different CPUs.
- The Fibonacci Search is also lighter on memory so it is ideal for lists or arrays that are too large for CPU cache or in RAM because it compares elements that are closer together which improves the efficiency of the program because less time is wasted retrieving items from memory. However, on modern computers and recent improvements in memory, this difference can be negligible.



index	0	1	2	3	3	4	5	6	7	8	9	10	11
array	7	12	14	15	19	22	29	38	45	57	61	68	70
						desiredItem = 61							
index	0	1	2	3	3	4	5	6	7	8	9	10	11
array	7	12	14	15	19	22	29	38	45	57	61	68	70
index	0	1	2	3	3	4	5	6	7	8	9	10	11
array	7	12	14	15	19	22	29	38	45	57	61	68	70
index	0	1	2	3	3	4	5	6	7	8	9	10	11
array	7	12	14	15	19	22	29	38	45	57	61	68	70
index	0	1	2	3	3	4	5	6	7	8	9	10	11
array	7	12	14	15	19	22	29	38	45	57	61	68	70

Figure 4: A self designed representation of the Fibonacci Search Algorithm

Whilst the diagram below may look like a heap of mashed up colours and numbers, it depicts how the Fibonacci search algorithm works. First, the largest Fibonacci number equal to or greater than the number of elements in the list is found. In our case, that would be 13 because it is equal to the number of items in the list. We want the number 61 from the array and it is at index 9.Now we take the two preceding Fibonacci numbers which are 5 and 8 which make up the number 13 in the sequence. The algorithm looks at the number at index 5 (29) which is highlighted in red. 29 $<$ 61 and so we discard all numbers below and including index 5 which is represented by the grey shading.In the second iteration, we move one down the Fibonacci sequence to the numbers 3,5 and 8 but we move are focus to the numbers higher than the 29 at index 5. So we compare the numbers at index location 8,10 and 13. At index 8 we see the number 57 which is still lower than 61. Once again, we move down the sequence once and eliminate all numbers below index 8. In iteration 3, we are at numbers 2,3 and 5 in the Fibonacci sequence and our offset, or the numbers we are discarding, is at 8. This means that the algorithm considers the numbers at index 10,11 and 13. At index 10, we have the number 68 which is greater than 61. So we move our sequence numbers down by 2 and our sequence numbers are 1,1 and 2. Combining this with our offset, we get index 9,9 and 10. After looking at index 9, we have reached our target number of 61 and the algorithm would return the index 9. The table below summarizes how the Fibonacci Numbers change and how the offset differs between iterations.



Fibonacci No.1	Fibonacci No.2	Fibonacci No.3	Offset	Consequence
5	8	13	0	Move down one fibonacci sequence
3	5	8	5	Move down one fibonacci sequence
2	3	5	8	Move down two fibonacci sequence
1	1	2	8	Return index

```
procedure FIBONACCISEARCH($array, desiredItem, len(array)$)
    $fibNum2 \leftarrow 0$
    $fibNum1 \leftarrow 1$
    $fibNum0 \leftarrow 0$
    while fibNum0 $<$ len(array) do
     $fibNum2 = fibNum1$
     $fibNum1 = fibNum0$
     $fibNum0 = fibNum2 + fibNum1$
    end while
    while fibNum0 $>$ 1 do
     $i = min(offset + fibNum0, len(array) - 1)$
        if array[i]$<$ desiredItem then
     $fibNum0 = fibNum1$
     $fibNum1 = fibNum2$
     $fibNum2 = fibNum0 - fibNum1$
     $offset = i$
        end if
        if  array[i]$>$ desiredItem then
     $fibNum0 = fibNum2$
     $fibNum1 = fibNum1 - fibNum2$
     $fibNum2 = fibNum0 - fibNum1$
        else return i
        end if
        if $fibNum1 AND array[offset + 1] == desiredItem$ then
     return -1
        end if
    end while
end procedure
```

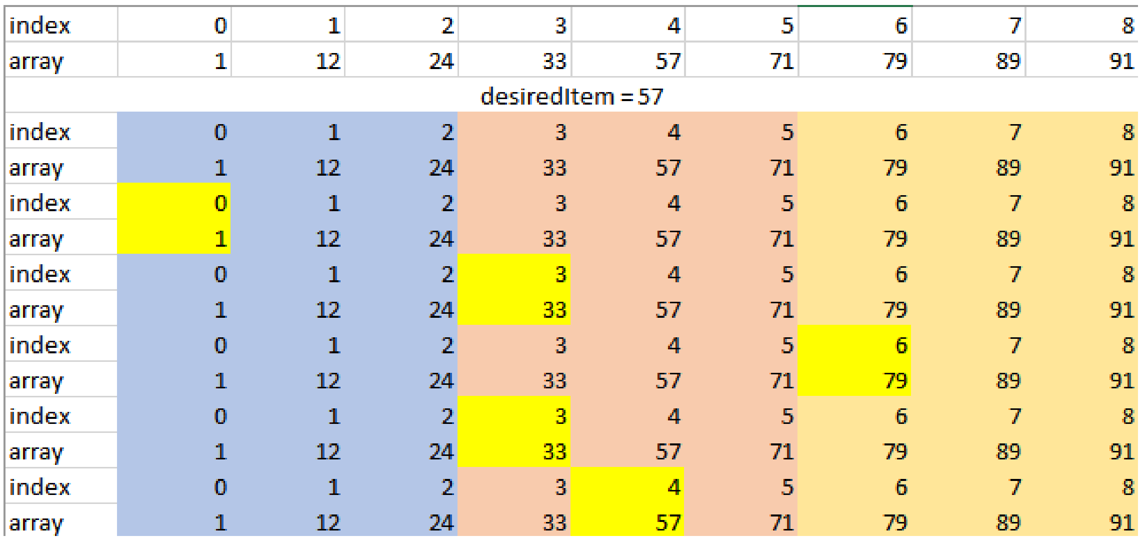
Best time complexity: O(1)
Average time complexity: O(log n)
Space complexity: O(1)

2.2.7 Jump Search

The basis of the jump search is very similar to binary search except that rather than breaking the list down into only two small arrays, it will break down the array into smaller fixed sizes. Then the algorithm jumps from index 0 to the first section. For instance if we had a list with 16 elements in it, then we could break down the list into 4 smaller lists with 4 elements in. Then the algorithm jumps from index 0 to index 4 which is the starting point of the next list. The number at index 4 is compared to our desired element and if it's larger then we jump to the next section or the search performs a linear search in that list. In terms of efficiency, jump search is more efficient than linear search but less efficient than binary search. It's hard to determine what the perfect block size is for each section but generally, a block size of

$$\sqrt{len(array)}$$

is ideal to work with because it means that all the sections are of equal size. The yellow box



index	0	1	2	3	4	5	6	7	8
array	1	12	24	33	57	71	79	89	91
				desiredItem = 57					
index	0	1	2	3	4	5	6	7	8
array	1	12	24	33	57	71	79	89	91
index	0	1	2	3	4	5	6	7	8
array	1	12	24	33	57	71	79	89	91
index	0	1	2	3	4	5	6	7	8
array	1	12	24	33	57	71	79	89	91
index	0	1	2	3	4	5	6	7	8
array	1	12	24	33	57	71	79	89	91
index	0	1	2	3	4	5	6	7	8
array	1	12	24	33	57	71	79	89	91
index	0	1	2	3	4	5	6	7	8
array	1	12	24	33	57	71	79	89	91

Figure 5: A visual representation of the jump search algorithm

indicates the number that's being compared against the desiredItem. First, the algorithm separates the list into equal sized section and because the list has 9 items, it has been broken down into 3 lists with 3 items in them. Then the algorithm compares the first item in each section against the desiredItem which is 57 in this example. So the computer compares the number 1 and 57 and because 57 is larger, it compares the next number which is at the front of the second section. 33<57 so it moves onto the next section. 79>57 and so the algorithm goes back to the previous section and performs a linear search on that section until the element is found or not found. However, there are some limitations of this program and one of which is that, like binary search, the list must be ordered for the algorithm to work and yet it's still less efficient than binary search.

Best time complexity: O(1)
Average time complexity: O(sqrt n)
Space complexity: O(1)

The jump search algorithm is only used where the binary search would be costly, such as if the desiredItem is just bigger or smaller than a middle value, but like I mentioned before, it is unlikely that the user would know this information preceding the search.

```
$x \leftarrow sqrt(len(array))$

for $i$ in range $x$ do
    if array[i * x] == desiredItem then
     return i*x
    end if
    if array[i*x] > desiredItem then
        for j do in array[((i-1)*x):(i*x)]:
            if j == desiredItem: then
     return j
            end if
        end for
    end if
end for
```

2.2.8 Interpolation Search

These last two algorithms are said to be improvements of the binary search algorithm which is the most efficient algorithm we have looked at so far. Interpolation search focuses on narrowing the search space to minimize the number of comparisons needed. This is done with the formula below.

$$pos = lo + [\frac{(x-arr[lo])*(hi-lo)}{(arr[hi]-arr[Lo])}]$$

x = Element to be found/ desiredItem
lo = Starting index of the array
hi = Ending index of the array

The algorithm calculates the value of pos in a while loop and uses this information to dictate the next step. If pos == desiredItem, then the index is returned or the next pos value is calculated. If pos < desiredItem, then the next pos value is calculated for the left sub-array or it is calculated for the right sub-array. This process is repeated until a match is found or there if there are no values left in the sub-array.

```python
def interpolationSearch(arr, lo, hi, x):

    # Since array is sorted, an element present
    # in array must be in range defined by corner
    if (lo <= hi and x >= arr[lo] and x <= arr[hi]):

        # Probing the position with keeping
        # uniform distribution in mind.
        pos = lo + ((hi - lo) // (arr[hi] - arr[lo]) *
                    (x - arr[lo]))

        # Condition of target found
        if arr[pos] == x:
            return pos

        # If x is larger, x is in right subarray
        if arr[pos] < x:
            return interpolationSearch(arr, pos + 1,
                                       hi, x)

        # If x is smaller, x is in left subarray
        if arr[pos] > x:
            return interpolationSearch(arr, lo,
                                       pos - 1, x)
    return -1
```

Listing 2: Interpolation algorithm

The above code from GeeksforGeeks is an implementation of the interpolation algorithm and reflects how the algorithm may be used. As we can see, the average time complexity has further been reduced from the log(n) of binary search but just like binary search, it requires a sorted list to function.

Best time complexity: O(1)
Average time complexity: O(log(log(n)))
Space complexity: O(1)

2.2.9 Exponential Search

This algorithm has a time complexity of log(n) which seems ironic compared to its name but the name exponential comes from the way it searches. It uses an exponential method to compare elements to the desiredItem. For example, it starts at index 0 then 1,2,4,8,16 and so on. This avoids the need to spend time breaking down the list into sections like Fibonacci, Binary and Jump Search. If the numbers at those indices are higher than the desiredItem, then a binary search is used to find the item in between the two indices. This combines the efficiency of the binary search and reduces the number of comparisons needed to narrow the test case. The best case scenario is only possible if the item is found at index 0 but this is the same as other algorithms like linear search.

Best time complexity: O(1)
Average time complexity: O((log(n))
Space complexity: O(log(n))

```python
# A recursive binary search function returns
# location  of x in given array arr[l..r] is
# present, otherwise -1
def binarySearch( arr, l, r, x):
    if r >= l:
        mid = l + ( r-l ) // 2

        # If the element is present at
        # the middle itself
        if arr[mid] == x:
            return mid

        # If the element is smaller than mid,
        # then it can only be present in the
        # left subarray
        if arr[mid] > x:
            return binarySearch(arr, l,
                                mid - 1, x)

        # Else he element can only be
        # present in the right
        return binarySearch(arr, mid + 1, r, x)

    # We reach here if the element is not present
    return -1

# Returns the position of first
# occurrence of x in array
def exponentialSearch(arr, n, x):
    # IF x is present at first
    # location itself
    if arr[0] == x:
        return 0

    # Find range for binary search
    # j by repeated doubling
    i = 1
    while i < n and arr[i] <= x:
        i = i * 2

    # Call binary search for the found range
    return binarySearch( arr, i // 2,
                         min(i, n-1), x)
```

Listing 3: Exponential algorithm

3 Implementation

3.1 Test Case

Now I'm going to test the efficiency of the following algorithms to see how noticeable the theoretical differences are. To make this a fair test, I will be using a sorted list of 100,000 numbers.I had considered using a list of 10,000 words, however it would be hard to determine one word as larger or smaller than another. It's worth noting that I am coding all of these algorithms, so there may be minor inefficiencies in my code and this could result in slight differences in time taken, however I will optimise my code as much as possible to avoid this. I will be searching for the numbers 19035,96421,56789,12345 and 99999.

3.1.1 Linear Search

```
import time
list = []
with open("numberlist.txt") as f:
    list = f.read().splitlines()
def linearSearch(lookup):
    for x in list:
        if x == lookup:
            return list.index(x)
start_time = time.perf_counter()
linearSearch("19035")
linearSearch("96421")
linearSearch("56789")
linearSearch("12345")
linearSearch("99999")
print("this took {}s in total".format(time.perf_counter()-start_time))
```

this took 0.010617099998853519s in total

3.1.2 Binary Search

```
import time
list = []
with open("numberlist.txt") as f:
    list = f.read().splitlines()
def binarySearch(lookup):
    low = 0
    high = 100000
    mid = 0
    found = False
    while found == False:
        mid = int(((high + low)//2))
        if int(list[mid]) == int(lookup):
            found = True
            return mid
        elif int(list[mid]) > int(lookup):
            high = mid
        elif int(list[mid]) < int(lookup):
            low = mid

start_time = time.perf_counter()
binarySearch("19035")
binarySearch("96421")
binarySearch("56789")
binarySearch("12345")
binarySearch("99999")
print("this took {}s in total".format(time.perf_counter()-start_time))
```

this took 7.389999882434495e-05s in total

3.1.3 Fibonacci Search

```
import time
list = []
with open("numberlist.txt") as f:
  list = f.read().splitlines()
def fibonacciSearch(lookup):
    fibNum2 = 0
    fibNum1 = 1
    fibNum0 = 0
    while fibNum0 < 100000:
        fibNum2 = fibNum1
        fibNum1 = fibNum0
        fibNum0 = fibNum2 + fibNum1
    offset = -1

    while fibNum0 > 1:
        i = min(offset + fibNum2, 100000-1)
        if (list[i] < lookup):
            fibNum0 = fibNum1
            fibNum1 = fibNum2
            fibNum2 = fibNum0 - fibNum1
            offset = i

        elif (list[i] > lookup):
            fibNum0 = fibNum2
            fibNum1 = fibNum1 - fibNum2
            fibNum2 = fibNum0 - fibNum1

        else:
            return i

    if (fibNum1 and list[100000-1] == lookup):
        return 100000-1

    return -1

start_time = time.perf_counter()
fibonacciSearch("19035")
fibonacciSearch("96421")
fibonacciSearch("56789")
fibonacciSearch("12345")
fibonacciSearch("99999")
print("this took {}s in total".format(time.perf_counter()-start_time))
```

this took 4.530000023805769e-05s in total

3.1.4 Jump Search

```
import time
list = []
with open("numberlist.txt") as f:
  list = f.read().splitlines()

import math
def jumpSearch(lookup,n):
    step = math.sqrt(n)
    prev = 0
    while list[int(min(step,n)-1)] < lookup:
        prev = step
        step += math.sqrt(n)
        if prev >= n:
            return -1

    while list[int(prev)] < lookup:
        prev += 1

        if prev == min(step,n):
            return -1

    if list[int(prev)] == lookup:
        return prev

    return -1

start_time = time.perf_counter()
jumpSearch("19035",100000)
jumpSearch("96421",100000)
jumpSearch("56789",100000)
jumpSearch("12345",100000)
jumpSearch("99999",100000)
print("this took {}s in total".format(time.perf_counter()-start_time))
```

this took 0.0002620000000206346s in total

3.1.5 Interpolation Search

```
import time
list = []
with open("numberlist.txt") as f:
  list = f.read().splitlines()

def interpolationSearch(low,high,lookup):
    if low <= high and lookup >= list[low] and lookup <= list[high]:
        position = low + ((high - low)// list[high] - list[low]* (lookup- list[low
    ]))

        if list[position] == lookup:
            return position

        if list[position] < lookup:
            return interpolationSearch(position+1, high, lookup)

        if list[position] > lookup:
            return interpolationSearch(low, position-1, lookup)

    return -1

start_time = time.perf_counter()
interpolationSearch(0,100000-1,"19035")
interpolationSearch(0,100000-1,"96421")
interpolationSearch(0,100000-1,"56789")
interpolationSearch(0,100000-1,"12345")
interpolationSearch(0,100000-1,"99999")
print("this took {}s in total".format(time.perf_counter()-start_time))
```

this took 6.499999926745659e-06s in total

3.1.6 Exponential Search

```
import time
list = []
with open("numberlist.txt") as f:
  list = f.read().splitlines()

def binarySearch(l, r, lookup):
    if r >= l:
        mid = l + ( r-l ) // 2

        if list[mid] == lookup:
            return mid

        if list[mid] > lookup:
            return binarySearch(l, mid - 1, lookup)

        return binarySearch(mid + 1, r, lookup)

    return -1

def exponentialSearch(lookup):
    if list[0] == lookup:
        return lookup

    i = 1
    while i < 100000 and list[i] <= lookup:
        i = i*2
    return binarySearch(i//2,min(i,100000-1),lookup)

start_time = time.perf_counter()
exponentialSearch("19035")
exponentialSearch("96421")
exponentialSearch("56789")
```

```python
exponentialSearch("12345")
exponentialSearch("99999")
print("this took {}s in total".format(time.perf_counter()-start_time))
```

```
this took 0.0004189999996530823s in total
```

4 Evaluation

By just comparing the numbers, we can understand that interpolation search was the most efficient considering that it took the least time with just 6.50e-6s (3s.f.) and even though we are searching 100,000 numbers, all the algorithms managed to finish in under 0.1s so to the naked eye, the differences would be indistinguishable. As expected, linear search took the longest time with 0.0106s and while that may not seem long for the average user, this sort of search is done in large scale projects with much larger numbers and so the time taken would multiply significantly. Having said that, it's obvious these algorithms are very efficient in their purposes and while it may seem like linear search is useless compared to the other algorithms, when handling small sized data sets, it's much easier for the user to use the linear search algorithm and the difference in efficiency will be invisible, even to a computer system. This is because when the number of items decrease significantly, it's inevitable that all other algorithms will have to resort to checking all items or carrying out a linear search. For example, for a binary search with 8 items, it will take 3 comparisons at most to end up at a result whereas a linear search could take 8 comparisons at most. This means that the linear search will take 8/3 * the time, but the time taken for 3 comparisons will be hard to record because of its insignificance. Interpolation search is about 15,000 times faster than linear search according to my research but at the same time, there could be the overhead of sorting items which linear search, exclusively, overcomes.
Another obstacle that I overcame, was that certain algorithms would only work with numbers because one item would have to be determined as larger or smaller than another item, and when using items that had words, this couldn't be done. After having coded the linear search pseudo-code, I moved on to the binary search pseudo-code, but quickly realized that I would have to change my approach to testing because while linear search accommodated for the possibility of searching for words in a list of words, binary search couldn't reenact this. My initial plan of testing was to use a list of 10,000 words found online, but I had to scrap this idea and use numbers instead to ensure that all the algorithms would function. Thus, another thing to consider about algorithms is the flexibility of them.
While writing this report, I have learnt that there is more to algorithms than just choosing a linear search or a binary search. It has also taught me about the Big O notation which is not only something exclusive to searching algorithms, but also found in maths and used to denote the time taken for algorithms or processes. Moreover, when selecting an algorithm to use, it's also important to understand the space complexities of programs to maximise efficiency. What I mean by this is that when dealing with much larger numbers or items, it's likely that several of these items may need to be stored in memory when using a particular algorithm, but if these numbers or items do not fully fit inside cache, then the program will have to keep returning to RAM to retrieve and update variables, and this can cost time. This report has also taught me the importance of high efficiency code, and while there's no such thing as bad code, there's a noticeable difference between well-optimized and refined code versus inefficient but functional code. I began coding all of the algorithms above using purely my knowledge in Python and my pseudo code, but when I compared the time taken to the model code found on professional websites like GeeksForGeeks, immediately, I noticed areas where my code faltered and caused unnecessary delays. Thus, for the purposes of my investigation, I used the code found on GeeksForGeeks website for the more complex searching algorithms to fairly conduct the experiment.

Thank you for reading

5 Bibliography

‘Binary Search’. GeeksforGeeks, 28 Jan. 2014, https://www.geeksforgeeks.org/binary-search/.

Duhigg, Charles. ‘How Companies Learn Your Secrets’. The New York Times, 16 Feb. 2012. NYTimes.com, https://www.nytimes.com/2012/02/19/magazine/shopping-habits.html.

‘Exponential Search’. GeeksforGeeks, 17 Feb. 2017, https://www.geeksforgeeks.org/exponential-search/.

‘Fibonacci Search’. GeeksforGeeks, 9 Dec. 2015, https://www.geeksforgeeks.org/fibonacci-search/.

Hickman, Leo. ‘How Algorithms Rule the World’. The Guardian, 1 July 2013. The Guardian, https://www.theguardian.com/science/2013/jul/01/how-algorithms-rule-world-nsa.

‘Interpolation Search’. GeeksforGeeks, 14 Oct. 2016, https://www.geeksforgeeks.org/interpolation-search/.

‘Jump Search’. GeeksforGeeks, 14 Oct. 2016, https://www.geeksforgeeks.org/jump-search/.

‘Linear Search’. GeeksforGeeks, 20 Oct. 2016, https://www.geeksforgeeks.org/linear-search/.

Mulongo, Cleophas. ‘The Importance of Algorithms in Computer Programming’. Technotification, 9 Feb. 2019, https://www.technotification.com/2019/02/importance-of-algorithms-programming.html.

‘Recursive Program to Linearly Search an Element in a given Array’. GeeksforGeeks, 25 May 2014, https://www.geeksforgeeks.org/recursive-c-program-linearly-search-element-given-array/.

Reed, Niamh. ‘What Is an Algorithm? An “in a Nutshell” Explanation’. ThinkAutomation, 29 Jan. 2019, https://www.thinkautomation.com/eli5/what-is-an-algorithm-an-in-a-nutshell-explanation/.

‘Searching Algorithms’. GeeksforGeeks, https://www.geeksforgeeks.org/searching-algorithms/. Accessed 6 July 2022.

‘What Is Big O Notation Explained: Space and Time Complexity’. FreeCodeCamp.Org, 16 Jan. 2020, https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/.

Zia, Ahmad Shoaib. A Survey on Different Searching Algorithms. Sharda University Greater Noida, 1 Jan. 2020, https://www.irjet.net/archives/V7/i1/IRJET-V7I1275.pdf.

Sultana, Najma and Chandra, Sourabh and Paira, Smita and Alam, Sk. (2017). A Brief Study and Analysis of Different Searching Algorithms. https://www.researchgate.net/publication/314175061_A_Brief_Study_and_Analysis_of_Different_Searching_Algorithms

(PDF) Comparative Analysis of Search Algorithms

A Comparative Analysis of Three-with-cover-page-v2.pdf

IRJET- A Survey on Different Searching Algorithms

Performance Analysis of Searching Algorithms in C

A-comparative-analysis-of-searching-algorithms.pdf

3.1.3 Searching Algorithms. https://bournetocode.com/projects/GCSE_Computing_Fundamentals/pages/3-1-3-searc_alg.html. Accessed 6 July 2022.

A Serial Search Algorithm. http://theteacher.info/index.php/algorithms-and-problem-solving-3/2-3-algorithms/2-3-1-algorithms/searching-algorithms/3311-serial-and-binary-searching-algorithms. Accessed 6 July 2022.

'Algorithmic Efficiency'. Wikipedia, 1 Apr. 2022. Wikipedia,https://en.wikipedia.org/w/index.php?title=Algorithmic_efficiency&oldid=1080535147.

GCSE Computer Science - 2.1.3 Searching and Sorting Algorithms. https://sites.google.com/horburycomputing.co.uk/gcse-computing/2-1-algorithms/2-1-3-searching-and-sorting-algorithms. Accessed 6 July 2022.

'Isaac Computer Science'. Isaac Computer Science, https://isaaccomputerscience.org/topics/searching?examBoard=all&stage=all. Accessed 6 July 2022.

Isaac Computer Science, https://isaaccomputerscience.org/concepts/dsa_search_searching_compared?examBoard=all&stage=all. Accessed 6 July 2022.

'Measuring an Algorithm's Efficiency | AP CSP (Article)'. Khan Academy, https://www.khanacademy.org/computing/ap-computer-science-principles/algorithms-101/evaluating-algorithms/a/measuring-an-algorithms-efficiency. Accessed 6 July 2022.

Ross, Joel. Chapter 7 Searching and Efficiency | Introduction to Programming. infx511.github.io, https://infx511.github.io/. Accessed 6 July 2022.

'Searching Algorithms'. GeeksforGeeks, https://www.geeksforgeeks.org/searching-algorithms/. Accessed 6 July 2022.

Sorting, Searching and Algorithm Analysis — Object-Oriented Programming in Python 1 Documentation. https://python-textbok.readthedocs.io/en/1.0/Sorting_and_Searching_Algorithms.html. Accessed 6 July 2022.

Unit 5 Lab 1: Search Algorithms and Efficiency, Page 4. https://bjc.edc.org/bjc-r/cur/programming/5-algorithms/1-searching-lists/4-efficiency.html?topic=nyc_bjc%2F5-algorithms.topic&course=bjc4nyc.html&novideo&noassignment. Accessed 6 July 2022.

'Why Do We Need Searching Algorithms? - Searching - KS3 Computer Science Revision'. BBC Bitesize, https://www.bbc.co.uk/bitesize/guides/zgr2mp3/revision/1. Accessed 6 July 2022.