Search Lesson Outline

1. Searching Lesson Outline
2. How to Find a Value in an Array?
3. Linear Search
4. Linear Search Code
5. Linear Search Example #1
6. Linear Search Example #2
7. Linear Search Example #3
8. Linear Search Example #4
9. Linear Search Example #5
10. Linear Search Example #6
11. How Long Does Linear Search Take?
12. Linear Search: Best Case
13. Linear Search: Worst Case
14. Linear Search: Average Case
15. Why Do We Care About Search Time?
16. "Big-O" Notation #1
17. "Big-O" Notation #2
18. "Big-O" Notation #4
19. "Big-O" Notation #5
20. "Big-O" Notation #6
21. Linear Search Code, Again
22. Linear Search is **O**(*n*) in the Average Case
23. A Better Search?
24. Faster Search Requires Sorted Data
25. Binary Search
26. Binary Search Code
27. Binary Search Code
28. Binary Search Example #1
29. Binary Search Example #2
30. Binary Search Example #3
31. Time Complexity of Binary Search #1
32. Time Complexity of Binary Search #2
33. Time Complexity of Binary Search #3
34. Need to Sort Array

How to Find a Value in an Array?

-23	97	18	21	5	-86	64	0	-37

Suppose you have a big array full of data, and you want to find a particular value.

How will you find that value?

Linear Search

-23	97	18	21	5	-86	64	0	-37

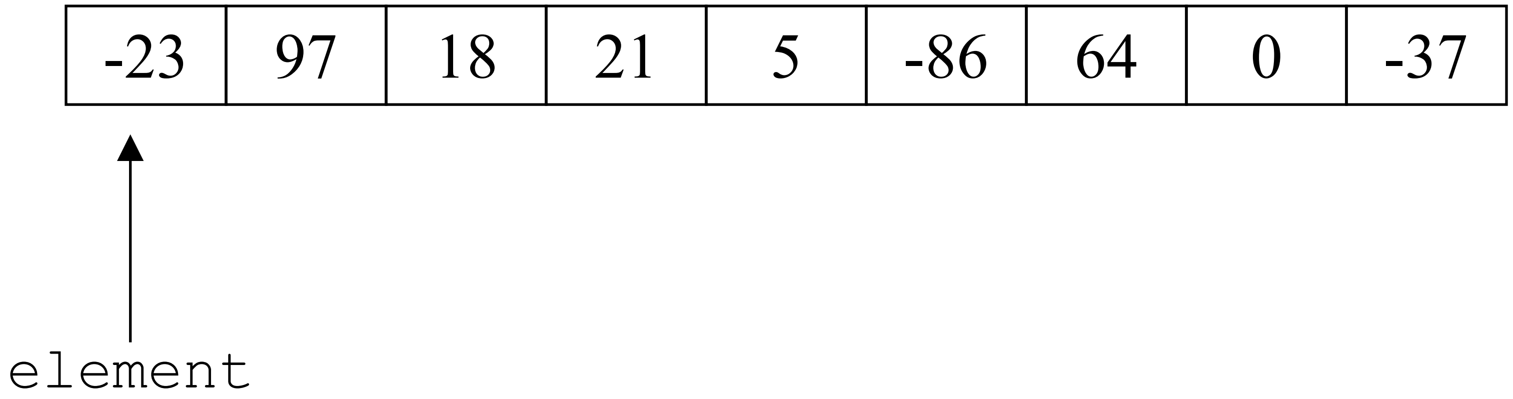
Linear search means looking at each element of the array, in turn, until you find the target value.

Linear Search Code

-23	97	18	21	5	-86	64	0	-37

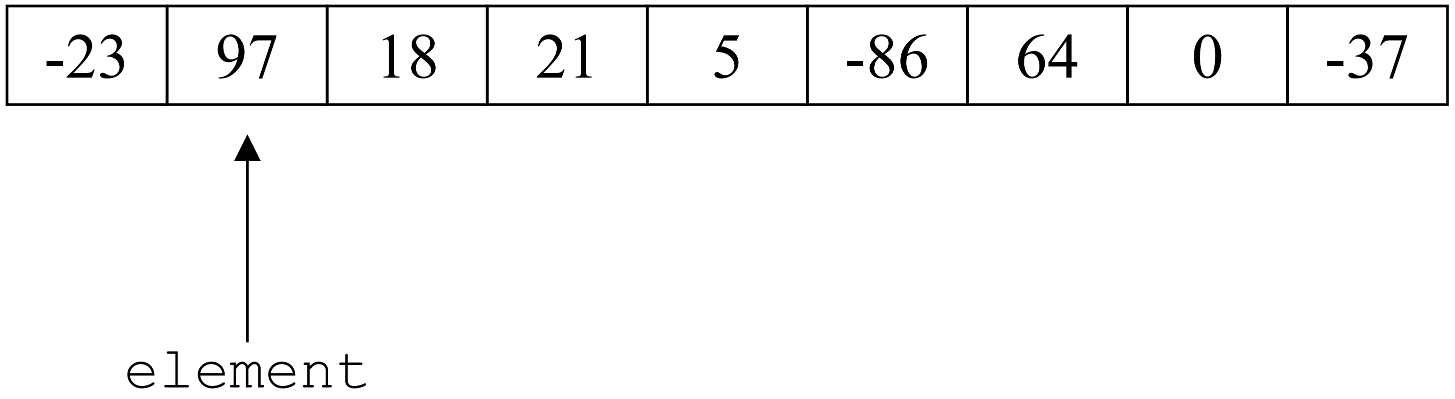
```
int linear_search (float* array,
                   int number_of_elements,
                   float target_value)
{ /* linear_search */
    const int first_element       = 0;
    const int nonexistent_element = first_element - 1;
    int element;

    /* Idiotproofing belongs here. */
    for (element = first_element;
         element < number_of_elements; element++) {
        if (array[element] == target_value) {
            return element;
        } /* if (array[element] == ...) */
    } /* for element */
    return nonexistent_element;
} /* linear_search */
```

Linear Search Example #1



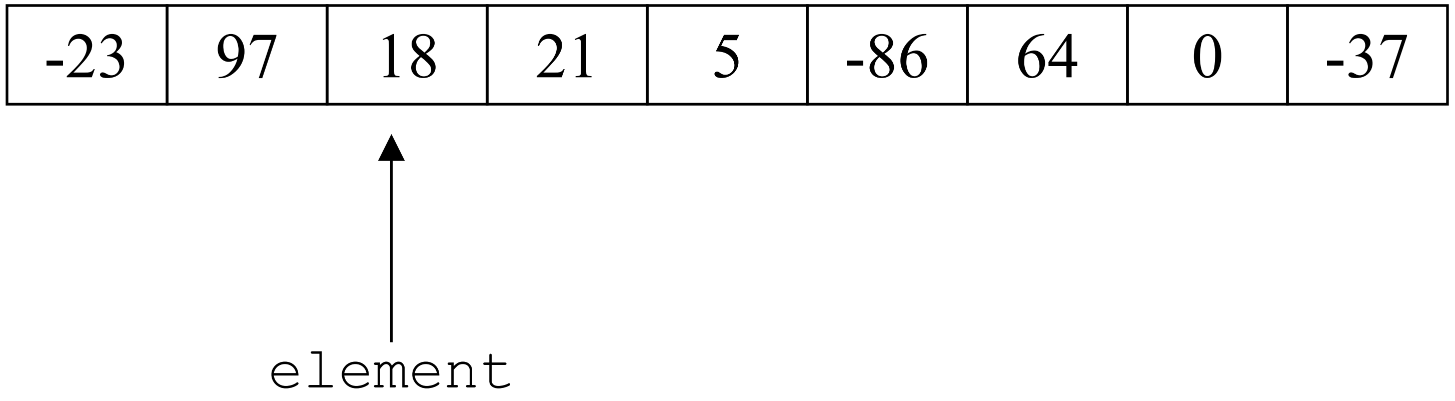
Searching for -86.

Linear Search Example #2



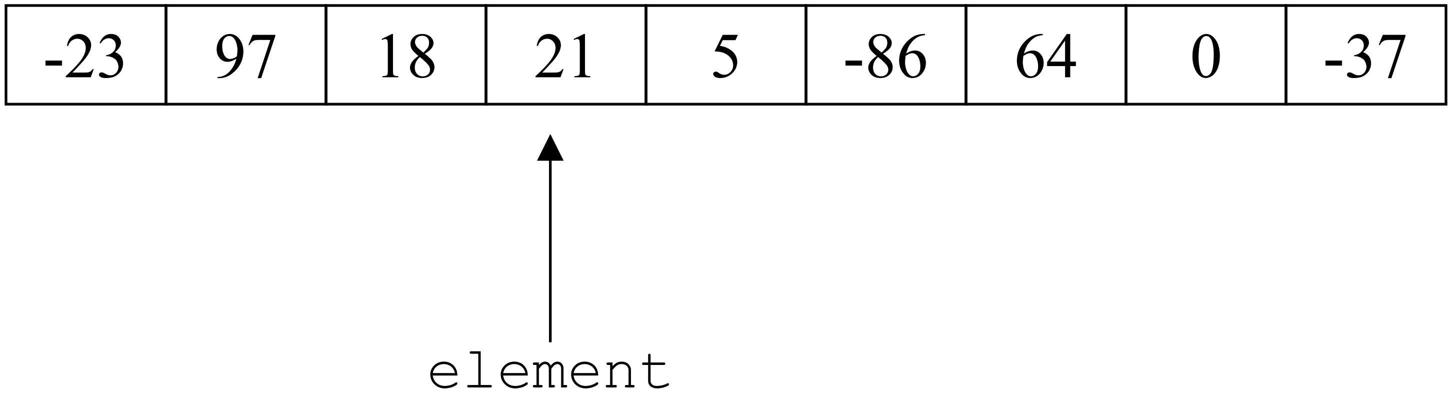
Searching for -86.

Linear Search Example #3



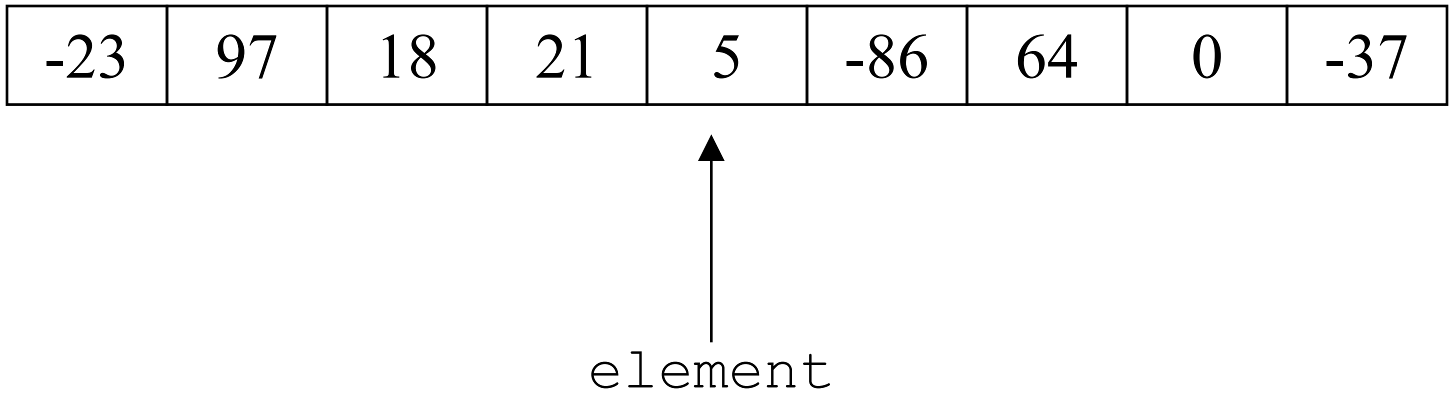
Searching for -86.

Linear Search Example #4



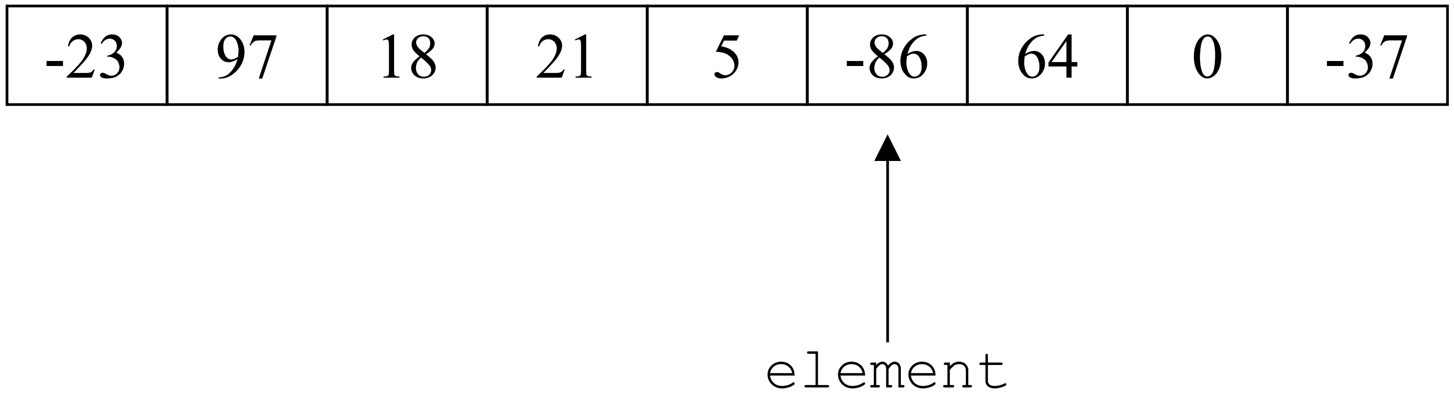
Searching for -86.

Linear Search Example #5



Searching for -86.

Linear Search Example #6



Searching for -86: **FOUND!**

How Long Does Linear Search Take?

Okay, great, now we know how to search.

But how long will our search take?

That's really three separate questions:

- How long will it take in the **best case**?
- How long will it take in the **worst case**?
- How long will it take in the **average case**?

Linear Search: Best Case

How long will our search take?

In the **best case**, the target value is in the first element of the array.

So the search takes some tiny, and constant, amount of time.

Computer scientists denote this **O(1)** – which will be explained later.

In real life, we don't care about the best case, because it so rarely actually happens.

Linear Search: Worst Case

How long will our search take?

In the **worst case**, the target value is in the last element of the array.

So the search takes an amount of time proportional to the length of the array.

Computer scientists denote this **O(*n*)** – which will be explained later.

Linear Search: Average Case

How long will our search take?

In the **average case**, the target value is somewhere in the array.

In fact, since the target value can be anywhere in the array, any element of the array is equally likely.

So on average, the target value will be in the middle of the array.

So the search takes an amount of time proportional to half the length of the array – which is proportional to the length of the array – **O(*n*)** again!

Why Do We Care About Search Time?

We know that time is money.

So finding the fastest way to search (or any task) is good, because then we'll save time, which saves money.

“Big-O” Notation #1

Suppose we can describe the amount of time it takes to do a task in terms of the number of pieces of data in the task.

For example, suppose that our search algorithm takes $3n + 12$ time units to execute, where n is the array length.

Well, as n becomes very big – a million, a billion, etc – then we stop caring about the 12,
because the 12 is basically zero compared to the $3n$ term.

"Big-O" Notation #2

Suppose we can describe the amount of time it takes to do a task in terms of the number of pieces of data in the task.

For example, suppose that our search algorithm takes $3n + 12$ time units to execute, where n is the array length.

No matter the size of n, the 3 in $3n$ isn't all that interesting, because running on a different kind of computer changes the actual time cost of a "time unit."

“Big-O” Notation #4

Suppose we can describe the amount of time it takes to do a task in terms of the number of pieces of data in the task.

For example, suppose that our search algorithm takes $3n + 12$ time units to execute, where n is the array length.

So, we really only care about the n in $3n + 12$.

"Big-O" Notation #5

Now, suppose we have an algorithm that, for n pieces of data, takes $0.03n^2 + 12n + 937.88$ time units.

Again, we don't care about the constants.

And, as n becomes big – a million, a billion, etc – we no longer care about the n term ($12n$), because the n term grows far far slower than the n^2 term.

Nor do we care about the constant term (937.88), which doesn't grow at all.

That is, as n becomes big, the smaller terms are so tiny as to be pretty much zero.

"Big-O" Notation #6

In the general case, suppose an algorithm on n pieces of data takes:

$$c_k n^k + c_{k-1} n^{k-1} + c_{k-2} n^{k-2} + \ldots + c_1 n + c_0$$

for constants c_0, c_1, etc.

Keep in mind the principles that we've already seen:

- We don't care about the constants c_i.
- We don't care about the terms smaller than n^k.

We really only care about n^k.

So we say that this algorithm has ***time complexity*** "of order n^k."
We denote this $\mathbf{O}(n^k)$.
We pronounce it "big-O of n to the k."

Linear Search Code, Again

-23	97	18	21	5	-86	64	0	-37

```
int linear_search (float* array,
                   int number_of_elements,
                   float target_value)
{ /* linear_search */
    const int first_element       = 0;
    const int nonexistent_element = first_element - 1;
    int element;

    /* Idiotproofing belongs here. */
    for (element = first_element;
         element < number_of_elements; element++) {
        if (array[element] == target_value) {
            return element;
        } /* if (array[element] == ...) */
    } /* for element */
    return nonexistent_element;
} /* linear_search */
```

Linear Search is **O**(n) in the Average Case

Recapping, linear search is **O**(n) in the average case (and in the worst case, but with different constants).

But what if we expect to do lots of searches through our dataset of length n?

What if we expect to do n searches on our n data?

Well, the time complexity will be $n \cdot$ **O**(n), which is to say **O**(n^2).

You can imagine that, when n is big – a million, a billion, etc – this is terribly inefficient.

Can we do better?

A Better Search?

Consider how you search for someone in the phone book – say, Henry Neeman.

You start with the first letter of their last name, N.

You guess roughly where N would be in the phonebook.

You open to that page.

If you're wrong, you move either forward or backward in the book – that is, if you actually opened to J, you move forward, but if you opened to T, you move backward.

You keep repeating this action until you find Neeman.

Faster Search Requires Sorted Data

Why not use linear search?

Linear search means start at the beginning, and look at every piece of data, until you find your target.

And you have to do this for each and every such search.

This is much much slower than
the way you search a phonebook in real life.

Why?

The reason you can do the phonebook search so quickly is because the names in the phonebook are **<u>sorted</u>** – specifically, they're in alphabetical order by last name, then by first name.

Binary Search

The general term for a smart search through **sorted** data is a ***binary search***.

1. The initial search region is the whole array.
2. Look at the data value in the middle of the search region.
3. If you’ve found your target, stop.
4. If your target is less than the middle data value, the new search region is the lower half of the data.
5. If your target is greater than the middle data value, the new search region is the higher half of the data.
6. Continue from Step 2.

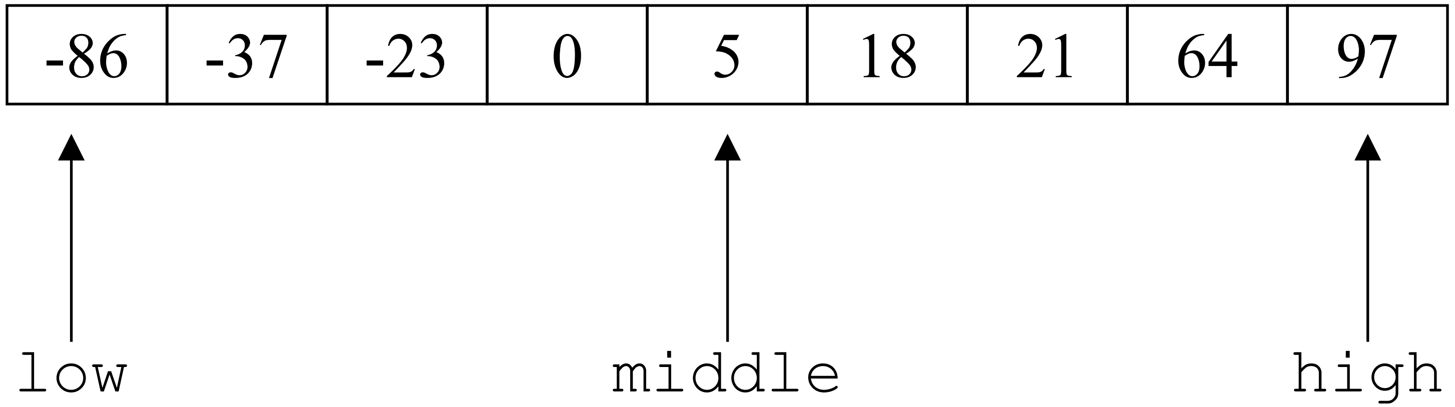
Binary Search Code

```
int binary_search (float* array, int number_of_elements,
                   float target_value)
{ /* binary_search */
    const int first_element         = 0;
    const int nonexistent_element = first_element - 1;
    int low_element, middle_element, high_element;

    /* Idiotproofing goes here. */
    /* Start with the entire array as the search region. */
    low_element  = first_element;
    high_element = number_of_elements - 1;
```

Binary Search Code

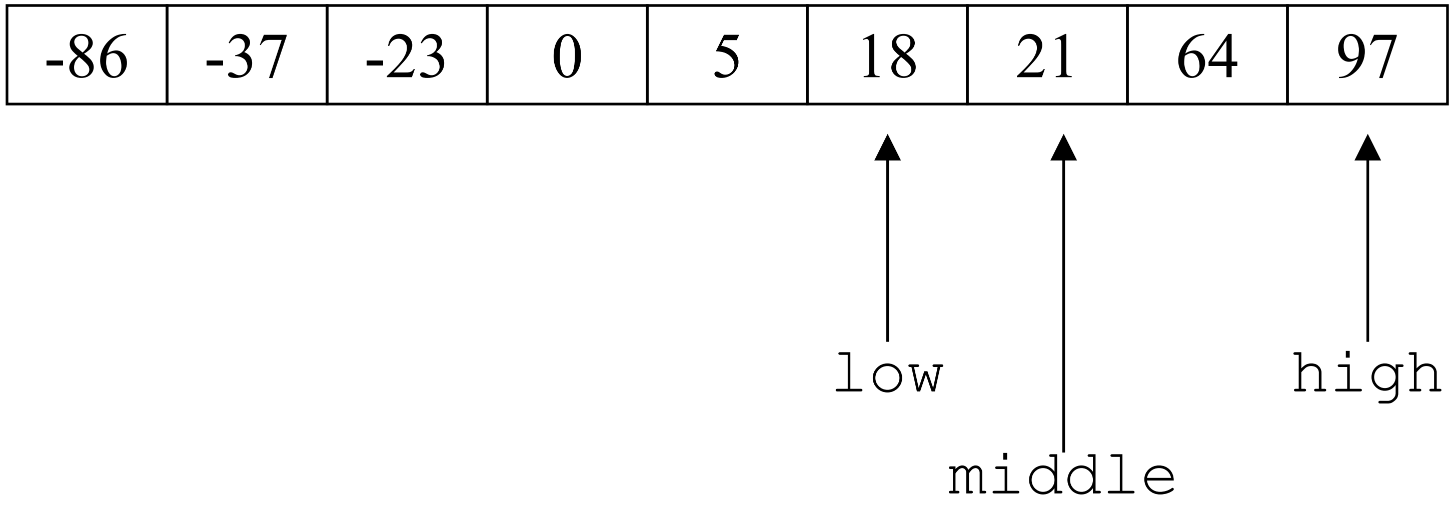
```
    while ((low_element  > first_element)      &&
           (high_element < number_of_elements) &&
           (low_element  < high_element)) {
        /* Examine the middle of the current search region. */
        middle_element = (low_element + high_element) / 2;
        /* What should we search next? */
        if (array[middle_element] < target_value) {
            /* Reduce the search region to the lower half. */
            high_element = middle_element - 1;
        } /* if (array[middle_element] < target_value) */
        else if (array[middle_element] > target_value) {
            /* Reduce the search region to the higher half. */
            low_element = middle_element + 1;
        } /* if (array[middle_element] > target_value) */
        else {
            /* Target has been found, so stop searching. */
            low_element  = middle_element;
            high_element = middle_element;
        } /* if (array[middle_element] > ...)...else */
    } /* while (low_element < high_element) */
    if (high_element == low_element) {
        return middle_element;
    } /* if (high_element == low_element) */
    return nonexistent_element;
} /* binary_search */
```

Binary Search Example #1



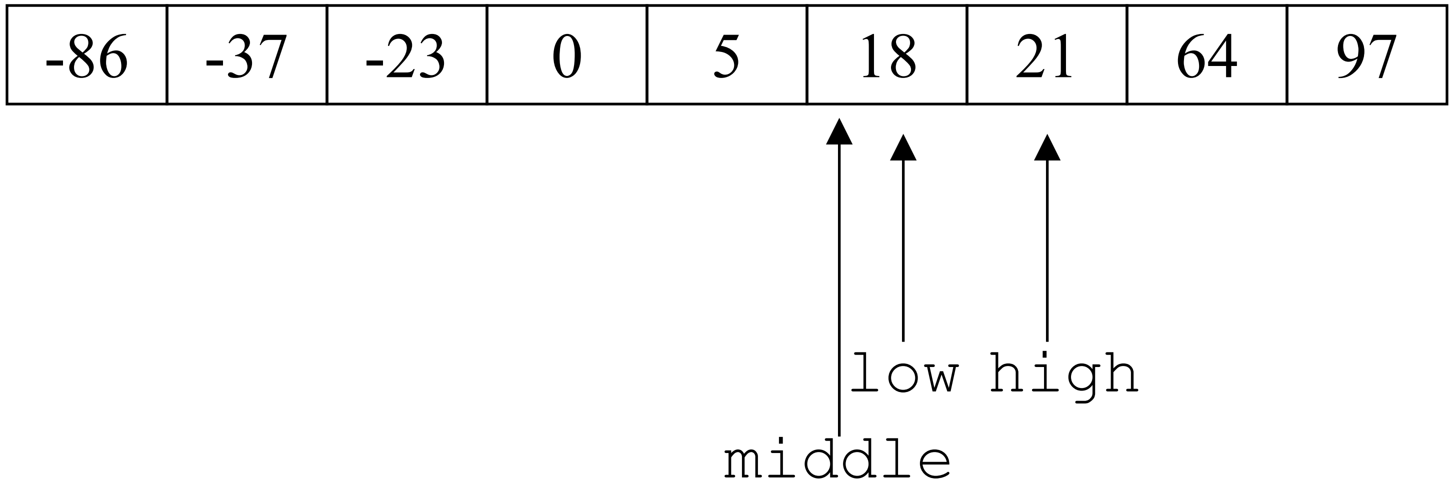
Searching for 18.

Binary Search Example #2



Searching for 18.

Binary Search Example #3



Searching for 18: **FOUND!**

Time Complexity of Binary Search #1

How fast is binary search?

Think about how it operates: after you examine a value, you cut the search region in half.

So, the first iteration of the loop, your search region is the whole array.

The second iteration, it's half the array.

The third iteration, it's a quarter of the array.

...

The kth iteration, it's $(1/2^{k-1})$ of the array.

Time Complexity of Binary Search #2

How fast is binary search?

For the *k*th iteration of the binary search loop, the search region is $(1/2^{k-1})$ of the array.

What's the maximum number of loop iterations?

$$\lceil \log_2 n \rceil$$

That is, we can't cut the search region in half more than that many times.

So, the time complexity of binary search is $\mathbf{O}(\log_2 n)$.

Time Complexity of Binary Search #3

How fast is binary search?

We said that the time complexity of binary search is **O**($\log_2 n$).

But, **O**($\log_2 n$) is exactly the same as **O**($\log_{10} n$)
is exactly the same as **O**($\ln n$)
is exactly the same as **O**($\log_b n$) for any base b.

Why?

Well, we know from math class that

$$\log_a x \equiv \log_b x \,/\, \log_b a$$

So the relationship between logs with different bases is simply a constant:

$$1 \,/\, \log_b a$$

Therefore, **O**($\log n$) is the same as **O**($\log_b n$) for any base b –
we don't care about the base,
because we don't care about the constant.

Need to Sort Array

Binary search only works if the array is already sorted.

It turns out that sorting is a huge issue in computing.