Algorithm Complexity I

Michael C. Hackett

Computer Science Department

Community
College
of Philadelphia

Lecture Topics

- Search Algorithms
  - Linear Search
  - Binary Search
- Algorithm Analysis
  - Time Complexity
    - Constant Time
    - Linear Time
    - Logarithmic Time
  - Space Complexity
    - Constant Space
    - Linear Space

- Asymptotic Notations
  - O-Notation
  - Ω-Notation
  - Θ-Notation
- Sorting Algorithms
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Analysis of these three algorithms
    - Polynomial Time
  - Sorting + Searching

Linear Search

- A **linear** (or **sequential**) **search** begins searching at the beginning of an array and continues until the item is found.

- Order of the elements (alphabetical, numerical, etc.) does not effect the searching process.

- Search algorithms will usually return:
  - The index where the data was found.
  - True or false (item was found vs. item was not found)

Linear Search (Pseudocode)

For *i* in indexes 0 through length-1 of array *a*:

If the element *a*[*i*] is what you are seeking:

Return *i*

Else, continue and check the next element

Linear Search (C++ Function)

```cpp
int linearSearch(int a[], int length, int searchValue) {
    for(int i = 0; i < length; i++) {
        if(a[i] == searchValue) {
            return i;
        }
    }
    return -1;
}
```

Indicates the value was not found.
The return statement in the loop would never return -1

Linear Search

- Order of the elements (alphabetical, numerical, etc.) does not effect searching.

- Best case scenario: The information sought is the first element.

- Worst case scenario: The information sought is the last element.

Binary Search

- Takes a "divide and conquer" approach.

- Begins searching in the middle of the array.
  - If the middle element is not what we are looking for, we then split the array in half:
    - If the value sought is greater* than the middle element, we will then check the middle element of the second half of the array.
    - If the value sought is less* than the middle element, we will then check the middle element of the first half of the array.
  - The process begins again with the new half.

*- The array must be in some order! (Alphabetically, numerically, etc.)
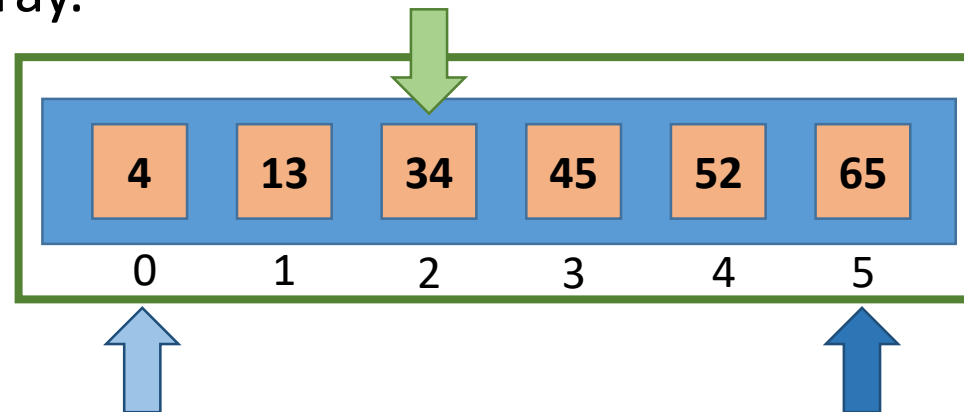
Binary Search (Searching for 45)

- In the first step of the algorithm:
  - The lower boundary is index 0.
  - The upper boundary is the last index.
  - To find the middle index: add the lower boundary index and the upper boundary index then divide by 2: (0 + 5)/2 = 2.5 -> 2

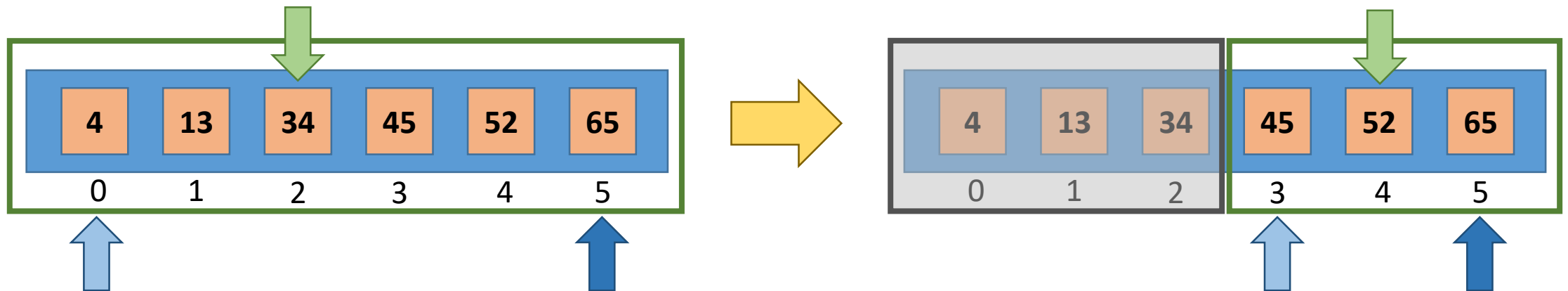| 4	13	34	45	52	65
0	1	2	3	4	5
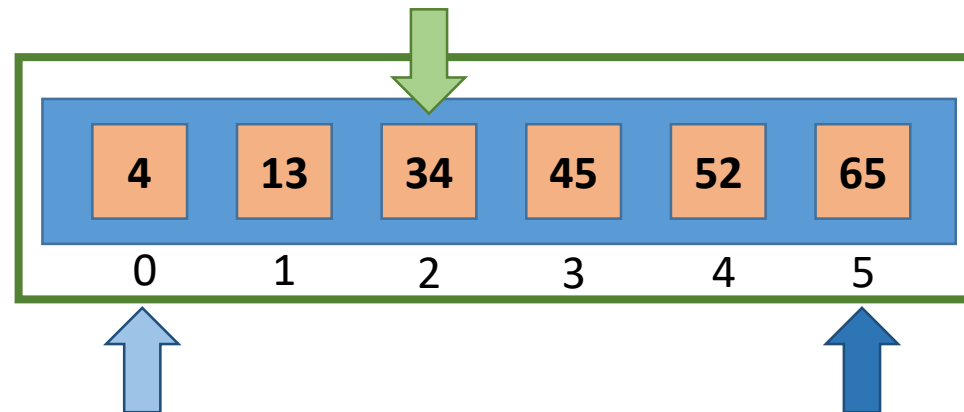
Binary Search (Searching for 45)

- Next, we do one of three things:
  - If the value we are seeking (45) is at this middle index, we are done searching.
  - **If the value we are seeking is greater than this value, then we will search the upper half of this array.**
  - If the value we are seeking is less than this value, then we will search the lower half of this array.
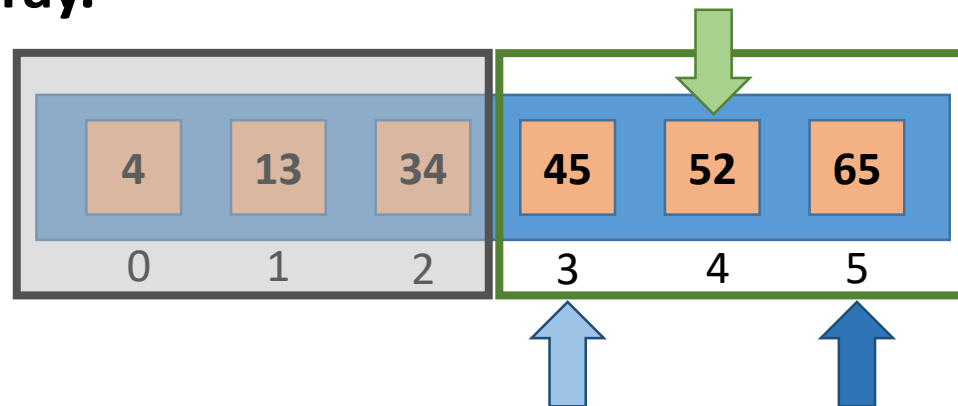
Binary Search (Searching for 45)

- We must now calculate the boundaries and middle index of the half we will search next.
  - Lower Boundary: Middle Index + 1 -> 2 + 1 = 3
  - Upper Boundary: Does not change.
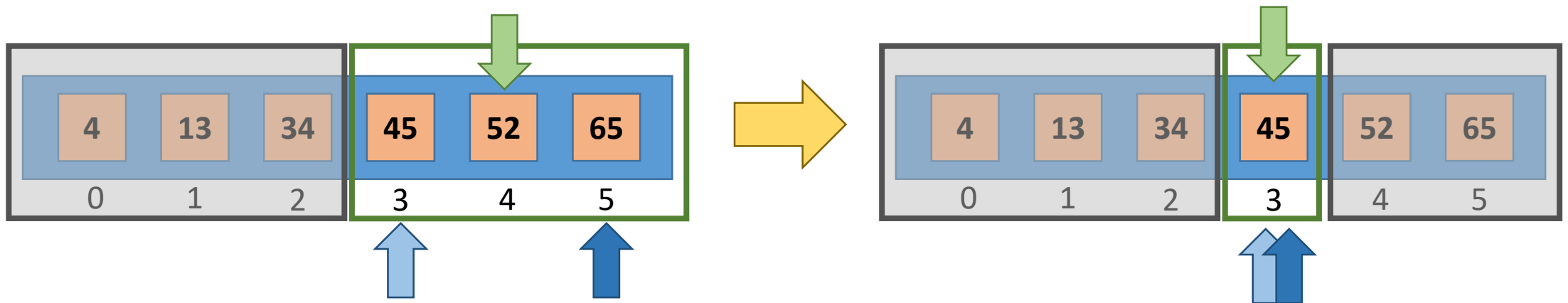 - Middle Index: (Lower + Upper) / 2 -> (3 + 5) /2 = 4

Binary Search (Searching for 45)

- We start the process over:
  - If the value we are seeking (45) is at this middle index, we are done searching.
  - If the value we are seeking is greater than this value, then we will search the upper half of this array.
  - **If the value we are seeking is less than this value, then we will search the lower half of this array.**
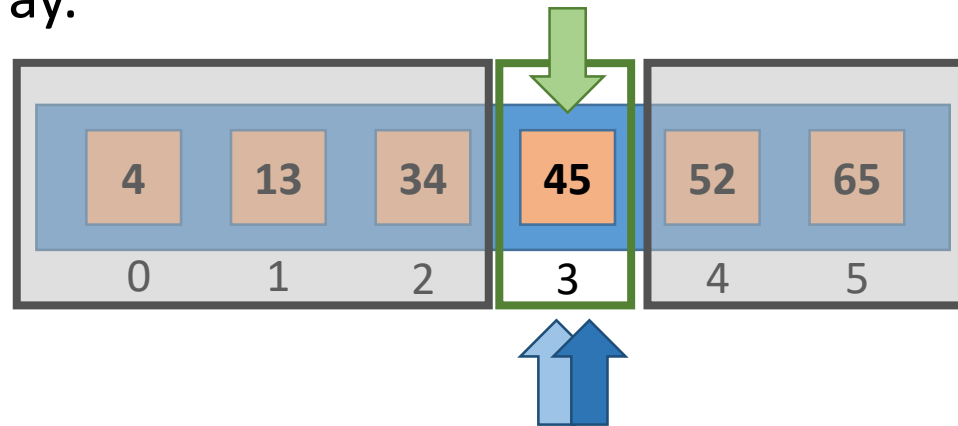
Binary Search (Searching for 45)

- We must now calculate the boundaries and middle index of the half we will search next.
  - Lower Boundary: Does not change.
  - Upper Boundary: Middle Index - 1 -> 4 - 1 = 3
 - Middle Index: (Lower + Upper) / 2 -> (3 + 3) /2 = 3

Binary Search (Searching for 45)

- We start the process over:
  - **If the value we are seeking (45) is at this index, we are done searching.**
  - If the value we are seeking is greater than this value, then we will search the upper half of this array.
  - If the value we are seeking is less than this value, then we will search the lower half of this array.
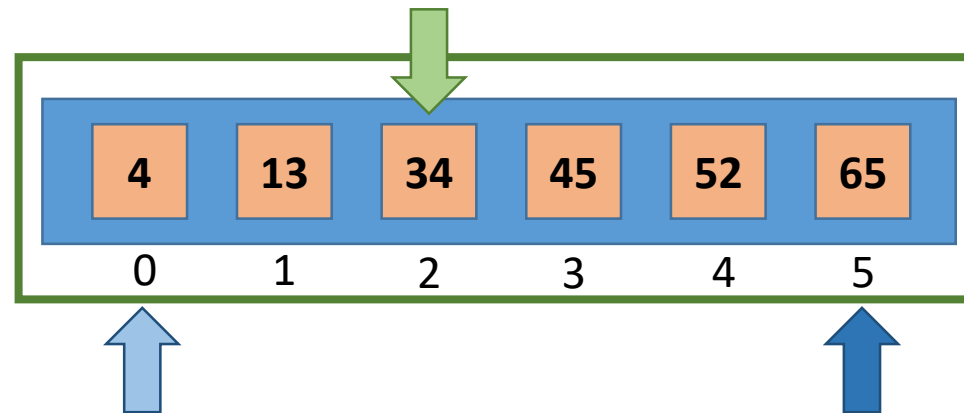
Binary Search

- What happens when the value we are looking for isn't in the array?
  - How does the algorithm know when to stop halving the array?
  - **When the upper boundary index is less than the lower boundary index.**

| 4	13	34	45	52	65
0	1	2	3	4	5
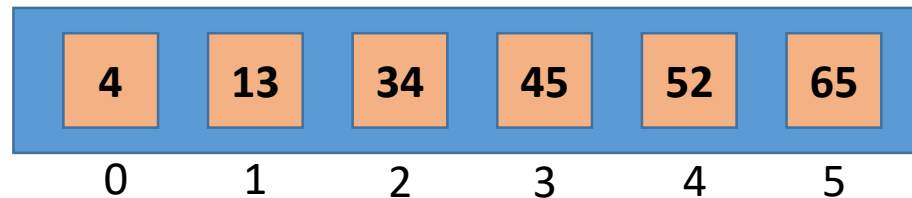
Binary Search (Searching for 12)

- The lower boundary is index 0.

- The upper boundary is the last index.

- To find the middle index: add the lower boundary index and the upper boundary index then divide by 2: (0 + 5)/2 = 2.5 -> 2

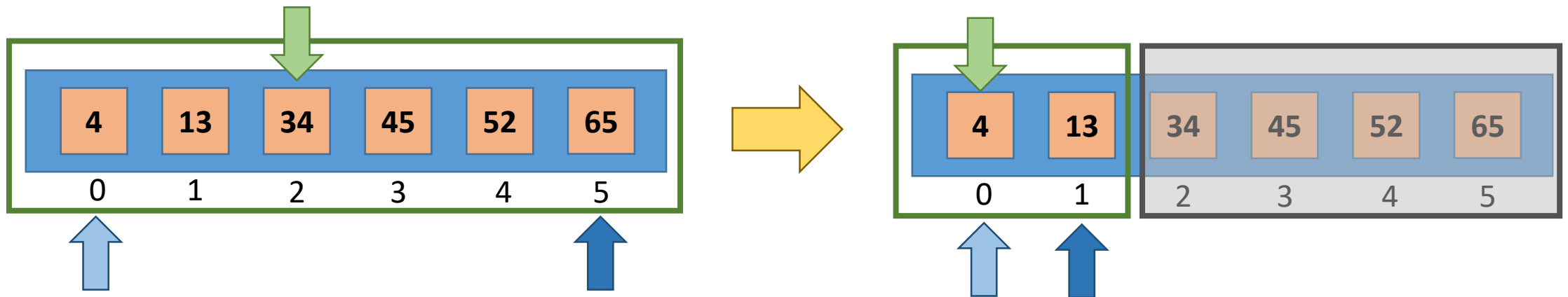| 4	13	34	45	52	65
0	1	2	3	4	5

Binary Search (Searching for 12)

- Next, we do one of three things:
  - If the value we are seeking (12) is at this middle index, we are done searching.
  - If the value we are seeking is greater than this value, then we will search the upper half of this array.
  - **If the value we are seeking is less than this value, then we will search the lower half of this array.**
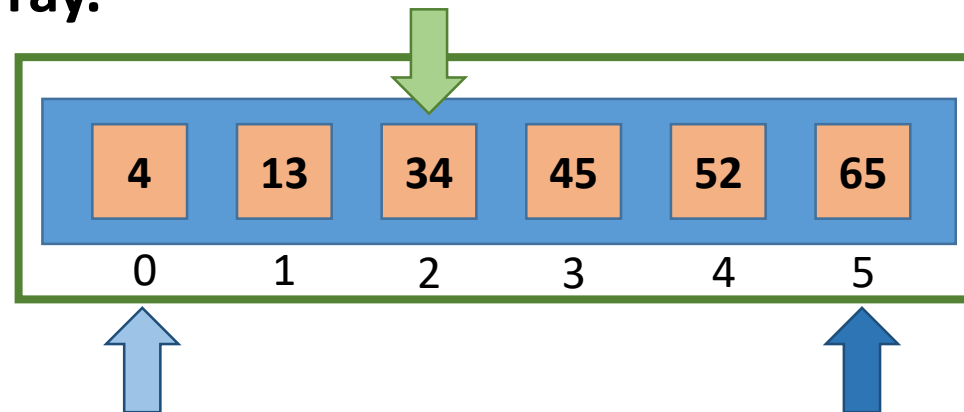
Binary Search (Searching for 12)

- We must now calculate the boundaries and middle index of the half we will search next.
    - Lower Boundary: Does not change.
    - Upper Boundary: Middle Index – 1 -> 2 – 1 = 1
    - Middle Index: (Lower + Upper) / 2 -> (0 + 1) /2 = 0.5 -> 0

Binary Search (Searching for 12)

- We start the process over:
  - If the value we are seeking (12) is at this index, we are done searching.
  - **If the value we are seeking is greater than this value, then we will search the upper half of this array.**
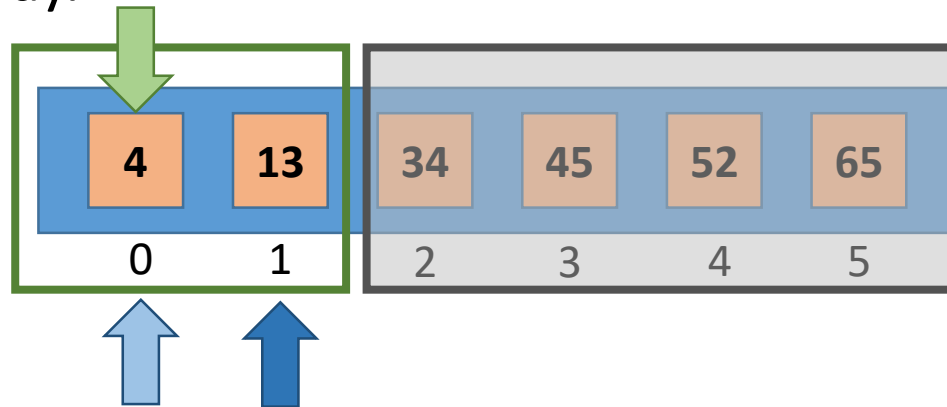 - If the value we are seeking is less than this value, then we will search the lower half of this array.

Binary Search (Searching for 12)

- We must now calculate the boundaries and middle index of the half we will search next.
  - Lower Boundary: Middle Index + 1 -> 0 + 1 = 1
  - Upper Boundary: Does not change.
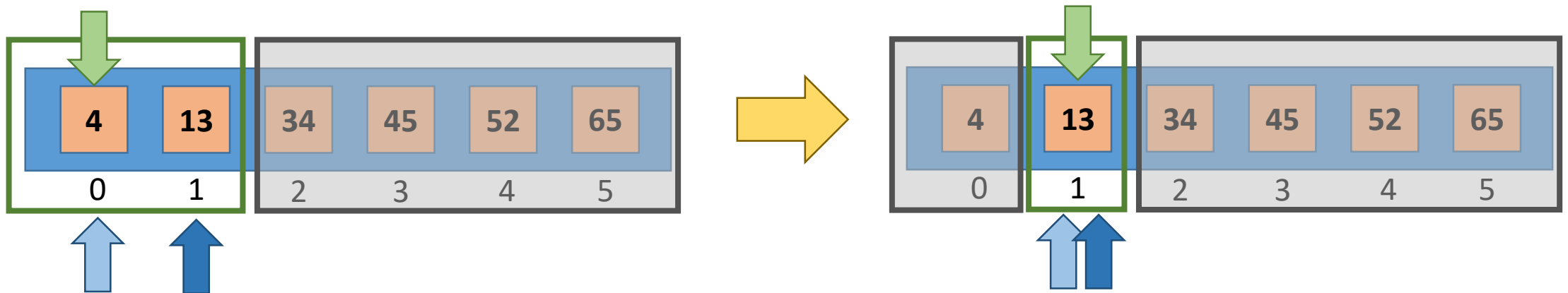 - Middle Index: (Lower + Upper) / 2 -> (1 + 1) /2 = 1

Binary Search (Searching for 12)

- We start the process over:
  - If the value we are seeking (12) is at this index, we are done searching.
  - If the value we are seeking is greater than this value, then we will search the upper half of this array.
  - **If the value we are seeking is less than this value, then we will search the lower half of this array.**
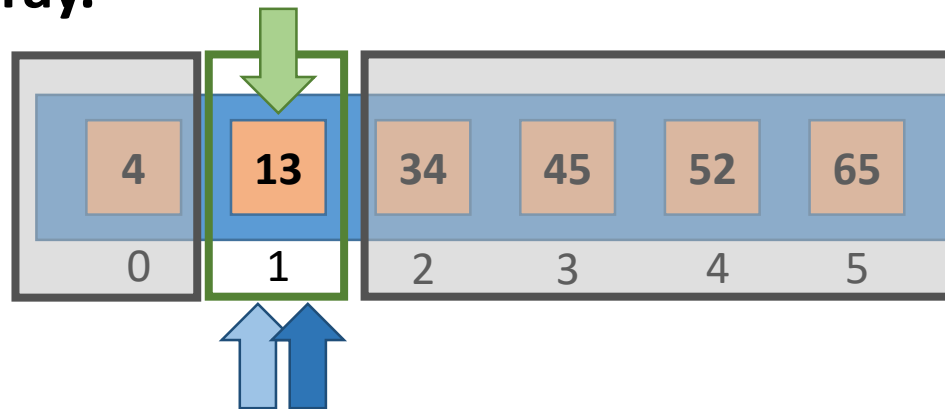
Binary Search (Searching for 12)

- We must now calculate the boundaries and middle index of the half we will search next.
  - Lower Boundary: Does not change.
  - Upper Boundary: Middle Index – 1 -> 1 – 1 = 0
  - Middle Index: (Lower + Upper) / 2 -> (1 + 0) /2 = 0.5 -> 0
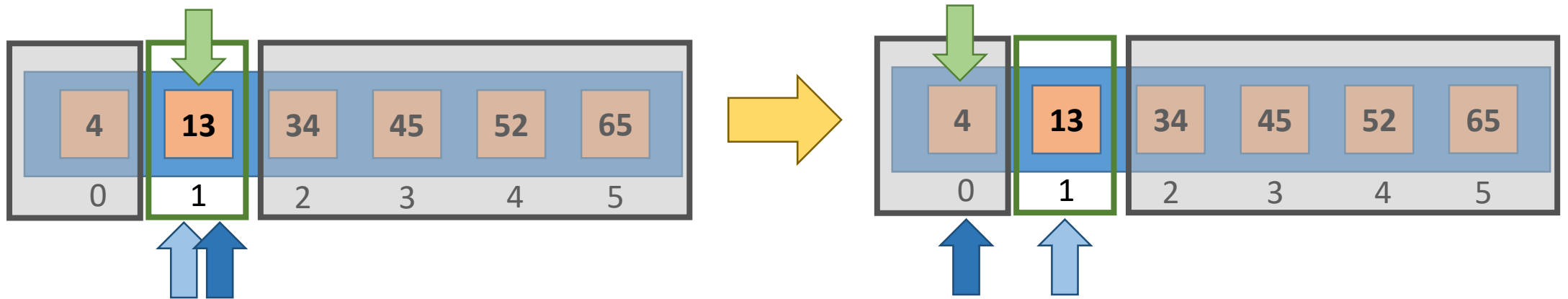
Binary Search (Searching for 12)

- When the upper boundary is less than the lower boundary, the algorithm will "give up."
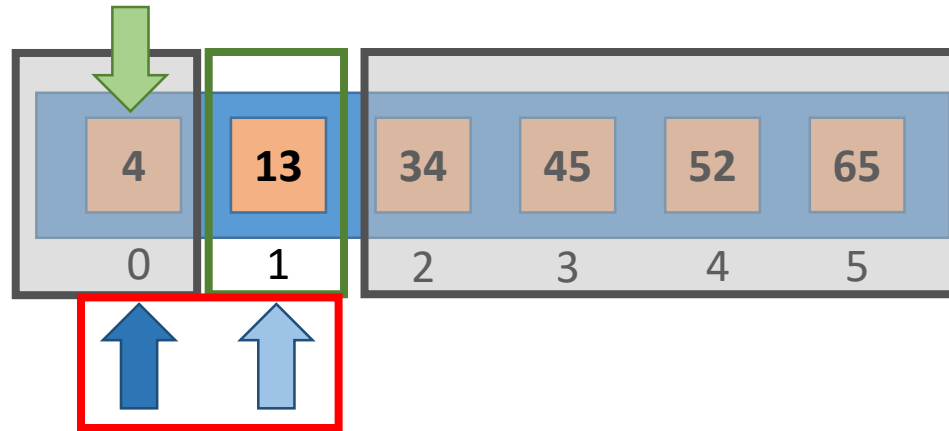
Binary Search (C++ Function)

```cpp
int binarySearch(int a[], int length, int searchValue) {
    int lowBoundary = 0;
    int highBoundary = length - 1;
    while(highBoundary >= lowBoundary) {
        int middle = (highBoundary + lowBoundary) / 2;
        if(searchValue == a[middle]) {
            return middle;
        }
        else if(searchValue > a[middle]) {
            lowBoundary = middle + 1;
        }
        else {
            highBoundary = middle - 1;
        }
    }
    return -1;
}
```

Indicates the value was not found.
The return statement in the loop would never return -1

Binary Search

- The elements **must** be sorted (alphabetically, numerically, some order) or a binary search will not work.

- Best case scenario: The information sought is the middle element.

- Worst case scenario: You check (at most) half of the elements in an array.

Linear Search vs Binary Search

- Linear Searches do not require the array to be sorted.
  - Consider the array: {24, 74, 91, 13, 67, 45, 33, 89}
  - You could not use a binary search here because the array is not in order.

- Binary Searches will eliminate half of the possible values it needs to check after each iteration.
  - This doesn't necessarily mean it is always faster, especially if you need to sort the array first.

Algorithm Analysis

- An algorithm's **complexity** is the time and space needed to complete its processes.

- **Algorithm analysis** is how we determine the complexity of a given algorithm.
  - *Temporal Complexity* – The time it takes an algorithm to run.
  - *Spatial Complexity* – The amount of space an algorithm needs.

Algorithm Analysis

- Software tools called *profilers* are used by programmers to measure runtimes and memory utilization to help to optimize programs.
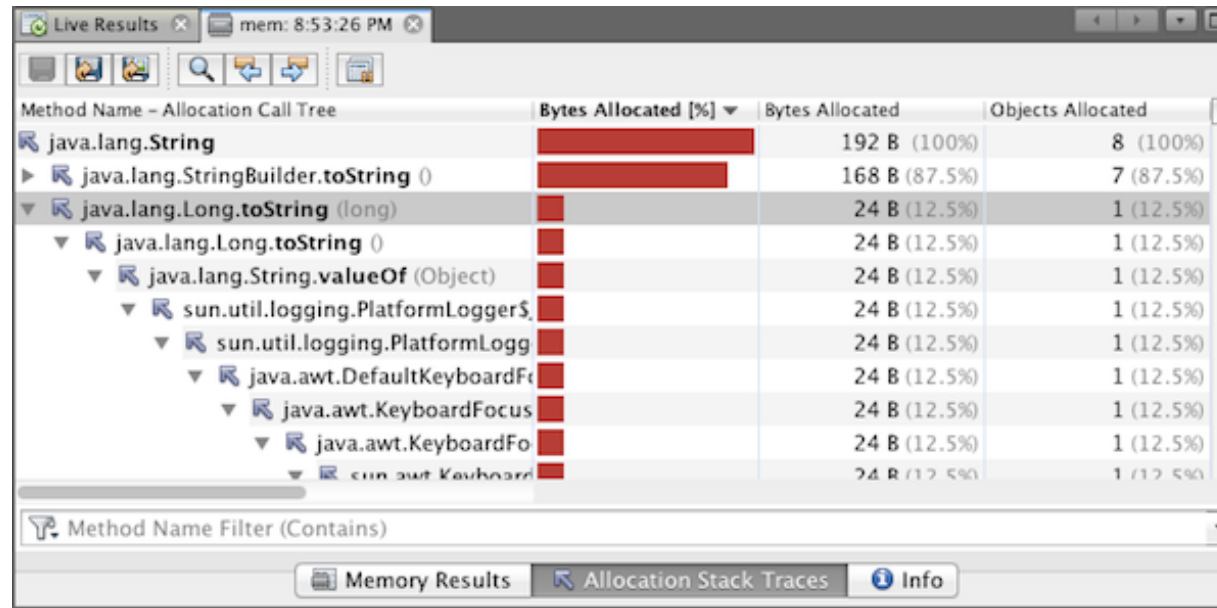
Algorithm Analysis

- Algorithm analysis is indifferent to programming languages or hardware considerations.
  - Profilers help with increasing efficiency but don't tell us about the complexity of an algorithm.

- Algorithm analysis is concerned with comparing and analyzing algorithms for what that they are:
  - Processes and procedures.
  - Solutions to computational problems.

- Algorithm analysis is also concerned with the algorithm's behavior as the input size grows.
  - If an algorithm takes one second to process an array with 1000 elements, how long will it take to process an array with 2000 elements? The answer may not be two seconds.

Algorithm Analysis

- Algorithm analysis produces estimates or generalizations about how an algorithm performs as its input size grows.
  - Independent from a particular programming language, processor speed, etc.
  - No specific units of time or space
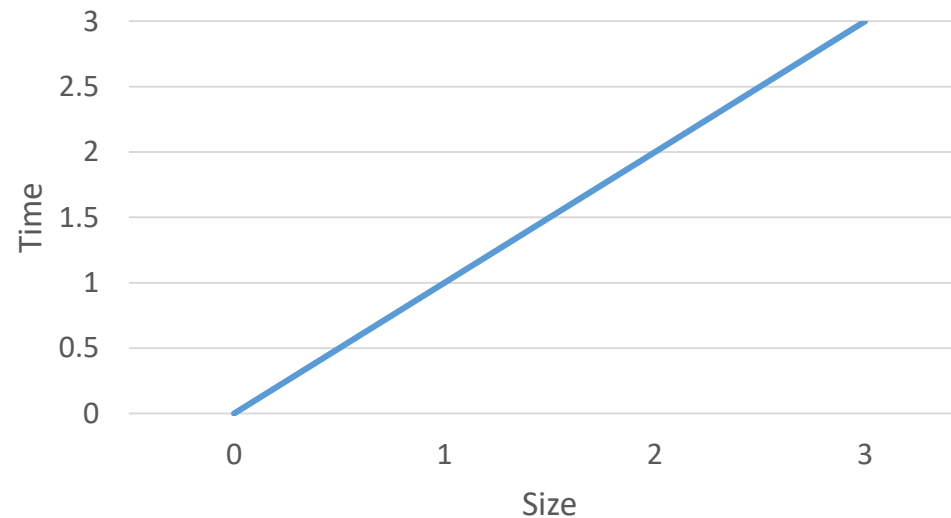
Time Cost of the Linear Search

- When we talk about time complexity, we aren't talking about specific units of time like how many seconds or minutes an algorithm takes to finish.
    - That is entirely dependent on the computer's hardware.

- It's more about the number of operations that need to be performed for the algorithm to complete its task.
    - Regardless of hardware, more operations means a longer time to finish.

Constant Time

- **Constant time** is when an algorithm completes the same number of operations regardless of the input size.
    - T(n) = c
    - The algorithm's time, no matter the value of the input "n", always performs a constant "c" number of operations

- For example, T(n) = 10
    - T(4) = 10
    - T(8) = 10
    - T(12) = 10

Constant Time

- When graphed, an algorithm that runs in constant time would show as a flat line.
  - T(n) = c

- T(n) = 1
  - T(4) = 1
  - T(8) = 1
  - T(12) = 1
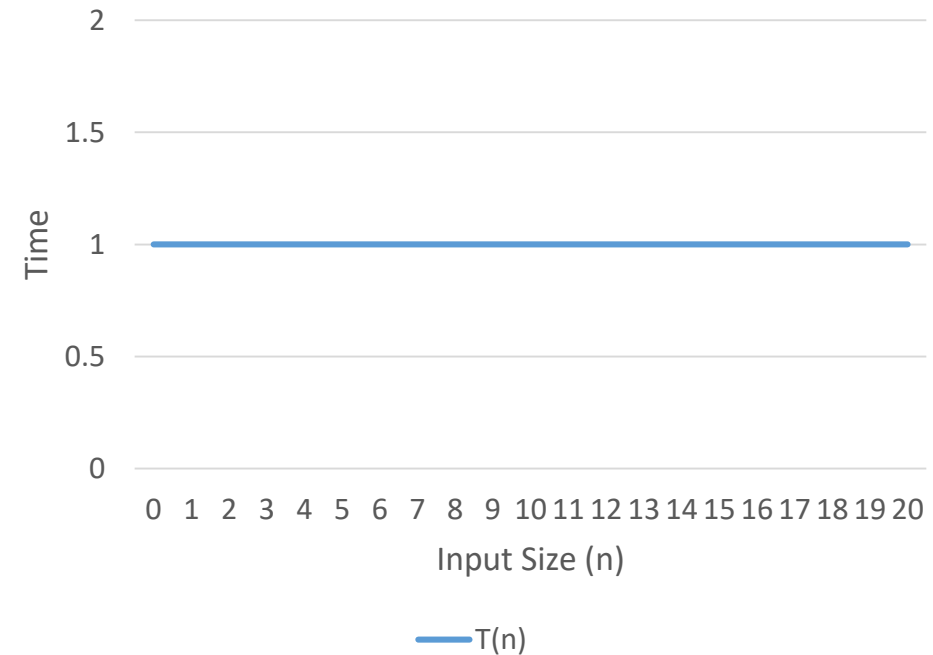
Constant Time

- Fundamental instructions are performed in constant time:
  - Assigning/retrieving a value from a variable.
  - Assigning/retrieving a value from an array.
  - Arithmetic operations.
  - Relational/logical operations and branching (if statement conditions).
  - (For simplicity sake, we'll also say printing a line of output is performed in constant time.)

- Examples:
  - Finding the median value of a sorted array.
    - Always array[length/2]
  - Finding the smallest value in a sorted array.
    - Always array[0] (or array[length-1] if in descending order)

Constant Time

```cpp
int main() {
    int number1 = 10;
    int number2 = 20;
    int number3 = 30;
    int sum = number1 + number2 + number3;

    cout << "The sum is " << sum << endl;
}
```

- Consider the above program.
  - Obviously not a search algorithm, but it shows a program that runs in constant time.
- It simply adds three numbers together and prints the result.

Constant Time

```
int main() {
    int number1 = 10;
    int number2 = 20;
    int number3 = 30;
    int sum = number1 + number2 + number3;

    cout << "The sum is " << sum << endl;
}
```

- It always performs the same number of operations.
  - 4 assignment operations
  - 2 addition operations
  - 1 line of output printed

Constant Time

```cpp
int main() {
    int number1 = 10;
    int number2 = 20;
    int number3 = 30;
    int sum = number1 + number2 + number3;

    cout << "The sum is " << sum << endl;
}
```

- The time cost of this program is 6
  - Again, not 6 "seconds" or "minutes"
  - 6 operations are always performed when this program runs.
    - It will be more than just 6 instructions when compiled, but it will still be a constant number of operations. Again, we're just estimating here.

Time Cost of the Linear Search

```
int linearSearch(int a[], int length, int searchValue) {
    for(int i = 0; i < length; i++) {
        if(a[i] == searchValue) {
            return i;
        }
    }
    return -1;
}
```

- The linear search function above completes the following operations:
  - 1 assignment operation (=)
  - 2 relational operations (< and ==)
  - 1 array retrieval (a[i])
  - 1 increment operation (i++)
  - 1 return statement
    - There are 2 return statements but only one will ever execute.

Time Cost of the Linear Search

- However, it's time cost is not explicitly 6.

- Since some of those operations are part of (and in) a loop, these operations are repeatedly executed.

- The more repetitions, the more operations that need to be performed.

Time Cost of the Linear Search

```
int linearSearch(int a[], int length, int searchValue) {
    for(int i = 0; i < length; i++) {
        if(a[i] == searchValue) {
            return i;
        }
    }
    return -1;
}
```

- The linear search function above completes the following operations:
  - 1 assignment operation **ONLY HAPPENS ONCE**
  - 2 relational operations (< and ==) **REPEATS length TIMES**
  - 1 array retrieval **REPEATS length TIMES**
  - 1 increment operation **REPEATS length TIMES**
  - 1 return statement **ONLY HAPPENS ONCE**

Time Cost of the Linear Search

- This means the time cost is (where "n" is the length of the array):
    - $T(n) = 1 + 2(n) + 1(n) + 1(n) + 1 =$ **4(n) + 2**
    - $T(10) = 4(10) + 2 =$ **42**
    - $T(1000) = 4(1000) + 2 =$ **4002**

- This assumes that the loop will iterate through the entire list.
    - Which is why the value sought being at the end of the array is the worst-case scenario (the algorithm uses the full time cost).
    - If the value sought is at the first element in the array, then the algorithm does the least work (best-case scenario).

Linear Time

- **Linear time** is when the number of operations an algorithm performs grows proportionately with the input size.
  - T(n) = n

- T(n) = n
  - T(4) = 4
  - T(8) = 8
  - T(12) = 12
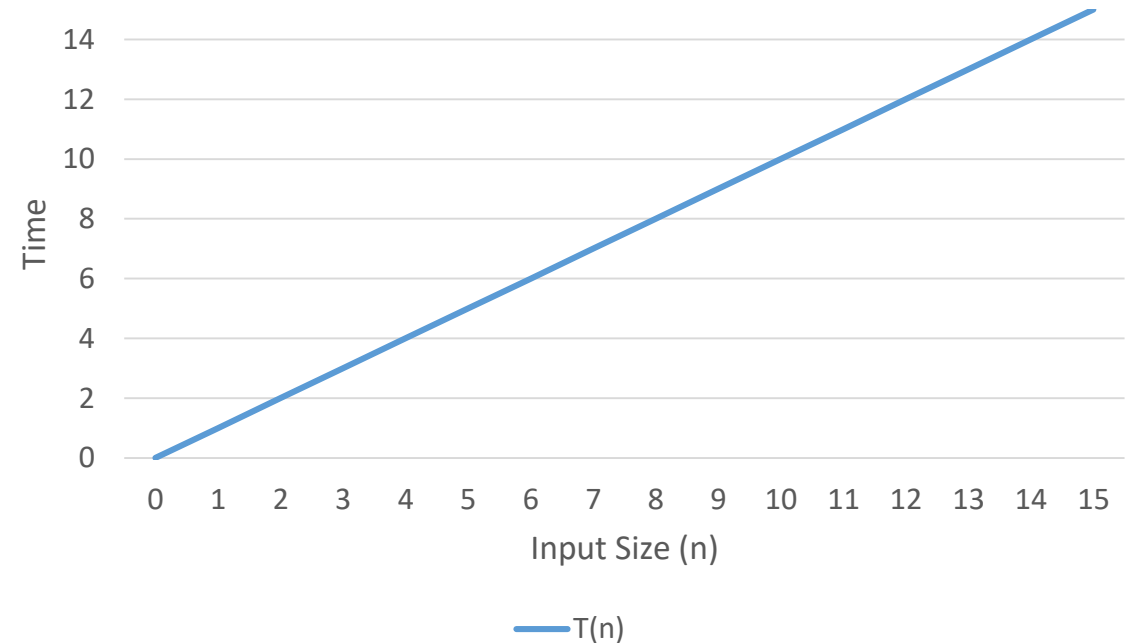
Time Cost of the Linear Search

- The Linear Search performs in linear time.

- Using our example:
  - T(n) = 4(n)+2
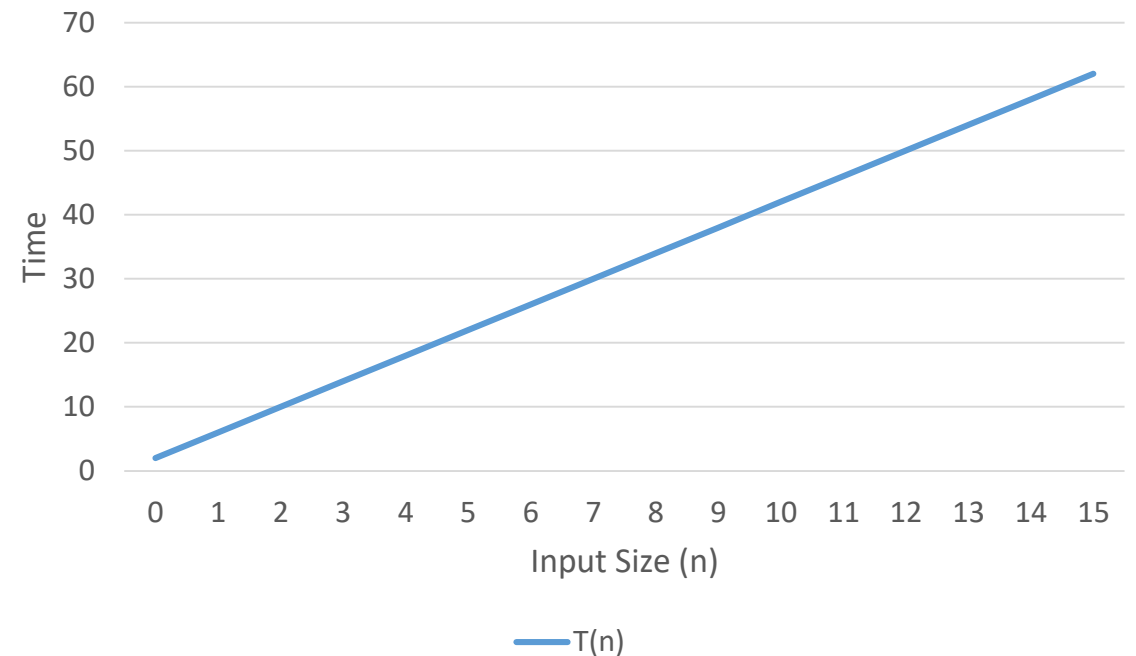
Time Cost of the Binary Search

- The binary search performs more constant time operations than the linear search.

- But it doesn't have to iterate through the entire array.
  - Recall that it "halves" the array every iteration.

Binary Search (C++ Function)

```cpp
int binarySearch(int a[], int length, int searchValue) {
    int lowBoundary = 0;
    int highBoundary = length - 1;
    while(highBoundary >= lowBoundary) {
        int middle = (highBoundary – lowBoundary) / 2;
        if(searchValue == a[middle]) {
            return middle;
        }
        else if(searchValue > a[middle]) {
            lowBoundary = middle + 1;
        }
        else {
            highBoundary = middle - 1;
        }
    }
    return -1;
}
```

Time Cost of the Binary Search

- 2 assignment operations (=) ONLY HAPPENS ONCE
- 3 relational operations (>=, ==, >) HAPPENS EVERY ITERATION
- 2 assignment operations HAPPENS EVERY ITERATION
  - middle is always assigned; either highboundary or lowboundary is reassigned.
- 3 arithmetic operations HAPPENS EVERY ITERATION
  - Middle calculation (- and /); either + 1 or – 1 when recalculating boundary.
- 2 array retrievals HAPPENS EVERY ITERATION
- 1 return statement ONLY HAPPENS ONCE

- This is all a rough estimate

Time Cost of the Binary Search

- In a Binary Search, the search space is halved every iteration.
  - For an array with a length of 20:
  - 20 * 1/2 = **10**; 10 * 1/2 = **5**; 5 * 1/2 = 2.5 or **2**; 2 * 1/2 = **1**;
  - Divided a total of 4 times/4 iterations
  - **20** * $(1/2)^4$ = 1.25 or **1**

- For any length, this can be generalized to:
  - n * $(1/2)^k$ = 1
  - n * $1/2^k$ = 1
  - n = 2^k
  - $\log_2 n = k$

- k is the number of iterations

Time Cost of the Binary Search

- This means the time cost is (where "n" is the length of the array):
  - $T(n) = 2 + 3(\log_2 n) + 2(\log_2 n) + 3(\log_2 n) + 2(\log_2 n) + 1 = \mathbf{10(\log_2 n) + 3}$
  - $T(10) = 10(\log_2 10) + 3 = \mathbf{36}$
  - $T(1000) = 10(\log_2 1000) + 3 = \mathbf{102}$

- This assumes that the loop will need to complete the full number of divisions.
  - Which is why the value sought being checked last is the worst-case scenario (the algorithm uses the full time cost).
  - If the value sought is the middle element in the array, then the algorithm does the least work (best-case scenario).

Logarithmic Time

- **Logarithmic time** is when the number of operations an algorithm performs grows logarithmically with the input size.
  - Uses base 2
  - $T(n) = \log_2 n$

- $T(n) = \log_2 n$
  - $T(4) = \log_2 4$
  - $T(8) = \log_2 8$
  - $T(12) = \log_2 12$
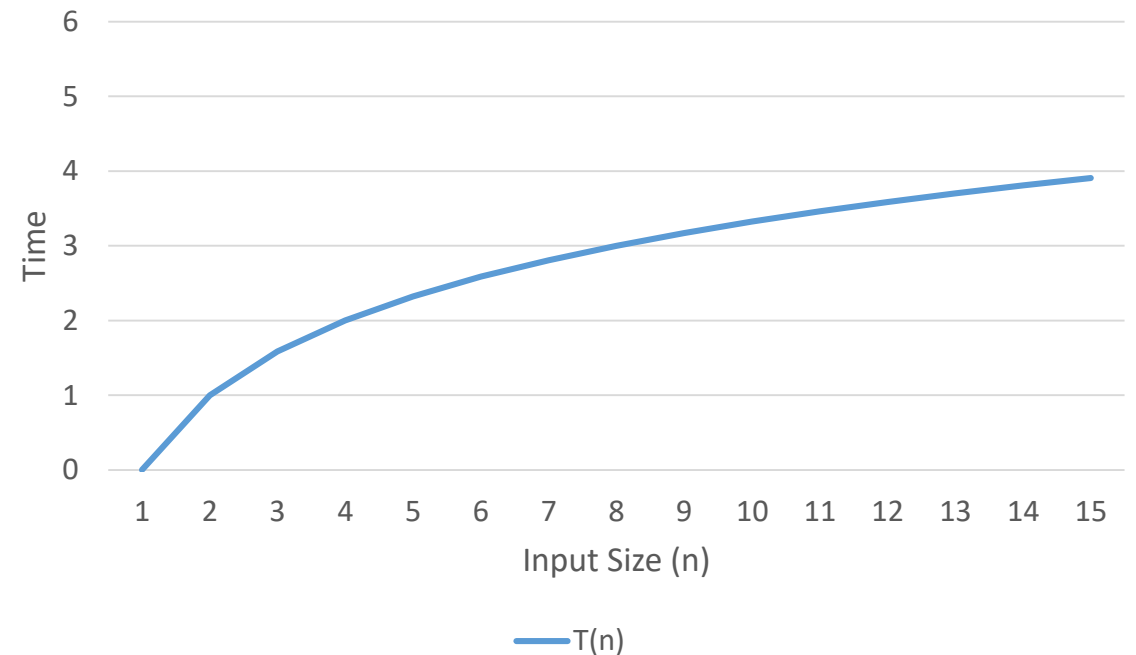
Time Cost of the Binary Search

- The Binary Search performs in logarithmic time.

- Using our example:
  - $T(n) = 10(\log_2 n) + 3$
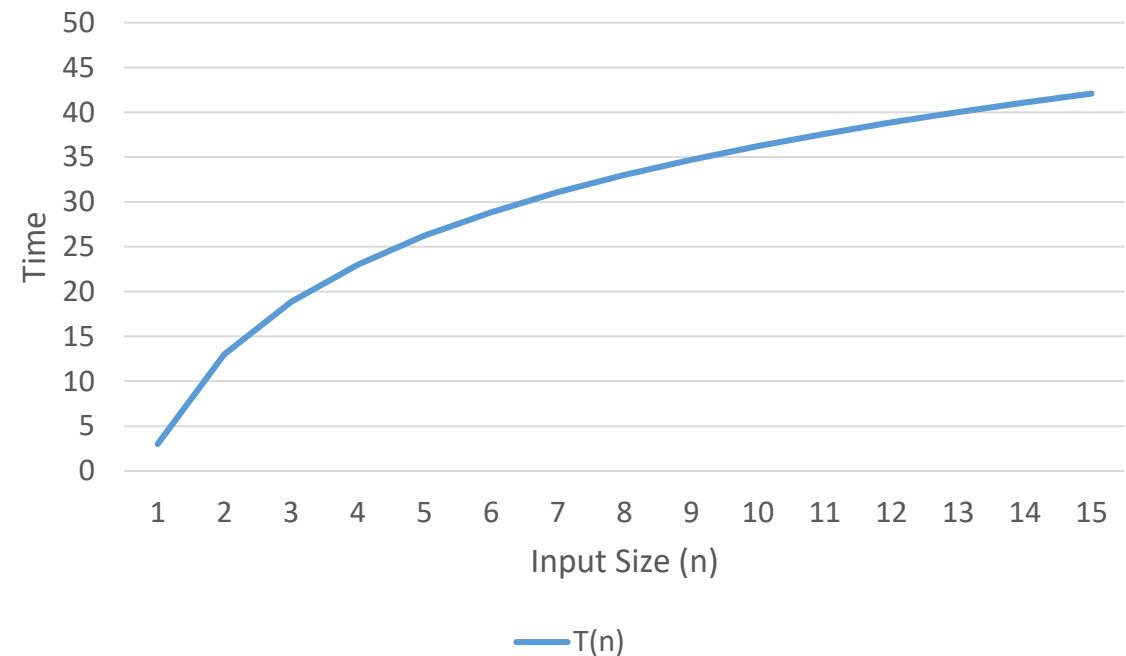
Time Costs of Linear vs Binary Search

- Using our examples:

- Linear Search
  - T(n) = 4(n) + 2

- Binary Search
  - $T(n) = 10(\log_2 n) + 3$
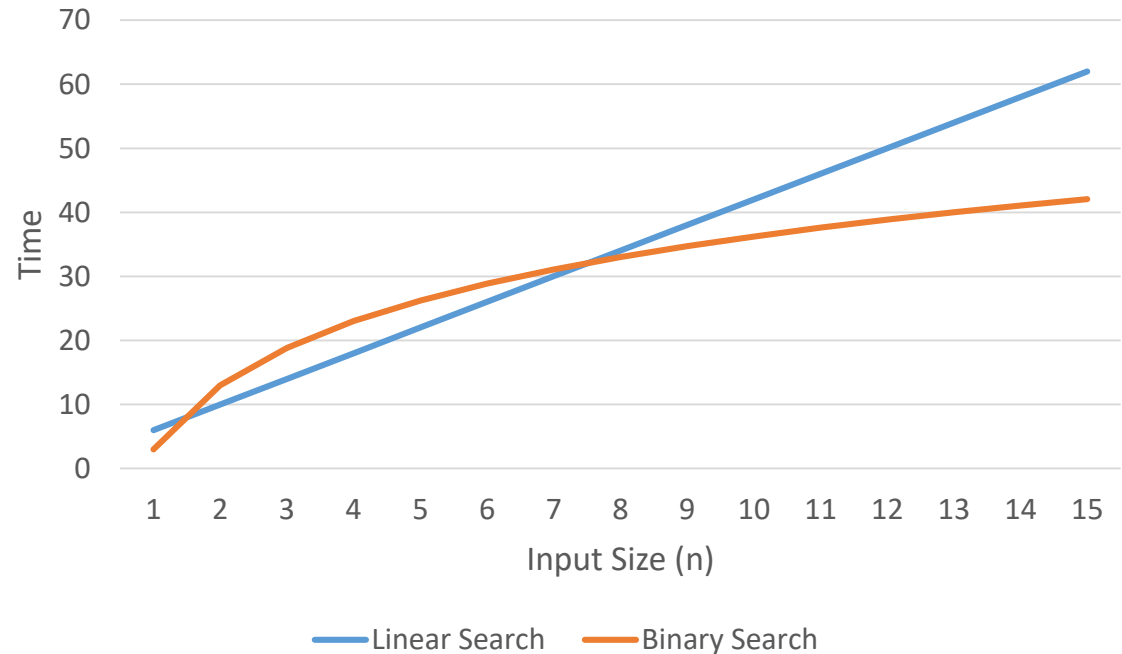
Time Costs of Linear vs Binary Search

- The graph seems to indicate that a linear search performs better than the binary search for searching small arrays (n < 8)
  - It will generally be true that the linear search will be better for small arrays.

- This also assumes the array was sorted for the binary search.
  - Linear search doesn't require a sorted array.
  - Sorting the array would add additional time cost. (We'll come back to this)

Time Costs of Linear vs Binary Search

- It's important to note that the time costs shown in these examples may not be exact.
  - We may have assumed some operations in a loop will always be performed.

- This is all to build a foundation of understanding what the different time complexities (constant, linear, logarithmic) mean.
  - Soon, we'll simply generalize an algorithm to a time complexity and not specific time costs.

Space Complexity

- Space complexity describes the amount of memory used by an algorithm.

- No specific unit of memory.

- **Input Space**- The size of the input to the algorithm

- **Auxiliary Space**- The extra or temporary space needed by an algorithm.

- Space Cost- The total space used with respect to the input.
  - Includes both the input space and the auxiliary space.

Constant Space

- **Constant space** is when an algorithm always uses the same amount of memory.
  - $S(n) = c$

- Could say 1 variable = 1 unit of space
  - Even though some variables will use more space than others.

Constant Space

```
int main() {
    int number1 = 10;
    int number2 = 20;
    int number3 = 30;
    int sum = number1 + number2 + number3;

    cout << "The sum is " << sum << endl;
}
```

- Uses 4 variables.
  - Only and always 4.

- Space cost of this program is 4.

Space Cost of the Linear Search

```
int linearSearch(int a[], int length, int searchValue) {
    for(int i = 0; i < length; i++) {
        if(a[i] == searchValue) {
            return i;
        }
    }
    return -1;
}
```

- The linear search function above requires the following space:
  - The array (the input space)
  - 2 parameters (length and searchValue)
  - 1 variable (i)

Space Cost of the Linear Search

- Input Space: n
    - We don't know how big the array is

- Auxiliary Space: 3

- Space Cost: n + 3

Space Cost of the Linear Search

- This means the space cost is (where "n" is the length of the array):
  - S(n) = **n + 3**
  - S(10) = 10 + 3 = **13**
  - S(1000) = 1000 + 3 = **1003**
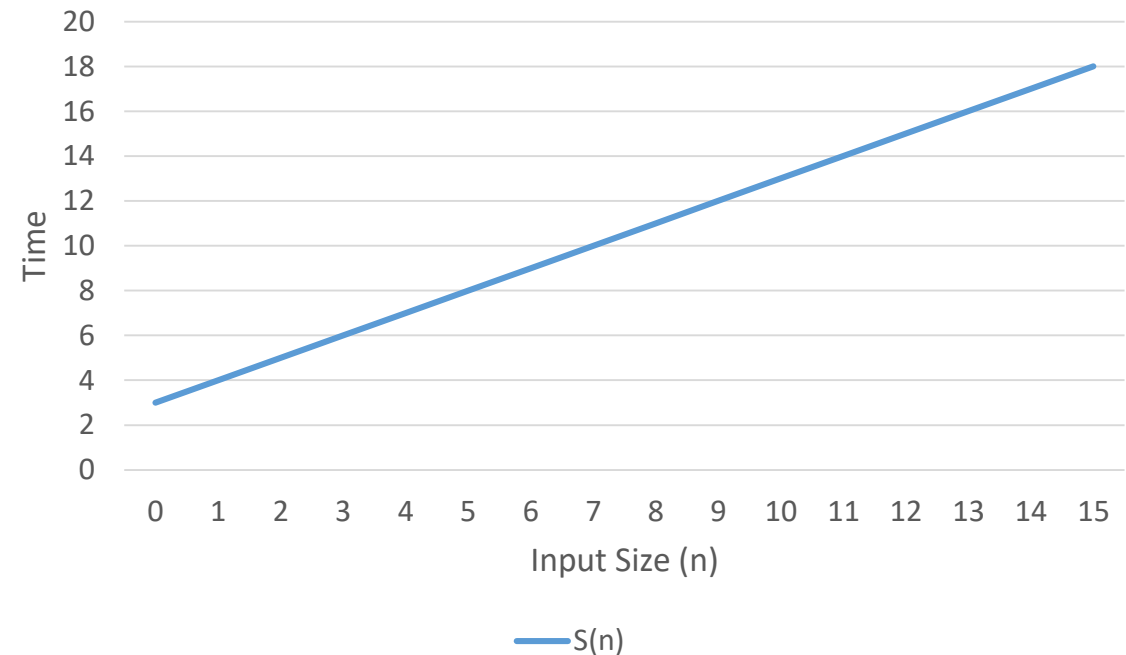
Linear Space

- **Linear space** is when an algorithm memory usage grows proportionately to the input.

- An array uses linear space.
  - If it's a literal array (meaning the array will always have, say, 5 elements) the space would be constant.

- A linear search (like the function demonstrated) typically has linear space complexity.

Space Cost of the Binary Search

- The binary search shown previously requires:
  - The array (the input space)
  - 2 parameters (length and searchValue)
  - 3 variables

Space Cost of the Binary Search

- Input Space: n
  - We don't know how big the array is

- Auxillary Space: 5

- Space Cost: n + 5

Space Cost of the Binary Search

- This means the space cost is (where "n" is the length of the array):
  - S(n) = **n + 5**
  - S(10) = 10 + 5 = **15**
  - S(1000) = 1000 + 5 = **1005**
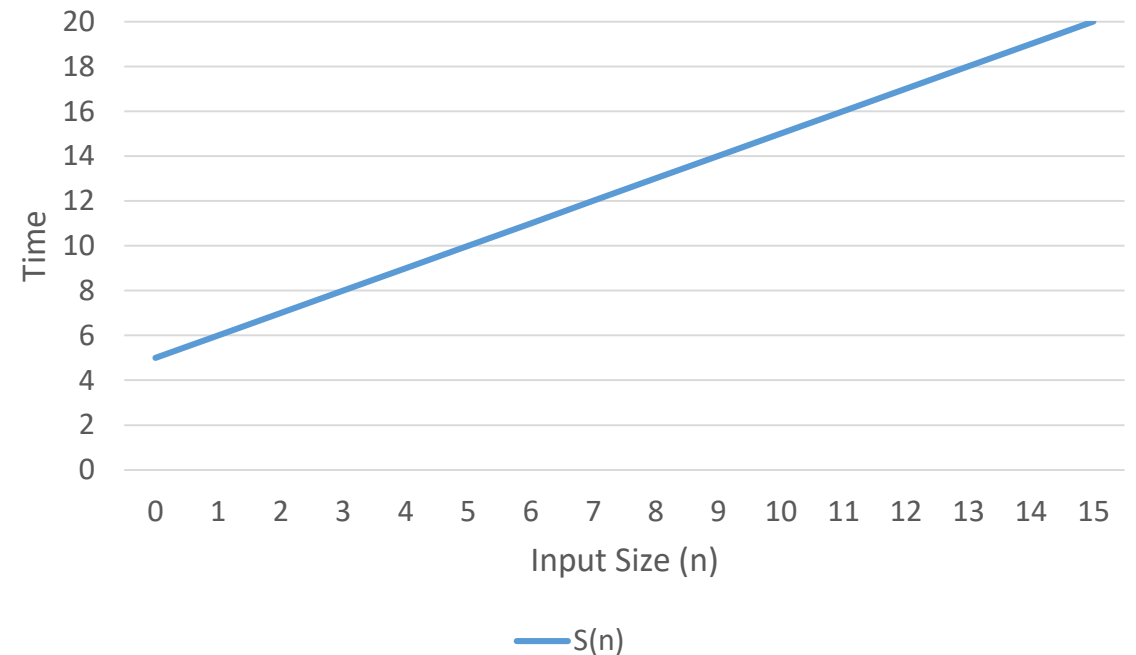
Space Costs of Linear vs Binary Search

- Like the linear search, the binary search also uses linear space.

- The binary search requires a little bit more space, but in most cases will outperform the linear search in lesser time cost.

- Time complexity
  - Linear Search – Linear
  - Binary Search – Logarithmic
- Space complexity
  - Linear Search – Linear
  - Binary Search – Linear

O-Notation

- O-notation ("Big O") is used when we are interested in describing an upper boundary on an algorithm's complexity.
  - Generally used to describe its behavior in the worst-case scenario.

- $O(g(n)) = \{ f(n) :$ there exist positive constant c, and n_0 such that
$$0 \leq f(n) \leq c*g(n) \text{ for all } n \geq n_0 \}$$

O-Notation

- Consider the algorithm below that sums the values of an array.

```
for(int i = 0; i < arrayLength; i++){
    sum += a[i];
}
```

- How many times does sum += a[i] get executed? *n-times*
- O(n)

O-Notation

- We estimated earlier that our Linear Search example had a time cost of around
  - T(n) = 4n + 2

- O(n)
  - O-notation ignores the smaller constants and terms

O-Notation

- We estimated earlier that our Binary Search example had a time cost of around
  - T(n) = 10(log n) + 3

- O(log n)

O-Notation

- O-notation for Linear Search and Binary Search:
    - Linear Search $O(n)$
    - Binary Search $O(\log n)$

Ω-Notation

- Ω-notation ("Big Omega") is used when we are interested in describing a lower boundary on an algorithm's complexity.
  - Generally used to describe its behavior in the best-case scenario.

- $\Omega(g(n)) = \{ f(n) :$ there exist positive constant c, and n_0 such that

$$0 \leq c*g(n) \leq f(n) \text{ for all } n \geq n_0 \}$$

Ω-Notation

- Consider the previously seen algorithm that summed the values of an array.

```
for(int i = 0; i < arrayLength; i++){
    sum += a[i];
}
```

- What is the fewest number of times that sum += a[i] gets executed? *Once,* if the array's length is 1
- $\Omega(1)$

Ω-Notation

- Ω-notation for Linear Search and Binary Search:
    - Linear Search   Ω(1)
    - Binary Search   Ω(1)

Θ-Notation

- Θ-notation ("Big Theta") is used when we are interested in describing both the upper and lower boundaries on an algorithm's complexity.
  - "Average" complexity
  - Sandwiched between Ω and O

- $\Theta(g(n))$ = { $f(n)$: there exist positive constants c_1, c_2, and n_0 such that

$$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ for all } n \geq n_0 \}$$

Θ-Notation

- Consider the same example that sums the values of an array.

```
for(int i = 0; i < arrayLength; i++){
    sum += a[i];
}
```

- Which is more likely to happen?
  - The array has a length of 1 (Constant Time)
    - One possible scenario
  - The array has a length greater than one (Linear Time)
    - Every other possible scenario

- $\Omega(1) \leq \Theta(?) \leq O(n)$
  - $\Theta(\mathbf{n})$

Θ-Notation

- Θ-notation for Linear Search and Binary Search:
    - Linear Search   Θ(n)
    - Binary Search   Θ(log n)

Space Complexities

- O-notation for algorithms shown (Auxiliary space only):
  - Linear Search   O(1)
  - Binary Search   O(1)

Sorting Algorithms

- A **sorting algorithm** is an algorithm that organizes a sequence of data (like an array) into some type of order.
  - Usually ascending or descending order.

- An **ascending sort** means arranging the values from smallest to largest.
  - {3, 4, 2, 5, 1}   {1, 2, 3, 4, 5}

- A **descending sort** means arranging the values from largest to smallest.
  - {3, 4, 2, 5, 1}   {5, 4, 3, 2, 1}

Bubble Sort Algorithm

- The Bubble Sort algorithm compares neighboring pairs of values and swaps them if they are in the wrong order.

- The algorithm repeats this process *n*-number of times, where *n* is the length of the array.

Bubble Sort Algorithm

- Here is a link to a video with a visualization of the Bubble Sort sorting a short array of numbers:

    https://www.youtube.com/watch?v=xli_FI7CuzA

- The pseudocode at the end of the video is a little different from the C++ implementation on the next slide.

Bubble Sort (C++ Function)

```cpp
void bubbleSort(int a[], int length) {
    for(int i = 0; i < length; i++) {
        for(int j = 1; j < length; j++) {
            if(a[j-1] > a[j]) {
                int temp = a[j-1];
                a[j-1] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

Selection Sort Algorithm

- The Selection Sort algorithm divides the array into sorted and unsorted partitions.

- The algorithm looks through the unsorted partition to find the smallest value.
  - If performing an ascending sort.

- The smallest value found is swapped to the beginning of the unsorted partition, thus extending the sorted partition by 1.

Selection Sort Algorithm

- Here is a link to a video with a visualization of the Selection Sort sorting a short array of numbers.

  https://www.youtube.com/watch?v=g-PGLbMth_g

- The pseudocode at the end of the video is a little different from the C++ implementation on the next slide.

Selection Sort (C++ Function)

```cpp
void selectionSort(int a[], int length) {
    for(int i = 0; i < length-1; i++) {
        int smallest = i;
        for(int j = i+1; j < length; j++) {
            if(a[j] < a[smallest]) {
                smallest = j;
            }
        }
        if(smallest != i) {
            int temp = a[smallest];
            a[smallest] = a[i];
            a[i] = temp;
        }
    }
}
```

Insertion Sort Algorithm

- The Insertion Sort algorithm, like the Selection Sort, divides an array into sorted and unsorted partitions.

- The algorithm takes the first element in the unsorted portion, and keep swapping backwards until it finds a smaller values in the sorted portion.
  - If performing an ascending sort

Insertion Sort Algorithm

- Here is a link to a video with a visualization of the Insertion Sort sorting a short array of numbers.

  https://www.youtube.com/watch?v=JU767SDMDvA

- The pseudocode at the end of the video is a little different from the C++ implementation on the next slide.

Insertion Sort (C++ Function)

```cpp
void insertionSort(int a[], int length) {
    for(int i = 1; i < length; i++) {
        int value = a[i];
        int j = i-1;
        while(j >= 0 && a[j] > value) {
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = value;
    }
}
```

Time Cost of the Bubble Sort

```
void bubbleSort(int a[], int length) {
    for(int i = 0; i < length; i++) {
        for(int j = 1; j < length; j++) {
            if(a[j-1] > a[j]) {
                int temp = a[j-1];
                a[j-1] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

- The bubbleSort function above completes the following operations:
  - 1 assignment operations (=) **REPEATS length TIMES**
  - 1 relational operations (<) **REPEATS length TIMES**
  - 1 increment operation (i++) **REPEATS length TIMES**
  - Total time cost: Length * 3
  - (Continues on next slide)

Time Cost of the Bubble Sort

```
void bubbleSort(int a[], int length) {
    for(int i = 0; i < length; i++) {
        for(int j = 1; j < length; j++) {
            if(a[j-1] > a[j]) {
                int temp = a[j-1];
                a[j-1] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

- The bubbleSort function above completes the following operations:
  - 1 assignment operation (=) **REPEATS length - 1 TIMES**
  - 2 relational operations (< and >) **REPEATS length - 1 TIMES**
  - 1 increment operation (j++) **REPEATS length - 1 TIMES**
  - 1 subtraction operation (-) **REPEATS length - 1 TIMES**
  - 2 array retrievals **REPEATS length - 1 TIMES**
  - Total time cost: (Length - 1) * 7
  - (Continues on next slide)

Time Cost of the Bubble Sort

```
void bubbleSort(int a[], int length) {
    for(int i = 0; i < length; i++) {
        for(int j = 1; j < length; j++) {
            if(a[j-1] > a[j]) {
                int temp = a[j-1];
                a[j-1] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

- The bubbleSort function above completes the following operations:
  - 3 assignment operations (=)
  - 2 subtraction operations (-)
  - 2 array retrievals
  - Total time cost: 7
  - (This only happens when the if statement is true)

Time Cost of the Bubble Sort

```
void bubbleSort(int a[], int length) {
    for(int i = 0; i < length; i++) {
        for(int j = 1; j < length; j++) {
            if(a[j-1] > a[j]) {
                int temp = a[j-1];
                a[j-1] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

- The inner for loop executes to completion for every iteration of the outer loop.
  - The inner for loop executes a total of *length* times.

Time Cost of the Bubble Sort

- This means the time cost is (where "n" is the length of the array):
  - $T(n) = 1 + 3n + (7(n-1) + 7)n =$
  - $1 + 3n + (\textbf{7n - 7 + 7})n =$
  - $1 + 3n + \textbf{(7n)n} =$
  - $1 + 3n + \textbf{7n}^2 =$
  - $\textbf{7n}^2 \textbf{+ 3n + 1}$

- $T(n) = 3n + ((7n-1) + 7)n + 1$

Outer loop

Inner loop

Outer loop counter initialization

Time Cost of the Bubble Sort

- This means the time cost is (where "n" is the length of the array):
  - T(n) = **$7n^2 + 3n + 1$**

  - T(10) = $7(10)^2 + 3(10) + 1 = $ **731**
  - T(1000) = $7(1000)^2 + 3(1000) + 1 = $ **7003001**

- This is assuming a swap will always occur.
  - We know a swap won't *always* happen.
  - But this gives us a good idea how the algorithm will behave in the worst case scenario:
    - The array's elements are in the opposite order of sorting.
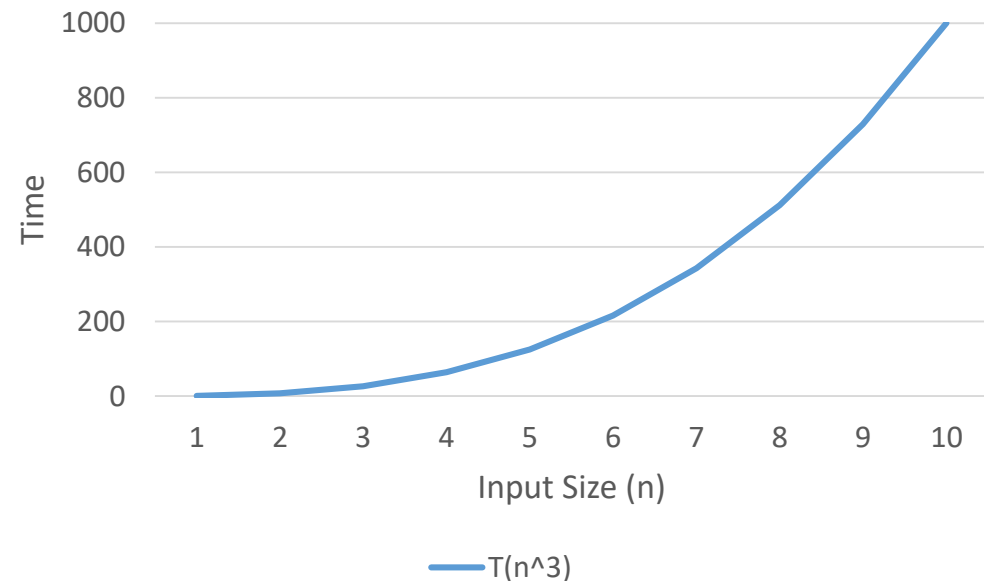
Time Cost of the Bubble Sort

- It also tells a bit about how the algorithm will behave in the best case scenario:
  - The array's elements are already in order.

- It won't do any swaps, but the outer loop will always iterate n times.
  - Forcing the inner loop to iterate n-1 times for every iteration of the outer loop.
  - n(n-1) = **n² - n**
    - (Ignores constant time operations)

- The time cost will still involve a factor of n^2

Polynomial Time

- **Polynomial time** is when the number of operations an algorithm performs is a polynomial function of the input size.
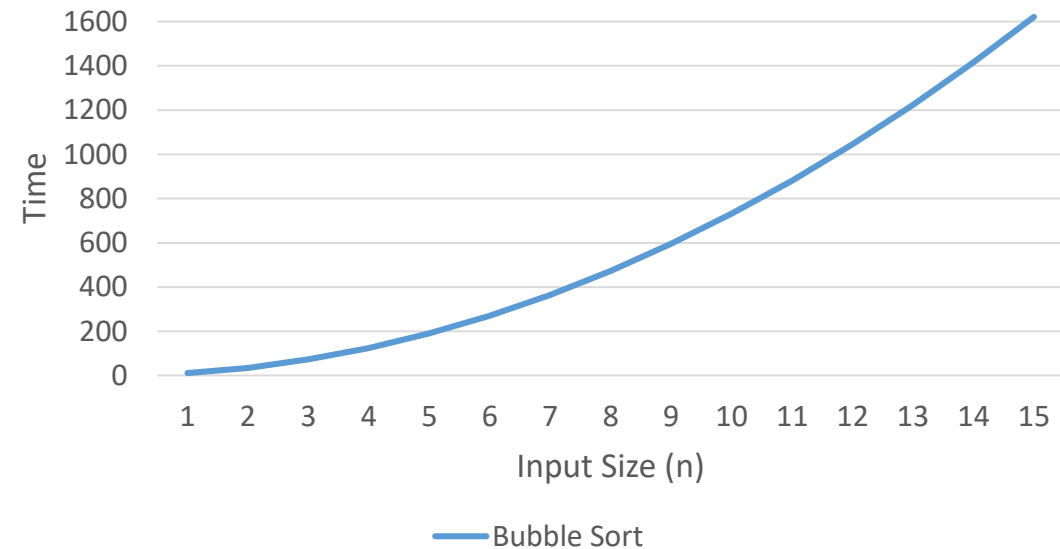 - $T(n) = n^k$
  - Where k is a positive integer

- $T(n) = n^3$
  - $T(4) = 64$
  - $T(8) = 512$
  - $T(12) = 1728$

- <u>Algorithms involving nested loops generally perform in polynomial time.</u>
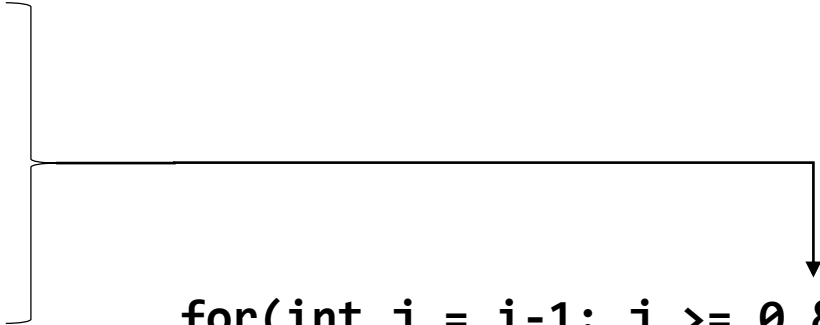
Time Cost of the Bubble Sort

- The Bubble Sort performs in polynomial time.

- Using our example:
  - $T(n) = 7n^2 + 3n + 1$

Time Cost of Selection and Insertion Sort

- Both algorithms contain a nested loop.
  - A tell-tale sign of polynomial time.

- The insertion sort's while loop could be re-written as:

```
int j = i-1;
while(j >= 0 && a[j] > value) {
    a[j+1] = a[j];
    j--;
}
```

```
for(int j = i-1; j >= 0 && a[j] > value; j--) {
    a[j+1] = a[j];
}
```

# Time Cost of Bubble, Selection, and Insertion Sort

- All three have polynomial time cost.
  - The bubble sort (as shown) typically does worse than selection and insertion sort.

- Insertion sort is usually the most efficient of the three, since it only goes "backwards" as far as necessary to insert a value in the sorted portion.
  - It doesn't need to always iterate through the entire sorted portion.

- Selection sort always reads the entirety of the unsorted portion.
  - Though, the unsorted portion shrinks with every iteration.

Space Cost of Bubble, Selection, and Insertion Sort

- All three have linear space cost.
  - The variables in the example functions all use constant space.
  - The only thing that may ever differ/change is the length of the array that is being sorted.

Sorting + Searching

- Earlier in the lecture, we saw the binary search did better than the linear search once the array got big enough.
  - We also assumed the array was already in order.
 - What if it wasn't?
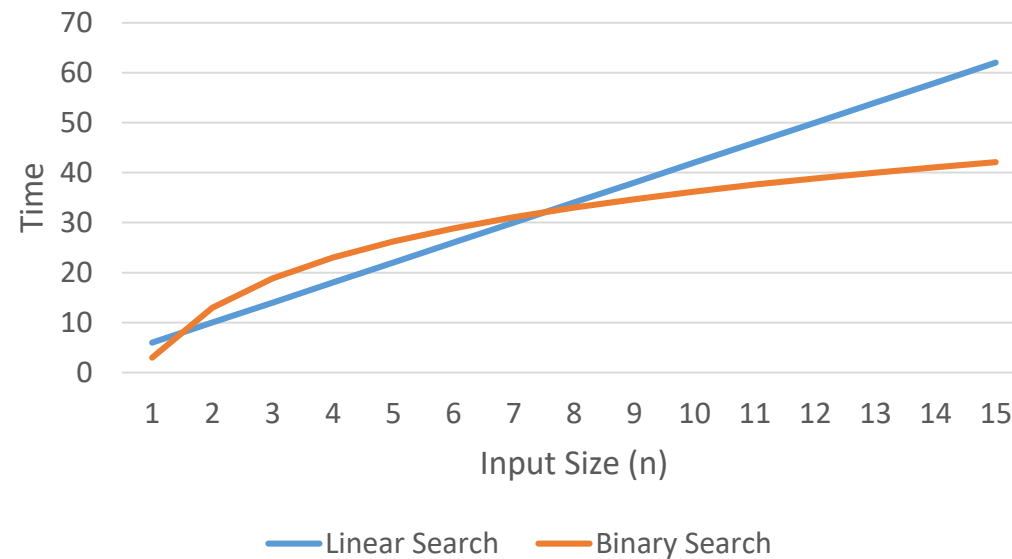
Sorting + Searching

- Let's say we want to perform a bubble sort prior to searching the array with a binary search.
  - The linear search doesn't care about order.

- For simplicity, we'll use the following for time costs:
  - Linear search – $T(n)$
  - Bubble Sort – $T(n^2)$
  - Binary Search – $T(\log_2 n)$

Sorting + Searching

- It appears as though it is typically faster to simply perform a linear search rather than spend the time to sort the array before using a binary search.
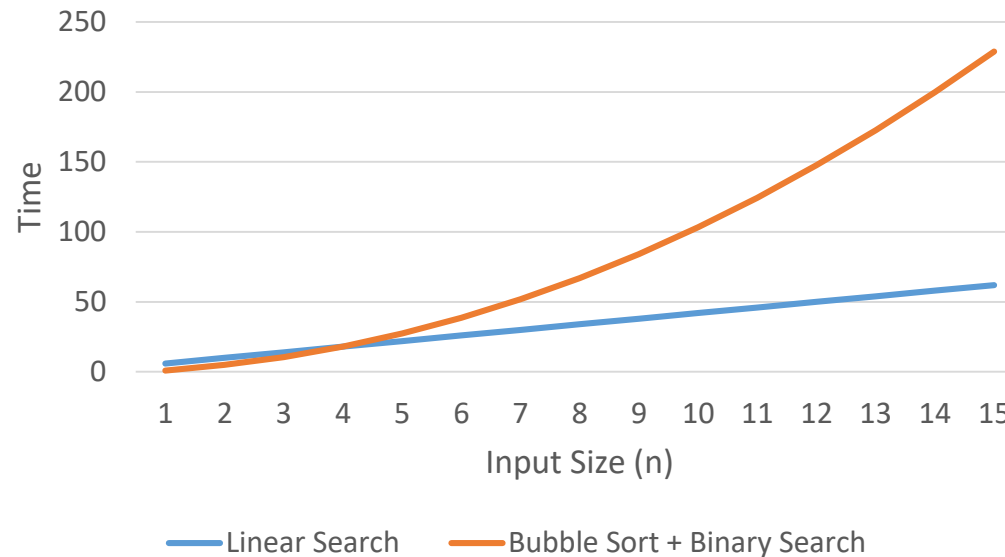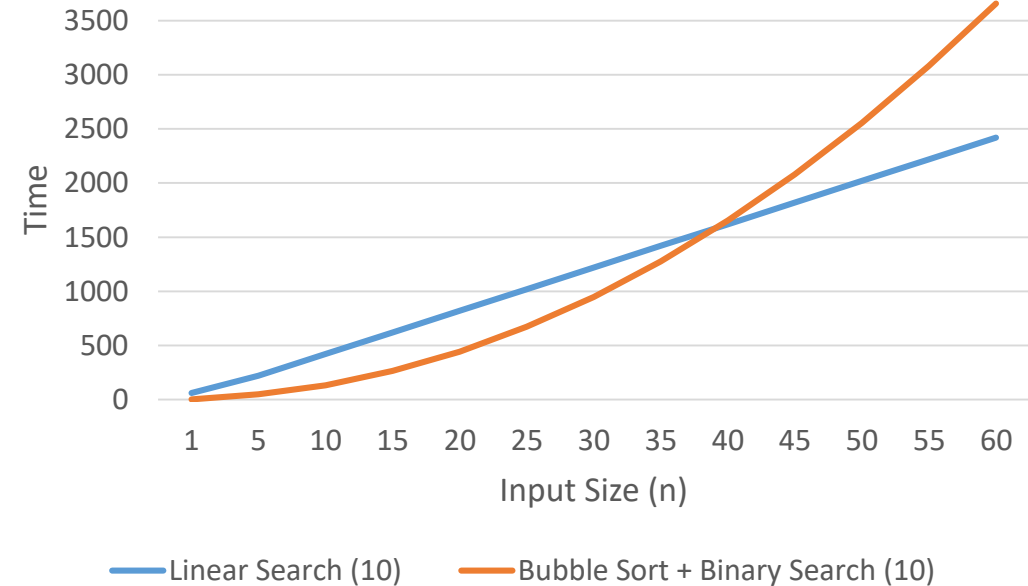
Sorting + Searching

- But the array only needs to be sorted once.

- What if we needed to search the array 10 times in our program?
  - Is it worth spending the time sorting the array that one time for multiple uses of the binary search?
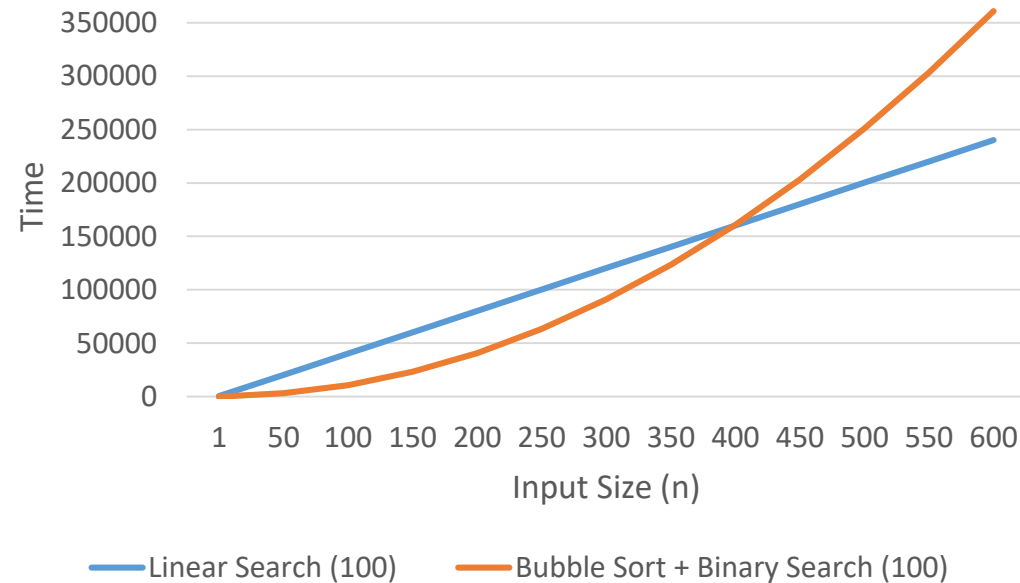
Sorting + Searching

- It appears it was worth it if the array was about 40 elements or less.
  - What if we had to search it 100 times?

Sorting + Searching

- For arrays with less than around 400 elements, it was better to perform the sort first before doing the 100 searches with the binary search vs the linear search (no sorting needed).

Sorting + Searching

- Hopefully this illustrates why algorithm analysis is important for choosing the most efficient solution to a computing problem.

- It's important to note there are better sorting algorithms than bubble, selection, and insertion sort.
  - We'll see those faster algorithms in the Algorithm Complexity II lecture.

Asymptotic Notations

- Worst Case:
  - Bubble Sort $\qquad\qquad$ $O(n^2)$
  - Insertion Sort $\qquad\quad$ $O(n^2)$
  - Selection Sort $\qquad\quad$ $O(n^2)$
- Best Case:
  - Bubble Sort $\qquad\qquad$ $\Omega(n^2)$
  - Insertion Sort $\qquad\quad$ $\Omega(n)$
  - Selection Sort $\qquad\quad$ $\Omega(n^2)$
- Average Case:
  - Bubble Sort $\qquad\qquad$ $\Theta(n^2)$
  - Insertion Sort $\qquad\quad$ $\Theta(n^2)$
  - Selection Sort $\qquad\quad$ $\Theta(n^2)$

Complexities Seen Thus Far

Time

50
45
40
35
30
25
20
15
10
5
0

1   2   3   4   5   6   7   8   9   10

Input Size (n)

— Constant - O(1)   — Logarithmic - O(log n)   — Linear - O(n)   — Polynomial - O(n^2)