Lecture 8 - Searching

Binary Search

NOTE: This only works with an ordered list of items

A binary search is a very efficient searching algorithm that works only if you have an **ordered** list of items or sequence of data. An ordered list is a list that has been sorted or is maintained in a specific order. A list can be ordered **ascending** (low to high) or **descending** (high to low). Depending on the way a list is ordered, the algorithm will need to be adjusted to ensure that it focuses on the correct range of items.

The following algorithm works for lists that are in **ascending** order (low to high):

- The algorithm calculates the **midpoint** position and checks the item that is stored at this position. The midpoint is the position in the middle of the list.
    - If the item at the midpoint position is the item you are searching for, the search can finish; you have found what you are looking for.
    - If the item at the midpoint position is **less than** the item that you are searching for, you can focus on the items after the midpoint item and ignore all the items before (and including) the item at the midpoint.
    - If the item at the midpoint position is **greater than** the item that you are looking for, you can focus on the items before the midpoint item and ignore all the items after (and including) the item at the midpoint.
- Then, the algorithm takes the remaining items of the list and repeats the process. Notice that each time the algorithm repeats this process, the range of items that is being checked is halved in size.

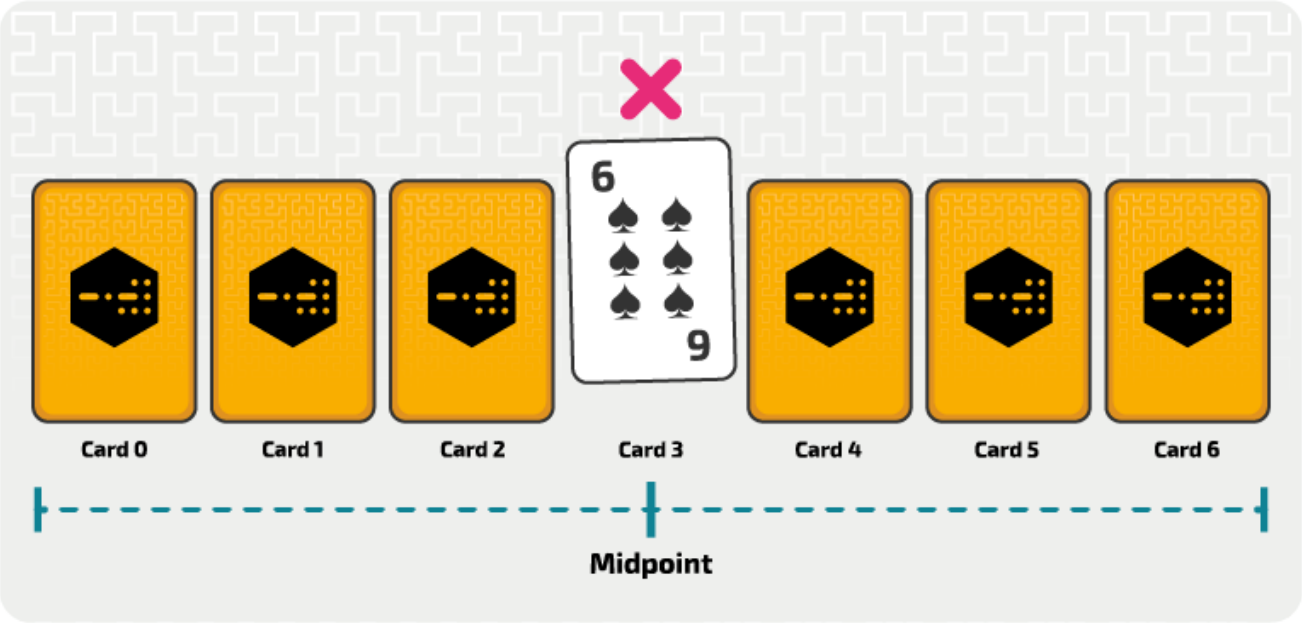
The concept of **less than** and **greater than** can sometimes be tricky.

- If the item you are looking for is a numeric value, the items in the list will be ordered by value (typically from lowest to highest).
- If the item you are looking for is a string value, the items in the list will be ordered by character code (typically alphabetically).

For example, suppose that you are searching for a specific playing card in a set of seven playing cards. The cards have been sorted in numeric order (from lowest to highest) and placed face down. 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.

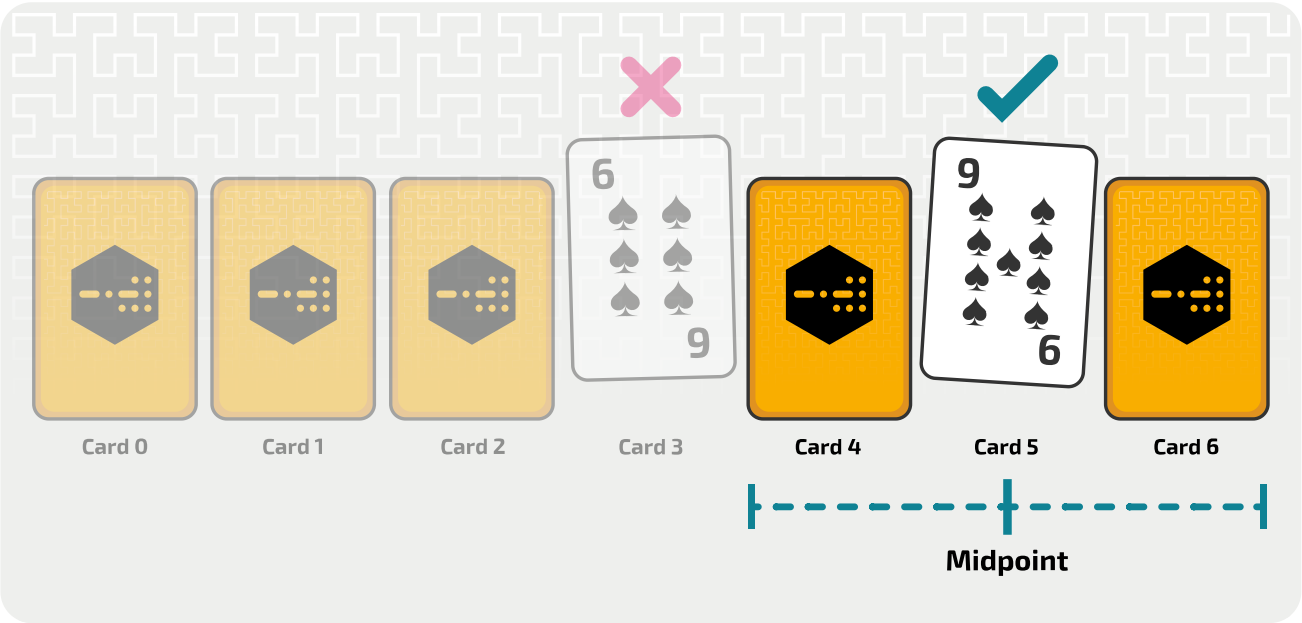
You are searching through the cards for the nine of spades. How would you do that using the binary search? You would have to check the cards at the midpoints until you find the card you are looking for or you don't find it at all.

The following figure shows that you check the card at the first midpoint. The first midpoint is the position in the middle of the cards, which corresponds to Card 3. The card you have found is the six of spades. This isn't the card you are searching for.



Card 3 is the six of spades, which is less than the nine of spades, so you ignore the cards before (and including) the six of spades. You then check the card at the midpoint of the remaining cards (Cards 4, 5, and 6).

The next figure shows that you check the card at the second midpoint. The second midpoint is the position in the middle of the remaining cards, which corresponds to Card 5. It is the nine of spades! You found the card that you are looking for so you stop searching.



The steps of binary search

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

- Step 1: Set the search range to be the entire list of ordered items.
- Step 2: Repeat steps 3–6 until you find the search item or there are no more items to check (the range is empty):
- Step 3: Find the item at the midpoint position (in the middle of the range).
- Step 4: Compare the item at the midpoint position to the search item.
- Step 5: If the item at the midpoint is equal to the search item, then stop searching.
- Step 6: Otherwise,
    - If the item < search item, change the range to focus on the items after the midpoint.
    - if the item > search item, change the range to focus on the items before the midpoint.

There are different ways to implement the binary search algorithm. For the following practice examples, we use the iterative binary search algorithm that is presented in the next section.

This code implementation stores the current `first` position (index) and the `last` position (index). The formula `midpoint = (first + last)//2` to calculate the midpoint position. This means that:

- If there is an odd number of items in the range, the midpoint is the position of the **middle** item in the range
- If there is an even number of items in the range, the midpoint is the position of the **middle-left** item in the range

How to perform a binary search with a list of numbers

Let's look at how the algorithm is performed using a list of numbers. Here is a list that contains the number of medals won by the top seven ranking countries at the 1998 Winter Olympics in Nagano, Japan. Notice that the list is ordered, which means that the numbers appear in ascending numerical order (i.e. lowest to highest) in the list.

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

- First, take an ordered list of data and an item that is being searched for (the search item).
    - The search item is 25.
- Then, maintain a range of items where the search item might be found.
    - Initially, set the range to be the entire list.

Initially, set the range to be the entire list.

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

- Next, find the item at the midpoint position: the midpoint position is 3, which contains the value 15.
    - Remember that if there is an odd number of items in the range, the midpoint is the position of the middle item in the range
- Compare the item at the midpoint to the search item: 15 is less than 25 (1st comparison).

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

- If the item at the midpoint is less than the search item, change the range to focus on the items after the midpoint. To do that, move the start of the range to the position after the midpoint.

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

- Find the item at the next midpoint position: the new midpoint position is 5, which contains the value 25.
- Compare the item at the midpoint to the search item: 25 is equal to 25 (2nd comparison).

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

- If the midpoint item is equal to the search item, then stop searching.

The algorithm has completed:

- The search item, 25, was found at position 5 of the list.
- The algorithm performed two comparisons in total before the search item was found.

Binary Search (iterative)

The standard version of the binary search algorithm uses a loop to iterate through the process of checking a range of items (and ignoring the remaining items in the list) until the item is found or there are no more items to check. The following code works for lists that are in ascending order (low to high).

The function takes two arguments:

- `items` is the ordered list of items
- `search_item` is the item being searched for

A flag (`found`) is used to indicate whether (or not) the item is found. 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`.

- The variable `first` is used for the index of the first item of the range that is being checked
- The variable `last` is used for the index of the last item of the range that is being checked
- The variable `midpoint` is used for the index of the item that is positioned in the middle of the range that is being checked

To calculate the midpoint position, the algorithm uses the formula `midpoint = (first + last) // 2`.

This formula adds the index of the first and last item in the list and performs an integer or floor division (DIV) by 2. The use of the integer or floor division ensures that the result of the division is a whole number, which is what is needed, since it corresponds to the position of the middle item in the range. Notice that:

- If there is an odd number of items in the range, the integer or floor division (DIV) will result in the position of the middle item in the range
- If there is an even number of items in the range, the integer or floor division (DIV) will round down to the position of the middle-left item in the list

It then checks if the item is less than, greater than or equal to the midpoint:

- Less than. Set `first` to be the `midpoint + 1`. Focus on the items after the midpoint
- Greater than. Set `last` to be the `midpoint - 1`. Focus on the items before the midpoint
- Equal. We have found the item, it is the midpoint

Python Code

```
def binary_search(items, search_item):

    # Initialise the variables
    found = False
    found_index = -1
    first = 0
    last = len(items) - 1

    # Repeat while there are still items between first and last
    # and the search item has not been found
    while first <= last and found == False:

        # Find the midpoint position (in the middle of the range)
        midpoint = (first + last) // 2

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

        # Check if the item at the midpoint is less than the search item
        elif items[midpoint] < search_item:
            # Focus on the items after the midpoint
            first = midpoint + 1

        # Otherwise the item at the midpoint is greater than the search item
        else:
            # Focus on the items before the midpoint
            last = midpoint - 1

    # Return the position 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	10	11	13	15	18	25	29

To trace the binary 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 18 in the array.

- The value of found_index is initialised to `-1` and is changed inside the `while` loop to get the value of the variable `midpoint`. The variable `midpoint` has the position (i.e. the index) of the midpoint item in the list.
- The value of found is initialised to False and is changed only when `search_item` is found.
- The `while` loop runs while there are still midpoints left to check in the range that is being searched and the item has not been found.
- The value of `midpoint` is calculated using the formula `midpoint = (first + last) // 2`.
    - When the variable `midpoint` takes the value `3`, the item in position `3` is `15`, which is less than `18`. The algorithm focuses on the items greater than the midpoint, and so `first` becomes `4`.
    - When the variable `midpoint` takes the value `5`, the item in position `5` is `25`, which is greater than `18`. The algorithm focuses on the items less than the midpoint, and so `last` becomes `4`.
    - When the variable midpoint takes the value `4`, the item in position `4` is `18`, which is equal to `18`. The `search_item` is found. The value of found is set to `True` and the value of `found_index` is set to the value of midpoint, i.e. `4`.
- When the `while` loop completes (in this case when the value of `found` is `True`), the value of found_index, i.e. `4`, is returned.

found_index	first	last	midpoint	items [midpoint]	found	return value
−1	0	6	3	15	False	
	4					
	4	6	5	25	False	
		4				
4	4	4	4	18	True	
						4

Binary Search (recursive)

The second version of the binary search algorithm uses recursion. It takes the same approach as the iterative version in splitting the list at the mid-point and discarding half at each comparison.

The function takes four arguments:

- `items` is the ordered list
- `search_item` is the item being searched for
- `first` is the index of the first item of the range that is being checked
- `last` is the index of the last item of the range that is being checked

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`.

- The variable `midpoint` is used for the index of the item that is positioned in the middle of the range that is being checked.

The algorithm checks if the item is less than, greater than or equal to the midpoint:

- Less than. Set `first` to be the `midpoint + 1`. `return` a call to the function with the current values of `items`, `search_item`, `first` and `last`
- Greater than. Set `last` to be the `midpoint - 1`. Focus on the items before the midpoint. `return` a call to the function with the current values of `items`, `search_item`, `first` and `last`
- Equal. We have found the item, it is the midpoint. `return` the `midpoint`

```
def binary_search_recursive(items, search_item, first, last):

    # Base case for recursion: The recursion will stop when the
    # index of the first item is greater than the index of last
    if first > last:
        return -1 # Return -1 if the search item is not found

    # Recursively call the function
    else:
        # Find the midpoint position (in the middle of the range)
        midpoint = (first + last) // 2

        # Compare the item at the midpoint to the search item
        if items[midpoint] == search_item:
            # If the item has been found, return the midpoint position
            return midpoint

        # Check if the item at the midpoint is less than the search item
        elif items[midpoint] < search_item:
            # Focus on the items after the midpoint
            first = midpoint + 1
            return binary_search_recursive(items, search_item, first, last)

        # Otherwise the item at the midpoint is greater than the search item
        else:
            # Focus on the items before the midpoint
            last = midpoint - 1
            return binary_search_recursive(items, search_item, first, last)
```

There is an alternative recursive version that takes two arguments. A list of items (which is halved on each recursive call) and the search item. In this alternative version, the value of `last` is determined, on each call, by considering the size of the list.

Why not try and code it.

Tracing the algorithm using a trace table

Consider the following list of items, stored in a list. Again, you will search for the value 3.

`items`						
index	0	1	2	3	4	5
value	1	3	7	8	9	13

Call	`first`	`last`	`midpoint`	`items [midpoint]`	`return midpoint`	**Line where next recursive call is made**
1	0	5	2	7		
	0	1				28
2			0	1		
	1	1				22
3			1	3	1	
2					1	
1					1	

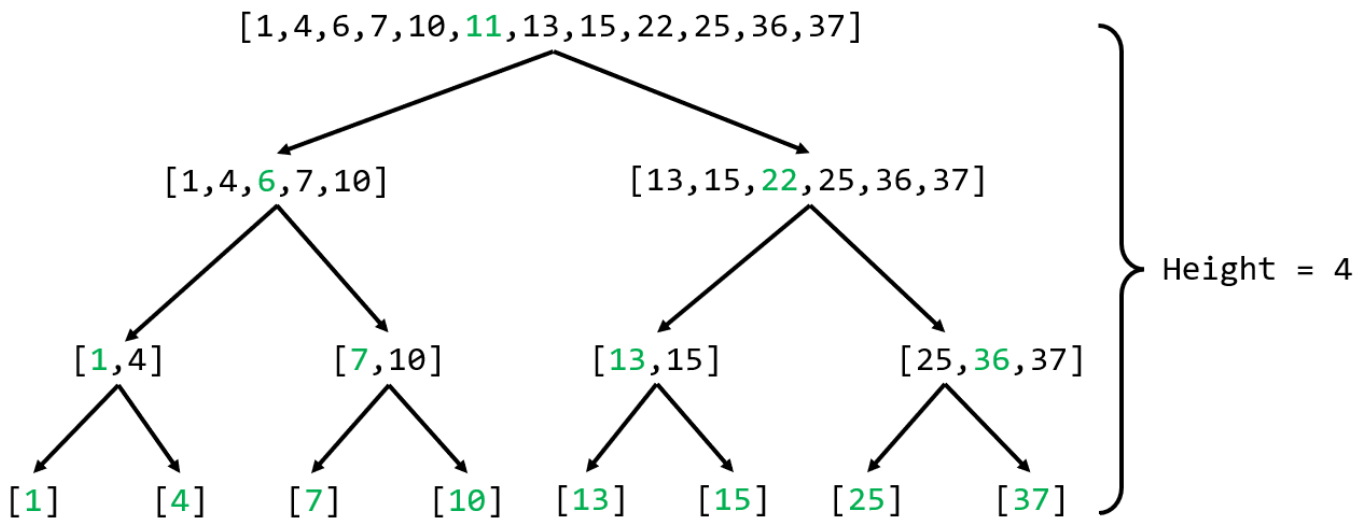
Binary Search Time Complexity

To determine the time complexity of the binary search algorithm, you must consider the **main operation**. The main operation here is the **comparison** between the midpoint item (of the section of the list that is being examined every time) and the item being searched for.

In the iterative version of the algorithm, the list is repeatedly halved until either the search item is found or the items is not found in the list.

- In the **worst case**, the main operation will be carried out $\log_2(n) + 1$ (where n is the number of items in the list). This is because if the item sought is found on the final comparison, or it is not found, then the maximum number of comparisons will be made. The **maximum number of comparisons** needed will be $\log_2(n) + 1$ (where n is the number of items in the list). In Big O notation, this time complexity is defined as $O(\log(n))$ because the cost of the final comparison is negligible when searching large data sets. This is known as **logarithmic complexity**. We can therefore say that at the **worst case** the time complexity of the algorithm is $O(\log(n))$.
- In the best case, the time complexity will be $O(1)$. This will be achieved if the item you are looking for is found at the first comparison.
- To calculate the average case time complexity requires a good understanding of statistics. The Big O notation of the complexity will still be expressed as $O(\log(n))$ as the constants become less significant as the size of the input grows.
- If you are asked only to state "time complexity", you should give the worst case which in $O\log(n))$.

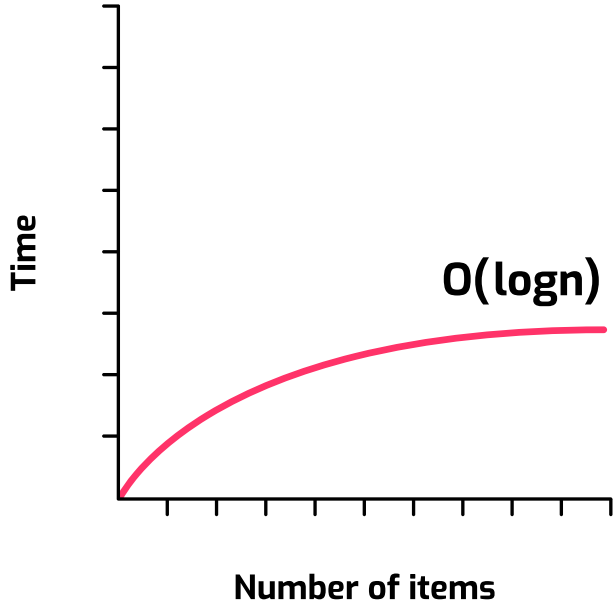
You can view this as a binary tree. You will notice that the height of the tree (number of levels) is $\log_2(n) + 1$.



Number of items $n = 12$
Height = $\lfloor \log(12) \rfloor + 1 = 4$

The recursive version of the algorithm has the same $O\log(n))$ time complexity. The list is repeatedly halved (approximately) and the search process continues until the search item is found or the list is exhausted.

However, the recursive version has higher space complexity, as recursion places more demand on the call stack.



Binary Search Space Complexity

Iterative Case

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 $O(1)$.

Recursive Case

We won't examine this in depth.

In the case of the recursive algorithm, we keep creating new variables for each stack frame that is created (each call to the function recursively), so we have to look at the call tree, this is the same as the way we looked at the tree for the time complexity above.

The space complexity is therefore $O(log(n))$

Summary

- Binary search (iterative)
  - Requires **ordered** data.
  - Worse-case time complexity - $O(log(n))$
  - Best-case time complexity - $O(1)$
  - Space complexity - $O(1)$
- Binary search (recursive)
  - Requires **ordered** data.
  - Worse-case time complexity - $O(log(n))$
  - Best-case time complexity - $O(1)$
  - Space complexity - $O(log(n))$

Acknowledgement

Searching Alogrithms - Isaac Computer Science.