Lecture 8 - Searching

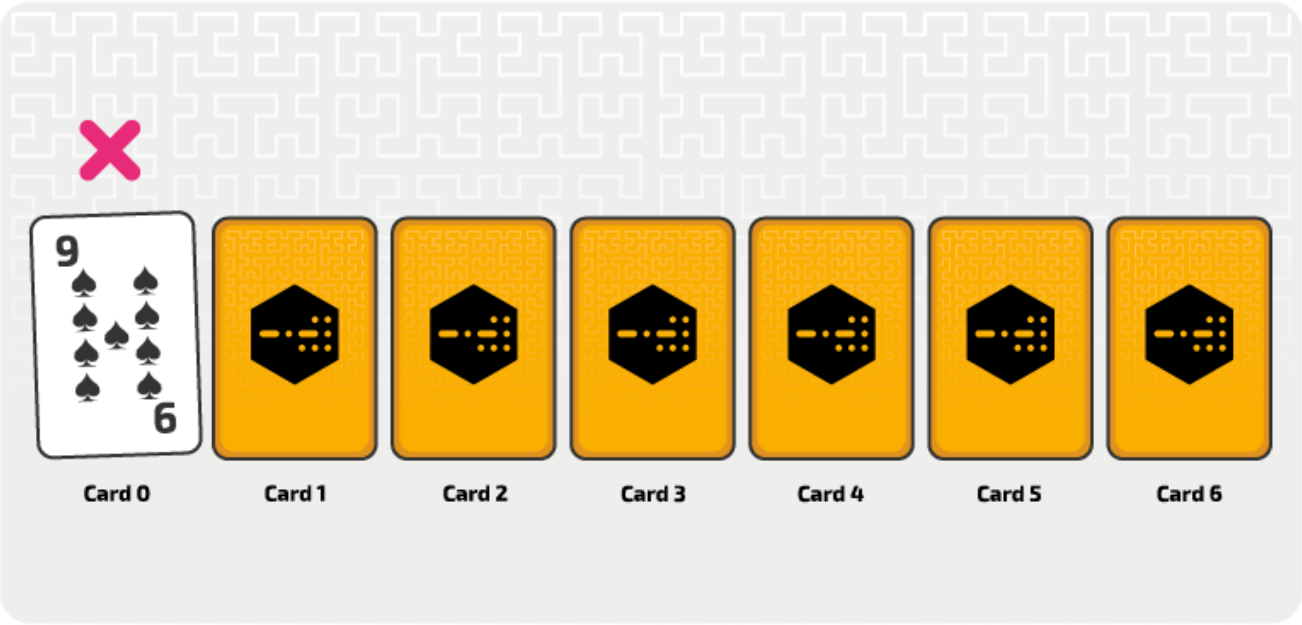
Linear Search

The only methodical way to find a specific item in an unordered list of items or a sequence of data is to look at every item in the list, one after another, and check if it is what you are looking for. The algorithm for this process is called linear search, because it involves starting from the beginning of the list and checking one item at a time to see if it's the right one.

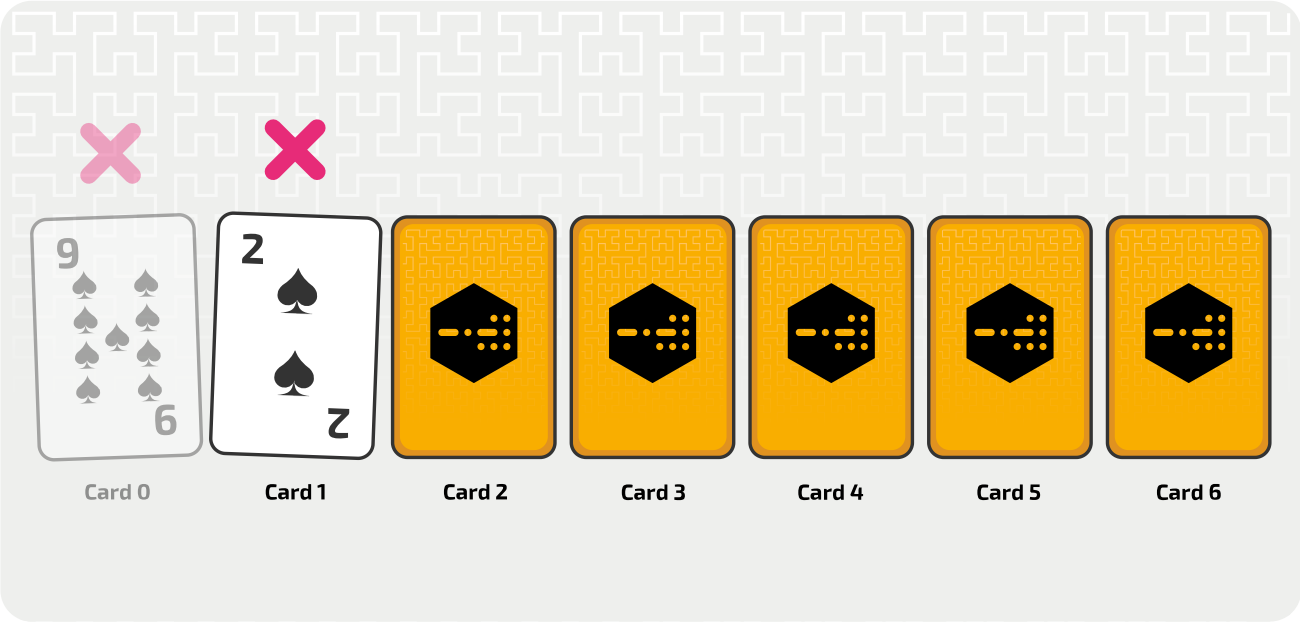
For example, suppose that you have seven playing cards placed face down and arranged randomly. Notice that the first card is 'Card 0' and the last one 'Card 6'. This is done on purpose because in computer science the first position in a list or array is position 0.

You are searching through the cards for the four of spades. How would you do that using the linear search? You would have to check the cards one by one until you find the card you are looking for or you don't find it at all.

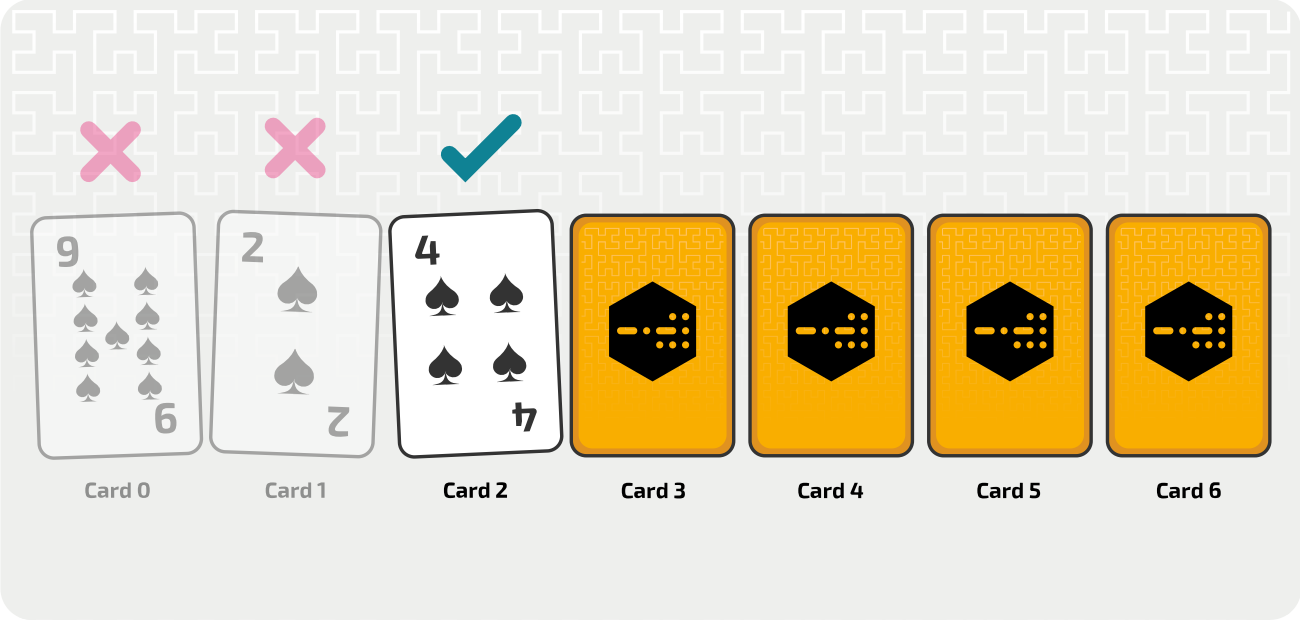
The following figure shows that you check the first card (Card 0). It is not the four of spades so you go to the next card.



The next figure shows that you check the second card (Card 1). It is not the four of spades so you go to the next card.



The next figure shows that you check the third card (Card 2). It is the four of spades! You have found the card that you are looking for so you stop searching.



Notice that if the four of spades wasn't one of the seven cards you were checking, you would have to check all seven cards to confirm that it's not there.

Thus our worse case complexity for this is $O(n)$.

The steps of linear search

The steps for performing a linear search can be described as follows:

- Step 1: Take a list of data and an item that is being searched for (the search item)
- Step 2: Starting from the first position in the list, repeat steps 3–5 until you find the search item or until the end of the list is reached:
- Step 3: Compare the item at the current position in the list (index) to the search item
- Step 4: If the item at the current position (index) is equal to the search item, then stop searching
- Step 5: Otherwise, go to the next position in the list

Linear Search (`for` loop)

The first version of the linear search algorithm to consider iterates through the whole list from start to end, even if the search item is found before reaching the end of the list.

In the code that follows, you will see that the function `linear_search_version_1()` takes two arguments:

- The list of items (`items`)
- The item that you are looking for (`search_item`)

If the search item is found, the function will return the position (`found_index`) of the item in the list. Otherwise, the function will return a value of `-1` which will indicate that the item was not found. Remember that the first item in the list will have an index of `0`.

```
def linear_search_version_1(items, search_item):

    # Initialise the variable
    found_index = -1

    # Repeat until the end of the list is reached
    for current in range(len(items)):

        # Compare the item at the current index to the search item
        if items[current] == search_item:
            # If the item has been found, store the current index
            found_index = current

    # Return the index of the search_item or -1 if not found
    return found_index
```

This implementation of the linear search is inefficient because the loop will keep repeating until it reaches the end of the list, even after the item is found. This means that, if there are 5,000 items in the list, all 5,000 will be inspected even if the item you are looking for is found near the start of the list. This is a waste of computational resources.

Tracing the algorithm using a trace table

Consider the following list of items, stored in a list.

items							
index	0	1	2	3	4	5	6
value	11	25	10	29	15	13	18

To trace the linear search algorithm, you will need to keep track of the values of the main variables that are used. A trace table is essential for documenting this process in a systematic way. The table below shows the steps of searching for the value 15 in the array.

The value of `found_index` is initialised to `-1` and is only changed when `search_item` is found. This is the value that is returned by the function.

On each iteration of the `for` loop, the value of `current` is incremented by `1`. The final value for `current` is `6` as the loop iterates through the whole array, even though in this example `search_item` is found at position `4`.

When `search_item` is found, the value of `found_index` is set to `current`, i.e. `4`.

When the for loop has completed its final iteration, the value of `found_index` is returned by the function.

found_index	current	items[current]	return value
-1			
	0	11	
	1	25	
	2	10	
	3	29	
4	4	15	
	5	13	
	6	18	
			4

Linear Search (`while` loop)

The for loop implementation of the linear search algorithm is inefficient. You can make the algorithm more efficient by replacing the for loop with a while loop. This condition-controlled loop will execute while the item has not yet been found and there are still items in the list to examine. The loop will terminate when the item is found, or the end of the list is reached.

In the code that follows, you will see that the function `linear_search_version_2()` takes two arguments:

- The list of items (items)
- The item that you are looking for (`search_item`)

A boolean **flag** is used to indicate whether (or not) the item is found. The flag variable is named `found`. It is initialised to `False` (the item has not yet been found). If the item is found, the value of the flag is set to `True`. The use of a flag is a common technique for controlling a while loop. Giving the flag variable a meaningful identifier (name) helps to make your code self-documenting.

```
def linear_search_version_2(items, search_item):

    # Initialise the variables
    found_index = -1
    current = 0
    found = False

    # Repeat while the end of the list has not been reached
    # and the search item has not been found
    while current < len(items) and found == False:

        # Compare the item at the current index to the search item
        if items[current] == search_item:
            # If the item has been found, store the current index
            found_index = current
            found = True # Raise the flag to stop the loop

        current = current + 1 # Go to the next index in the list

    # Return the index of the search_item or -1 if not found
    return found_index
```

Tracing the algorithm using a trace table

Consider the following list of items, stored in a list.

items							
index	0	1	2	3	4	5	6
value	11	25	10	29	15	13	18

To trace the improved version of the linear search algorithm, you will use the same table to trace the algorithm. This time you will search for the value `29` (in the list).

- The value of found_index is initialised to `-1` and is only changed when the `search_item` is found.
- The value of found is initialised to `False` and is changed only when the `search_item` is found.
- The `while` loop runs while there are still items left in the list and the item has not been found. The value of `current` is incremented by `1` on each iteration of the loop.
    - The `search_item` is found when the variable `current` has the value `3`. The value of found is set to `True` and the value of `found_index` is set to the value of `current`, i.e. `3`.
    - It is important to observe that the value of current is incremented by `1` one more time after the search term is found as it is incremented at the end of the code block within the loop.
- When the `while` loop completes (in this case when the value of found is `True`), the value of `found_index`, i.e. `3`, is returned.

found_index	current	found	items [current]	return value
−1	0	False		
	0		11	
	1		25	
	2		10	
3	3	True	29	
	4			3

Linear Search Time Complexity

To determine the time complexity of the linear search algorithm, you must consider the main operation. The main operation here is the comparison between each item in the list and the item being searched for. Thus, the time complexity is directly related to the number of items in the list that you have to search through.

In the first version of the algorithm that uses a for loop, the comparison is always carried out n times, where n is the number of items in the list. In Big O notation, this algorithm's time complexity can be described as $O(n)$; this is known as **linear time** complexity.

In the second version of the algorithm, the use of the flag and a while loop makes the algorithm more efficient.

- In the **worst case**, the main operation will still be carried out n times. If the item sought is found at the end of the list, or is not found, all of the items will need to be examined. We can therefore say that at the worst case the time complexity of the algorithm is $O(n)$.
- In the **best case**, the time complexity will be $O(1)$. This will be achieved if the item you are looking for is the first item in the list.
- The **average case** time complexity is difficult to calculate unless you assume that the distribution of data in the list is uniform and the possibility of the search item being in each position in the list is equally likely. In this case, to calculate the **average case** time complexity consider how many comparisons you would have to perform on average to find an item in the list.

If the item you are looking for is in the first position, then you will find it after 1 comparison

If the item you are looking for is in the second position, then you will find it after 2 comparisons

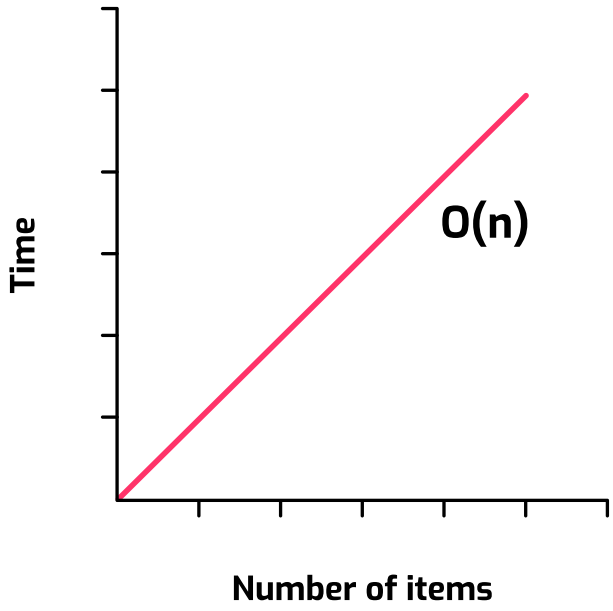
...

If the item you are looking for is in the last position, then you will find it after n comparisons

---

In order to find the average you need to add up the total number of comparisons and divide it by the number of positions in the list which is n. This evaluates to $\frac{1+2+3+\cdots+n}{n} = \frac{n+1}{2}$. However, this will still be expressed as $O(n)$ as the constants become less significant as the size of the input (n) grows.

If you are asked only to state "time complexity", you should give the worst case which in $O(n)$.



Linear Search Space Complexity

Irrespective of the size of the list (the size of n), the algorithm for linear search just maintains a few key variables such as `current` and `found_index`.

This does not change as n gets larger.

Thus the space is fixed, it is constant, it is $O(1)$.

Summary

- Linear search loops over all items.
  - Does not require **ordered** data
  - Worse-case time complexity - $O(n)$
  - Best-case time complexity - $O(1)$
  - Space complexity - $O(1)$

Acknowledgement

Searching Alogrithms - Isaac Computer Science.