Searching

- Determining whether or not a target is contained in a data structure (such as an array)

  or

- Looking up an item by a key (value that uniquely identifies something, like an ID)

- Very common problem in CS
- Data structures can get very large, so we are interested in algorithms that can search quickly

- Time complexity analysis
  - When analyzing time complexity we talk about the number of steps an algorithm takes, not the actual amount of time it takes
  - In particular we want to know how long the algorithm will take based on the size of the input (in this case, the size of an array)

- Searching for an element in an array of n elements

  - Worst case

    {1, 2, -10, 40, 183}

    - The target is not in the array
    - Every element must be checked before returning false
    - Takes n steps
    - The amount of time taken to search grows linearly with the size of the array
    - If n doubles, the number of steps doubles

- Best case
  - The target is the first element in the array
  - No elements beyond the first element need to be checked
  - The amount of time taken in the best case is constant
  - No matter the size of the array, the best case always takes the same time

- Binary Search

{-20, -10, 1, 2, 18, 40, 42}

  - Only works on a sorted array
  - Algorithm
    - If the array has 0 elements, return false
    - Compare the middle element to the target
      - If they are equal, return true
      - If the target is less than the middle element, repeat the search on the portion of the array to the left of the middle
      - If the target is greater than the middle, repeat the search on the portion of the array to the right of the middle

{-20, -10, 1, 2, 18, 40, 42}

Search for 42

- Middle is 2, less than 42
  - Repeat search on {18, 40, 42}
- Middle is 40, less than 42
  - Repeat search on {42}
- Middle is 42, return true

Search for -15

- middle is 2, greater than -15
  - repeat search on {-20, -10, 1}
- middle is -10, greater than -15
  - repeat search on {-20}
- middle is -20, less than -15
  - repeat the search on {}
- base case, return false

```
bool binary_search(int target, int *array, size_t size);
```

Say array is {1, 2, 5, 10, 30, 45, 46, 48}
0 1 2 3 4 5 6 7

Size is 8

middle index is size / 2

- with odd size gives the true middle
- with even size gives the leftmost element of the right half

Say we're searching for 6

middle index is 4 (value 30, so search to the left)

```
return binary_search(target, array, middle_index);
```

Search to the right of middle_index

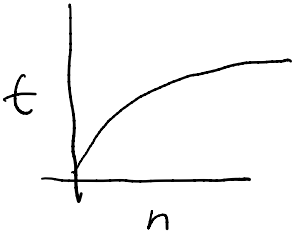
{ 1, 2, 5, 10, 30, 45, 46, 48 }
  0 1 2 3 4 5 6 7

return binary_search(target, array + middle_index + 1, size - middle_index - 1);

{ 45, 46, 48 }
  0   1   2

{ 1, 2, 5, 10, 30, 45, 46, 48, 49, 50, 51, 52, 53, 54, 55, 56 }
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Size	# of calls to binary search (worst case)
2^3 8	5
2^4 16	6
2^5 32	7
$2^{\log_2(n)}$ n	$\log_2(n) + 2$



logarithmic time complexity