Search algorithms

Lecturer: Arran Stewart

Outline

We discuss:

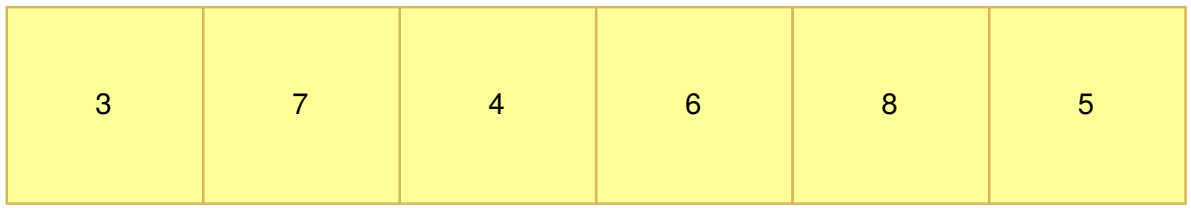
- What are some search algorithms?
- How can they be implemented?
- What is their complexity?

Search

- A very common use of computers is to look up data – for instance, in a library catalog, or some other database
- A very simple form of this which we will look at is searching through a collection (such as an array) for a particular item.

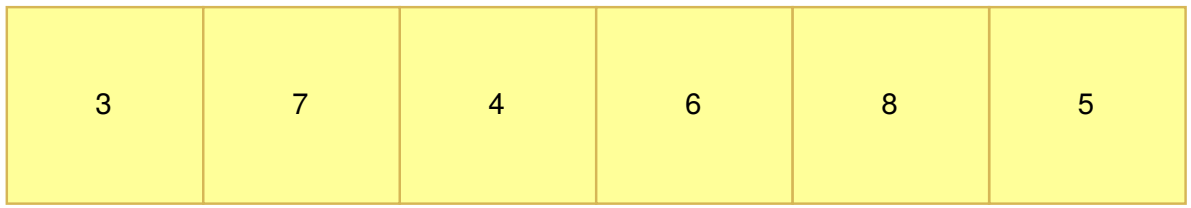
Sequential search

Suppose we have an array where the elements are in no particular order . . .



Sequential search

Suppose we have an array where the elements are in no particular order . . .



- In that case, if we want to find whether the array contains some particular number, we have no choice but to search it *sequentially* from beginning to end.
  - This is therefore called *sequential search*
- e.g. Suppose we want to know if the number 8 appears, and where – we must look through most of the array to find it.

Sequential search code

Java code for sequential search is available in the `SearchAlgorithms` class in the code samples. Here is a simplified version we will discuss:

```
 /*
  * @param a array to be searched
  * @param key item being searched for
  * @return index of key in a if key is found, and -1 otherwise
  */  public static int SequentialSearch(int[] a, int key) {
  for (int i = 0; i < a.length; i++) {
    if (a[i] == key) { // found it!
      return i;
    }
  }
  return -1; // if we get here, the item was not found
}
```

Sequential search

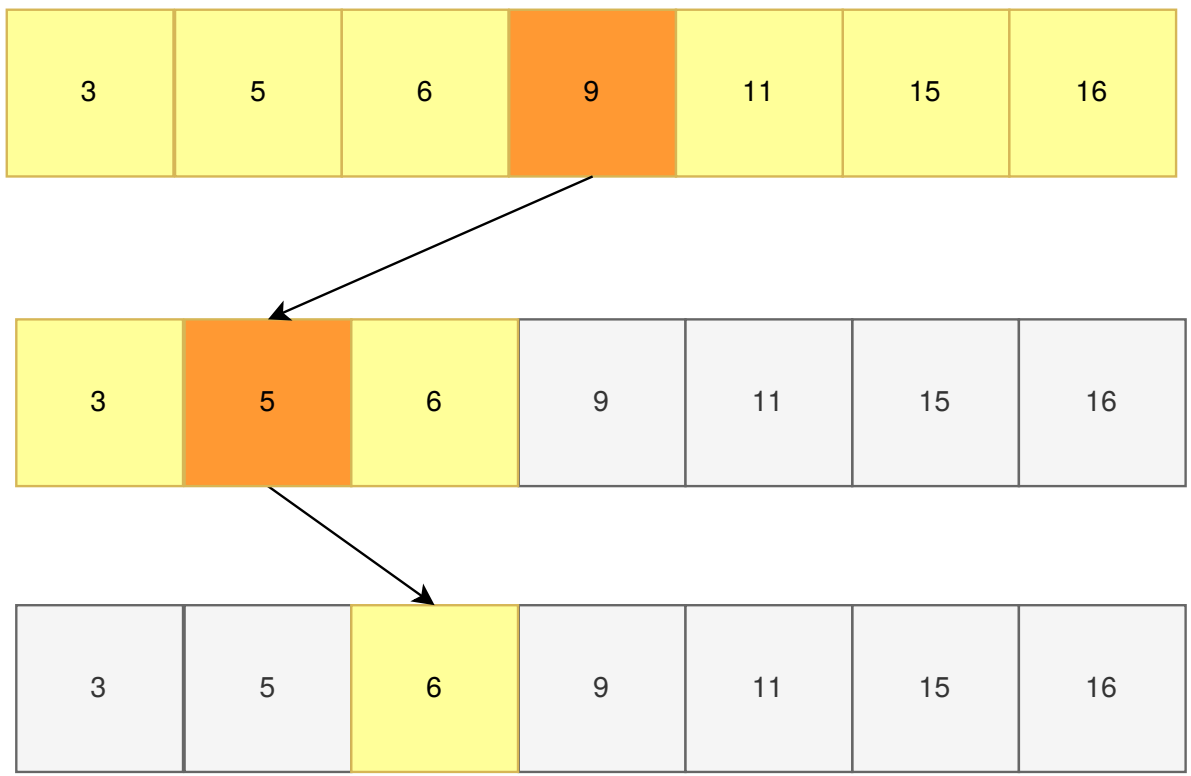
- In the *best* case, we may be lucky – perhaps the item we are looking for is the very first element in the array.
- But in the *worst* case, we may have to search the whole array before concluding that the element is not contained in it.
- The time for searching an array in the worst case is proportional to the size of the array;
  - Therefore the worst-case complexity of sequential search is $O(n)$.

Binary search

- But if the input array has been sorted, there is a better way of searching for an element.
- Instead of starting our search from the *end*, we start from the *middle*.
- Once we have examined the middle element –
  Since we know the array is sorted, we can work out whether (if the item we are after is in the array) it is on our left or our right.

Binary search example

- For instance, suppose we are searching the array below to see whether it contains the number 6.
  - The orange square shows the element we are "currently considering";
  - gray squares show elements we know do *not* contain the element we are after, and can therefore rule out.



Binary search code

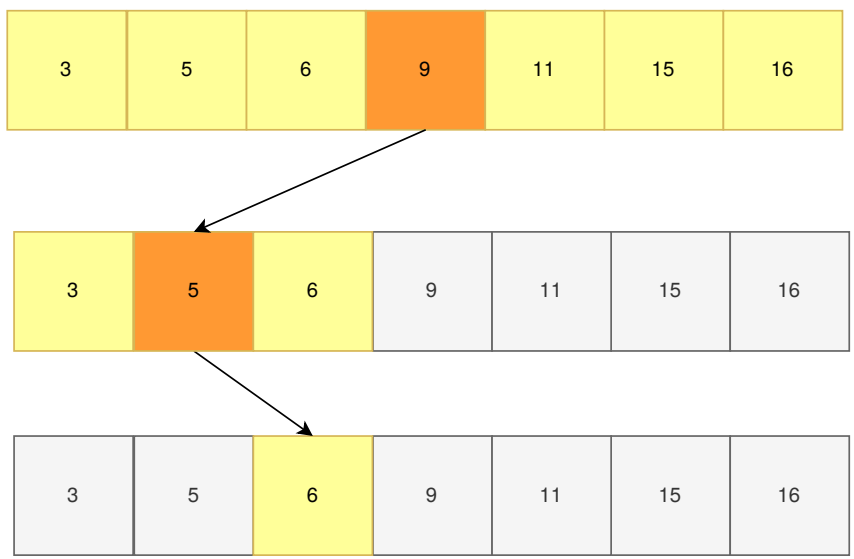
We will discuss the following code for binary search.

```
/**
 * @param a array to be searched, assumed to be sorted
 * @param key item being searched for
 * @return index of key in a if key is found, and -1 otherwise
 */
public static int BinarySearch(int[] a, int key) {
  int low = 0;
  int high = a.length - 1;
  int mid;

  while (low <= high) {
    mid = (low + high) / 2;
    if (a[mid] < key) { // continue to search lower part
      low = mid + 1;
    } else if (a[mid] > key) { // continue to search upper part
      high = mid - 1;
    } else { // we've found it
      return mid;
    }
  }
  // if we get here, the item was not found
  return -1;
}
```

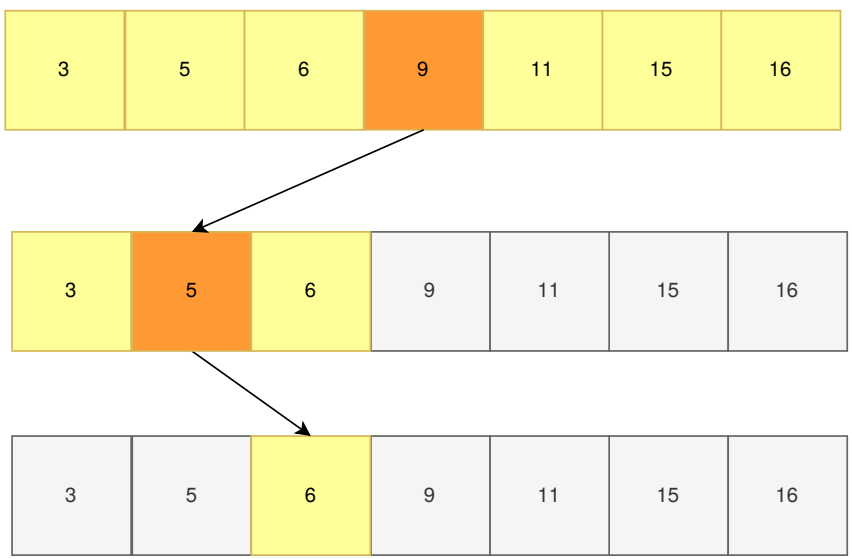
Binary search complexity

- What is the worst-case complexity of the binary search algorithm? One way to think about it is as follows.



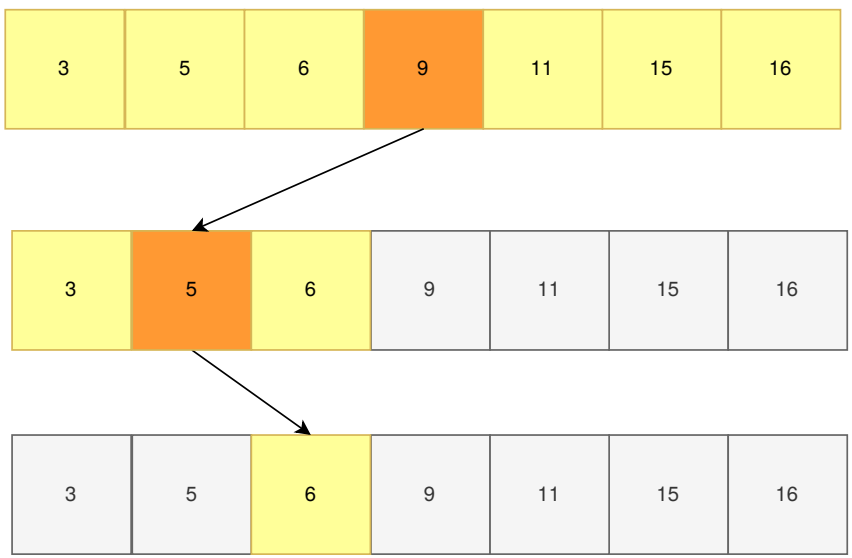
Binary search complexity

- What is the worst-case complexity of the binary search algorithm? One way to think about it is as follows.
- When we start to search an array of size n, we don't know at all where the element we are searching for could be.



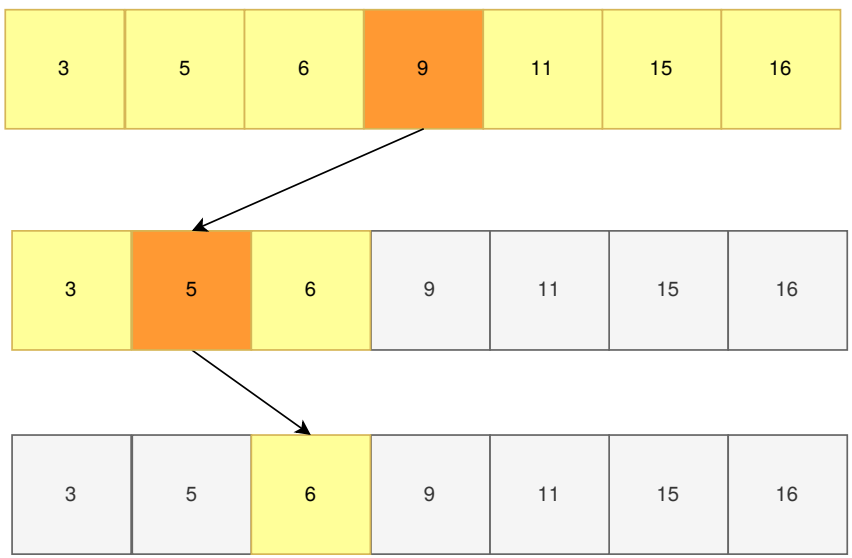
Binary search complexity

- What is the worst-case complexity of the binary search algorithm? One way to think about it is as follows.
- When we start to search an array of size n, we don't know at all where the element we are searching for could be.
- But after the first "step", we have reduced the size of the array portion we need to search – the new portion is half the old.



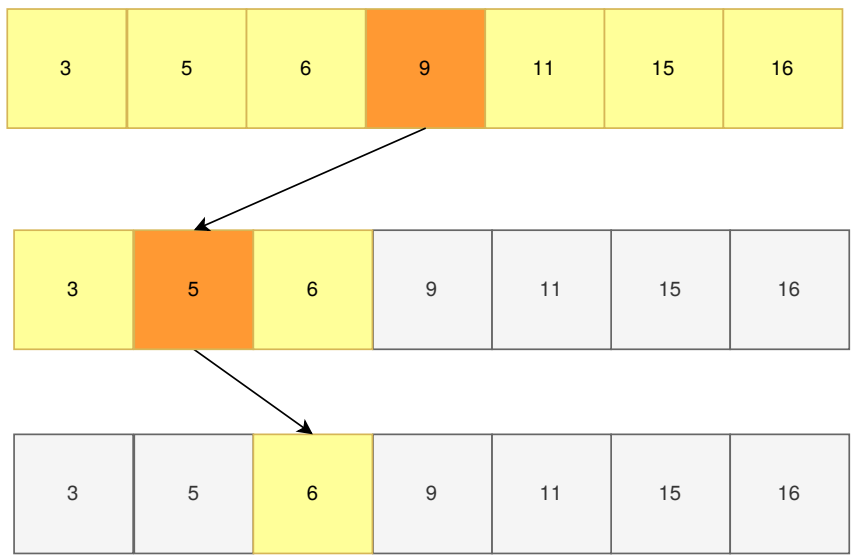
Binary search complexity

- What is the worst-case complexity of the binary search algorithm? One way to think about it is as follows.
- When we start to search an array of size n, we don't know at all where the element we are searching for could be.
- But after the first "step", we have reduced the size of the array portion we need to search – the new portion is half the old.
- And eventually, the size of the portion we need to search will be only 1 element in size.



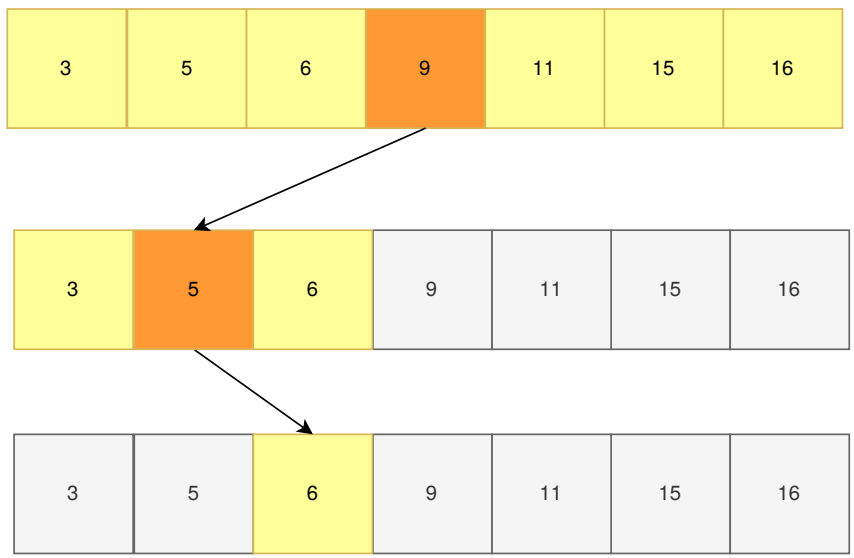
Binary search complexity

- Once the size of the portion we need to search is of size 1 (i.e. a single element) . . .



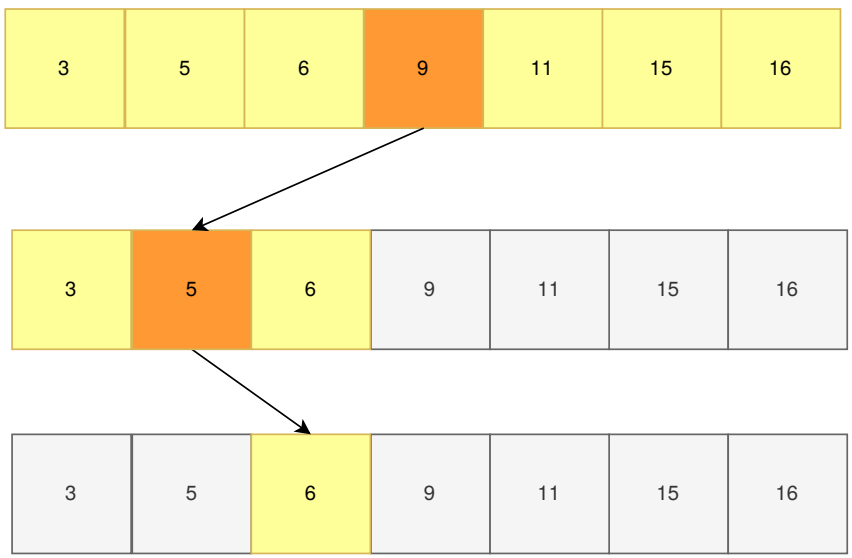
Binary search complexity

- Once the size of the portion we need to search is of size 1 (i.e. a single element) ...
- At that point, we know exactly where the element is (or, we know that the element is not to be found in the array)



Binary search complexity

- Once the size of the portion we need to search is of size 1 (i.e. a single element) . . .
- At that point, we know exactly where the element is (or, we know that the element is not to be found in the array)
- (Alternatively – we may hit the element earlier – but here we will consider the *worst case*.)



Binary search complexity

- So for instance, if our original array is of size 32, then the sizes of the portions we need to search are (in the worst case):

  32
  16
  8
  4
  2
  1

- We must perform 5 “halvings”;
- And the formula for the number of halvings we have to perform on a number before we get to 1 is $\log_2 n$

Binary search complexity

- So – for an array of size n, we must (in the worst case) perform $\log_2 n$ of these halvings
- So the worst-case complexity of binary search is $O(\log n)$.
  - (Why do we leave of the subscript 2? Because when dealing with logarithms, changing the base amounts to multiplication by some factor – and "big 'O' " complexity ignores multiplications by a constant factor.)

Recursive binary search

- The method we have seen is an *iterative* binary search – it contains a loop (a `while` loop) and does not call itself.
- We can also implement binary search using recursion.
- We will discuss the following code:

```
boolean binarySearch(int[] a, int lf, int rt, int key) {
  if (rt - lf == 1)
    return a[lf] == key || a[rt] == key;
  int mid = (lf+rt)/2;
  if (a[mid] > key)
    return binarySearch(a,lf,mid,key);
  else
    return binarySearch(a,mid,rt,key);
}
```

Recursive binary search

- Binary search can be expressed recursively in a very natural fashion, because we repeatedly perform the same operation of calculating the middle element and then searching in an array of half the size
- What is the "simplest case" in this situation?
  - The length of the array is the parameter that is reduced at each stage of binary search and so the base case is when the left and the right bounds of the array are adjacent
  - Once this happens, it is trivial to determine whether or not the element we are looking for is in the array or not.