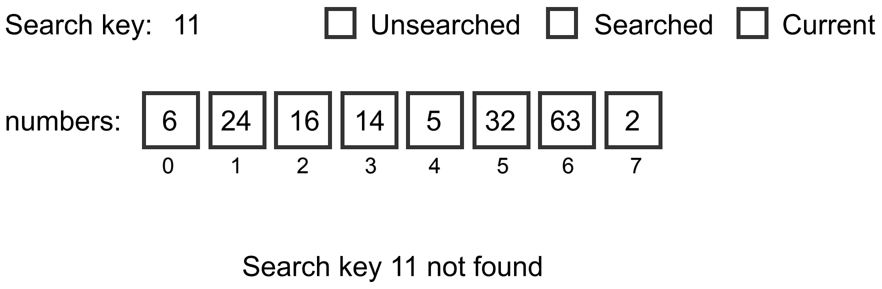
2.1 Searching and algorithms

Algorithms

An ***algorithm*** is a sequence of steps for accomplishing a task. ***Linear search*** is a search algorithm that starts from the beginning of an array, and checks each element until the search key is found or the end of the array is reached.

PARTICIPATION ACTIVITY 2.1.1: Linear search algorithm checks each element until key is found.



Animation content:

Static figure: An array named "numbers" is shown: [6, 24, 16, 14, 5, 32, 63, 2].

Step 1: Linear search starts at the first element and compares elements one-by-one.
The search key is set to 14. Numbers are highlighted as each is compared against the search key. At the end, searched elements 6, 24, and 16 are highlighted in gray. The current element 14 is highlighted in blue. Remaining unsearched elements 5, 32, 63, and 2 are highlighted in white.

Step 2: Linear search must compare all elements to determine that key 11 is not present.
The search key is set to 11. Numbers are highlighted as each is compared against the search key. At the end, all elements have been searched, and so are highlighted in gray.

Animation captions:

1. Linear search starts at the first element and compares elements one-by-one.

2. Linear search must compare all elements to determine that key 11 is not present.

Figure 2.1.1: Linear search algorithm.

```
LinearSearch(numbers, numbersSize, key) {
   i = 0

   for (i = 0; i < numbersSize; ++i) {
      if (numbers[i] == key) {
         return i
      }
   }

   return -1 // not found
}

main() {
   numbers = {2, 4, 7, 10, 11, 32, 45, 87}
   NUMBERS_SIZE = 8
   i = 0
   key = 0
   keyIndex = 0

   print("NUMBERS: ")
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()

   print("Enter a value: ")
   key = getIntFromUser()

   keyIndex = LinearSearch(numbers, NUMBERS_SIZE, key)

   if (keyIndex == -1) {
      printLine(key + " was not found.")
   }
   else {
      printLine("Found " + key + " at index " + keyIndex + ".")
   }
}
```

```
NUMBERS: 2 4 7 10 11 32 45 87
Enter a value: 10
Found 10 at index 3.
...
NUMBERS: 2 4 7 10 11 32 45 87
Enter a value: 17
17 was not found.
```

PARTICIPATION ACTIVITY 2.1.2: Linear search algorithm execution.

Given array: [20, 4, 114, 23, 34, 25, 45, 66, 77, 89, 11].
How many array elements does linear search compare against each search key below?

1) Search key: 77

Check Show answer

2) Search key: 114

Check Show answer

3) Search key: 58

Check Show answer

Linear search efficiency

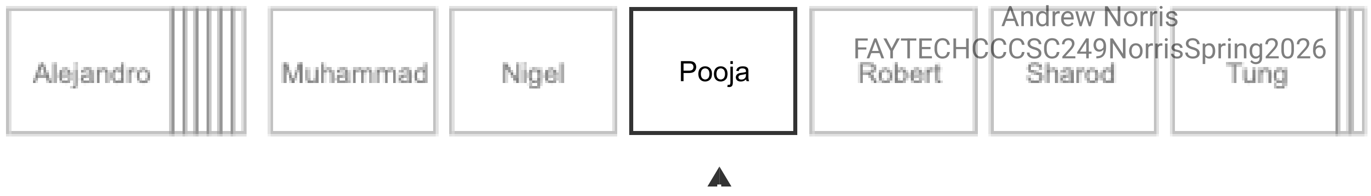
In the worst case, linear search compares the search key against all array elements. In the best case, linear search compares the search key against only one element, the array's first element. So if comparing an array element against the search key is a constant time operation, then linear search's worst case runtime complexity is $O(N)$ *and the best case is* $O(1)$.

2.2 Binary search

Linear search vs. binary search

Linear search may require searching all array elements, which can lead to long runtimes. For example, searching for a contact on a smartphone one-by-one from first to last can be time consuming. Because contacts are sorted, a faster search, known as binary search, checks the middle contact first. If the desired contact comes alphabetically before the middle contact, binary search will then search the first half and otherwise the last half. Each step reduces the contacts that need to be searched by about half.

PARTICIPATION ACTIVITY 2.2.1: Using binary search to search contacts on your phone.



Animation content:

Static figure:
Contacts are shown: Alejandro, Muhammad, Nigel, Pooja, Robert, Sharod, and Tung. A list of hidden contacts are peeking out from behind Alejandro as well as Tung. An arrow, representing the contact checked during a search, is positioned under Pooja.

Step 1: Contacts are stored sorted by name. Searching for Pooja using a binary search starts by checking the middle contact.
The first, middle, and last contacts are revealed: Alejandro, Muhammad, and Nigel. An arrow is revealed under Muhammad, indicating that the middle contact is search first.

Step 2: The middle contact is Muhammad. Pooja is alphabetically after Muhammad, so the binary search only searches the contacts after Muhammad. Only half the contacts now need to be searched.
The arrow moves from the middle contact, Muhammad, to the rest of the contacts starting with Nigel.

Step 3:Binary search continues by checking the middle element between Muhammad and the last contact.
Sharod is revealed as the middle contact between Nigel and the last contacts which is revealed to start with Tung.
The arrow moves to under Sharod.

Step 4: Pooja is before Sharod, so the search continues with only those contacts between Muhammad and Sharod, which is one fourth of the contacts.

The arrow moves backward to contacts between Nigel and Sharod.

Step 5: The middle element between Muhammad and Sharod is Pooja. Each step reduces the number of contacts to search by about half.
All contacts between Muhammad and Sharod are revealed. The contacts between Muhammad and Sharod are Nigel, Pooja, and Robert. Arrow moves to under Pooja.

Animation captions:

1. Contacts are stored sorted by name. Searching for Pooja using a binary search starts by checking the middle contact.
2. The middle contact is Muhammad. Pooja is alphabetically after Muhammad, so the binary search only searches the contacts after Muhammad. Only half the contacts now need to be searched.
3. Binary search continues by checking the middle element between Muhammad and the last contact.
4. Pooja is before Sharod, so the search continues with only those contacts between Muhammad and Sharod, which is one fourth of the contacts.
5. The middle element between Muhammad and Sharod is Pooja. Each step reduces the number of contacts to search by about half.

PARTICIPATION ACTIVITY 2.2.2: Using binary search to search an array of contacts.

The following array of contacts is searched for Bob: [Amy, Bob, Chris, Holly, Ray, Sarah, Zoe]

1) What is the first contact searched?

Check **Show answer**

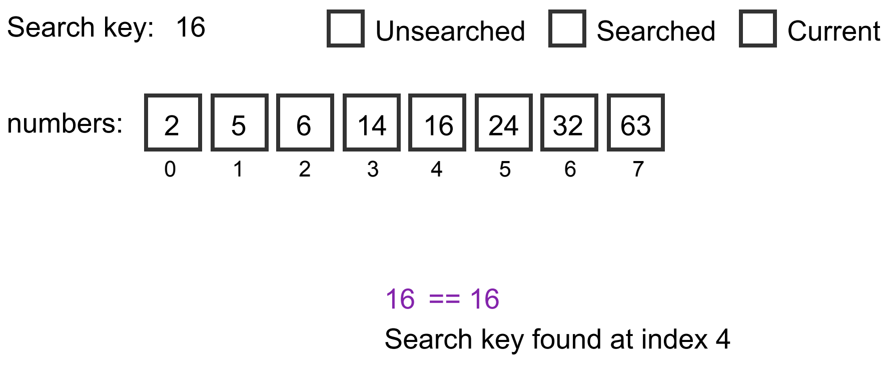
2) What is the second contact searched?

Check **Show answer**

Binary search algorithm

Binary search is an algorithm for searching a sorted array. Binary search first checks the middle element of the array. If the search key is found, the algorithm returns the matching location. If the search key is not found, the algorithm repeats the search on the remaining left subarray (if the search key was less than the middle element) or the remaining right subarray (if the search key was greater than the middle element).

PARTICIPATION ACTIVITY 2.2.3: Binary search efficiently searches a sorted array.



Animation content:

Static figure:
A list named "numbers" contains 8 elements: [2, 5, 6, 14, 16, 24, 32, 63]. Index labels are shown under corresponding list elements starting from 0 to 7. Elements 14 and 24 have a gray background indicating that the elements have been searched. The list element 16 has a light blue background indicating that 16 is the current element being looked at. A label "Search key: 16" is shown above the array. The labels "16 == 16" and "Search key found at index 4" are shown below the list.

Step 1: Indices low and high indicate the range to search. Initially, low is 0 and high is 7, meaning the entire list must be searched.
The list named "numbers" is revealed with elements: 2, 5, 6, 14, 16, 24, 32, 63. Index labels reveal under corresponding list elements: low = 0 and high = 7.

Step 2: Search starts by checking the middle element.
Index named "mid" is computed as (high + low) // 2 = 3. The list element at index 3, 14, is compared with 16 and is not equal.

Step 3: The search key is greater than the element, so only elements in the right subarray must be searched.
low moves to 4 and mid to 5.

Step 4: Each iteration reduces search space by about half. Search continues until the key is found or the search space is empty.
16 does not equal 24, so high and mid get recalculated, each then equal to 4. The list element at index 4 is 16, so the search key is found.

Animation captions:

1. Indices low and high indicate the range to search. Initially, low is 0 and high is 7, meaning the entire array must be searched.
2. The middle index is computed. Integer division is used, so (7 + 0) / 2 = 3.
3. The search key is compared to the middle element.
4. The search key is greater than the element, so only elements in right subarray must be searched.
5. Each iteration reduces search space by about half. Search continues until the key is found or the search space is empty.

Figure 2.2.1: Binary search algorithm.

```
BinarySearch(numbers, numbersSize, key) {
   mid = 0
   low = 0
   high = numbersSize - 1

   while (high >= low) {
      mid = (high + low) / 2
      if (numbers[mid] < key) {
         low = mid + 1
      }
      else if (numbers[mid] > key) {
         high = mid - 1
      }
      else {
         return mid
      }
   }

   return -1 // not found
}

main() {
   numbers = [ 2, 4, 7, 10, 11, 32, 45, 87 ]
   NUMBERS_SIZE = 8
   i = 0
   key = 0
   keyIndex = 0

   print("NUMBERS: ")
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()

   print("Enter a value: ")
   key = getIntFromUser()

   keyIndex = BinarySearch(numbers, NUMBERS_SIZE, key)

   if (keyIndex == -1) {
      printLine(key + " was not found.")
   }
   else {
      printLine("Found " + key + " at index " + keyIndex + ".")
   }
}
```

```
NUMBERS: 2 4 7 10 11 32 45 87
Enter a value: 10
Found 10 at index 3.
...
NUMBERS: 2 4 7 10 11 32 45 87
Enter a value: 17
17 was not found.
```

PARTICIPATION ACTIVITY 2.2.4: Binary search algorithm execution.

Given array: [4, 11, 17, 18, 25, 45, 63, 77, 89, 114].

1) How many array elements will be checked to find the value 77 using binary search?

Check Show answer

2) How many array elements will be checked to find the value 17 using binary search?

Check Show answer

3) Binary search compares ____ array elements against the key when searching for key 2.

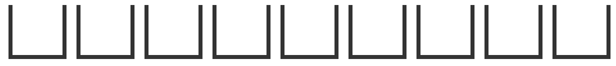
Check Show answer

Binary search efficiency

Binary search is incredibly efficient in finding an element within a sorted array. During each iteration or step of the algorithm, binary search reduces the search space (i.e., the remaining elements to search within) by about half. The search terminates when the element is found or the search space is empty (element not found). For an N element array, the maximum number of steps required to reduce the search space to an empty subarray is $\lfloor \log_2 N \rfloor + 1$.

PARTICIPATION ACTIVITY 2.2.5: Speed of linear search versus binary search.

Search key: 32 ☐ Unsearched ☐ Searched ☐ Current



Animation content:

Static figure:
The label "Search key: 32" is shown.
An array labeled "Linear search" containing 9 elements is shown. The array elements are, 2, 4, 7, 10, 11, 32, 45, 87, 99 consecutively. Index labels are shown under the array elements from 2 to 32. The index labels are 1, 2, 3, 4, 5, 6 consecutively. Array elements 2, 4, 7, 10 and 11 have a gray background indicating that the elements have been searched. The array element 32 has a light blue background indicating that 32 is the current element being looked at. A green checkmark is shown above the element 32.
An array labeled "Binary search" containing 9 elements is shown. The array elements are, 2, 4, 7, 10, 11, 32, 45, 87, 99 consecutively. Array elements 11 and 45 have a gray background indicating that the elements have been searched. The array element 32 has a light blue background indicating that 32 is the current element being looked at. The index number 1 is shown under element 11. The index number 3 is shown under element 32. The index number 3 is shown under element 45. A green checkmark is shown above the element 32.

Step 1: Binary search begins with the middle element. Each subsequent search reduces the search space by about half. Binary search finds a match after 3 comparisons.
Same array is shown for linear and binary search: [2, 4, 7, 10, 11, 32, 45, 87, 99]. First three comparisons are highlighted: 2, 4, and 7 for linear search, 11, 45, and 32 for binary search.

Step 2: Using linear search, a match is found after 6 comparisons. The binary search is more efficient, requiring only 3 comparisons.
Additional elements are highlighted for linear search: 10, 11, 32.

Animation captions:

1. Binary search begins with the middle element. Each subsequent search reduces the search space by about half. Binary search finds a match after 3 comparisons.

2. Using linear search, a match is found after 6 comparisons. The binary search is more efficient, requiring only 3 comparisons.

PARTICIPATION ACTIVITY 2.2.6: Linear and binary search efficiency.

For each question, assume that a search occurs on an array with 1024 elements. For questions 2 and 3, assume the array is sorted.

1) A linear search occurs. How many distinct array elements are compared against a search key that is greater than all elements in the array?

`elements`

Check **Show answer**

2) A binary search occurs. How many distinct array elements are compared against a search key that is *greater* than all elements in the array?

`elements`

Check **Show answer**

3) A binary search occurs. How many distinct array elements are compared against a search key that is *less* than all elements in the array?

`elements`

Check **Show answer**

CHALLENGE ACTIVITY 2.2.1: Binary search.

704586.1152092.qx3zqy7

Start

A surnames array is searched for Love using binary search.
Surnames array: [Ball, Cruz, Diaz, Hall, Lee, Love, Pena, Reed, Webb, West]

What is the first surname searched?

Pick

What is the second surname searched?

Pick

1	2	3	4	5

Check Next

View solution (Instructors only)

2.3 C++: Linear and binary search

Linear search

Search algorithms are used to find the location (index) of some key element in an array, or otherwise indicate that the key is not in the array. ***Linear search*** is a search algorithm that starts from the beginning of an array, and checks each element until the search key is found or the end of the array is reached.

The LinearSearch() function shown in the figure below compares each item in a given array, one at a time. If the key value is found, the index is returned. If the key value is not found, -1 is returned.

Figure 2.3.1: Linear search algorithm.

```
#include <iostream>
using namespace std;

int LinearSearch(int* numbers, int numbersSize, int key) {
   for (int i = 0; i < numbersSize; i++) {
      if (numbers[i] == key) {
         return i;
      }
   }
   return -1; // not found
}

int main() {
   int numbers[] = { 2, 4, 7, 10, 11, 32, 45, 87 };
   int numbersSize = sizeof(numbers) / sizeof(numbers[0]);
   cout << "NUMBERS: [" << numbers[0];
   for (int i = 1; i < numbersSize; i++) {
      cout << ", " << numbers[i];
   }
   cout << "]" << endl;

   // Prompt for an integer to search for
   cout << "Enter an integer value: ";
   int key = 0;
   cin >> key;

   int keyIndex = LinearSearch(numbers, numbersSize, key);
   if (keyIndex == -1) {
      cout << key << " was not found." << endl;
   }
   else {
      cout << "Found " << key << " at index ";
      cout << keyIndex << "." << endl;
   }
}
```

```
NUMBERS: [2, 4, 7, 10, 11, 32, 45, 87]
Enter an integer value: 10
Found 10 at index 3.
...
NUMBERS: [2, 4, 7, 10, 11, 32, 45, 87]
Enter an integer value: 17
17 was not found.
```

PARTICIPATION ACTIVITY 2.3.1: Linear search algorithm.

How many comparisons are performed by LinearSearch() for each example?

1) Array: [12, 75, 18, 22, 94, 16, 22]
Key: 94

Check Show answer

2) Array: [12, 75, 18, 22, 94, 16, 22]
Key: 22

Check Show answer

3) Array: [12, 75, 18, 22, 94, 16, 22]
Key: 10

Check Show answer

Binary search

The binary search algorithm also finds the location of a key value in an array, but is much faster than linear search because the search is performed on a sorted array. Binary search compares the middle element of the array to the key. If the middle element matches the key, then the algorithm is complete and returns the index. If the key is smaller than the middle element, the loop proceeds by searching the first half of the array. If the key is larger than the middle element, the loop proceeds by searching the last half of the array. In this way the search field is always cut in half, leading to many fewer comparisons.

To calculate the index in the middle of low and high indices, the sum of high and low is divided by 2 using integer division.

Figure 2.3.2: Binary search algorithm.

```
#include <iostream>
using namespace std;

int BinarySearch(int* numbers, int numbersSize, int key) {
   // Variables to hold the low and high indices of the area being searched.
   // Starts with entire range.
   int low = 0;
   int high = numbersSize - 1;

   // Loop until "low" passes "high"
   while (high >= low) {
      // Calculate the middle index
      int mid = (high + low) / 2;

      // Cut the range to either the left or right half,
      // unless numbers[mid] is the key
      if (numbers[mid] < key) {
         low = mid + 1;
      }
      else if (numbers[mid] > key) {
         high = mid - 1;
      }
      else {
         return mid;
      }
   }

   return -1; // not found
}

int main() {
   int numbers[] = { 2, 4, 7, 10, 11, 32, 45, 87 };
   int numbersSize = sizeof(numbers) / sizeof(numbers[0]);
   cout << "NUMBERS: [" << numbers[0];
   for (int i = 1; i < numbersSize; i++) {
      cout << ", " << numbers[i];
   }
   cout << "]" << endl;

   // Prompt for an integer to search for
   cout << "Enter an integer value: ";
   int key = 0;
   cin >> key;

   int keyIndex = BinarySearch(numbers, numbersSize, key);
   if (keyIndex == -1) {
      cout << key << " was not found." << endl;
   }
   else {
      cout << "Found " << key << " at index ";
      cout << keyIndex << "." << endl;
   }
}
```

```
NUMBERS: [2, 4, 7, 10, 11, 32, 45, 87]
Enter an integer value: 10
Found 10 at index 3.
```

```
...
NUMBERS: [2, 4, 7, 10, 11, 32, 45, 87]
Enter an integer value: 17
17 was not found.
```

PARTICIPATION ACTIVITY 2.3.2: Binary search algorithm.

The array (12, 75, 18, 22, 94, 16, 22) can be used as input to the BinarySearch() function.

- True
- False

PARTICIPATION ACTIVITY 2.3.3: Binary search algorithm.

How many array items are compared to the key by BinarySearch() for each example?

1) Array: (12, 18, 22, 34, 41, 74, 88)
Key: 74

Check Show answer

2) Array: (12, 18, 22, 34, 41, 74, 88)
Key: 10

Check Show answer

zyDE 2.3.1: Linear and binary search algorithms.

Both linear and binary search algorithms appear in the IDE below. The functions are modified to count the total number of comparisons each algorithm performs. The main program calls both search algorithms and the total number of comparisons each algorithm performs is displayed as part of the output. Try different arrays and key values.

LinearAndBinarySearchDe...

Load default template...

```
#include <iostream>
using namespace std;

int LinearSearch(int* numbers, int numbersSize, int key, int& comparisons) {
   // Added parameter to hold total number of comparisons.
   comparisons = 0;

   for (int i = 0; i < numbersSize; i++) {
      comparisons++;
      if (numbers[i] == key) {
         return i;
      }
   }
   return -1; // not found
}

int BinarySearch(int* numbers, int numbersSize, int key, int& comparisons) {
   // Added parameter to hold total number of comparisons.
   comparisons = 0;

   // Variables to hold the low and high indices of the area being searched.
   // Starts with entire range.
   int low = 0;
   int high = numbersSize - 1;

   // Loop until "low" passes "high"
   while (high >= low) {
      // Calculate the middle index
      int mid = (high + low) / 2;

      // Cut the range to either the left or right half,
      // unless numbers[mid] is the key
      comparisons++;
      if (numbers[mid] < key) {
         low = mid + 1;
      }
      else if (numbers[mid] > key) {
         high = mid - 1;
      }
      else {
         return mid;
      }
   }

   return -1; // not found
}

// Main program to test both search functions
int main() {
   int numbers[] = { 2, 4, 7, 10, 11, 32, 45, 87 };
   int numbersSize = sizeof(numbers) / sizeof(numbers[0]);
   cout << "NUMBERS: [" << numbers[0];
   for (int i = 1; i < numbersSize; i++) {
      cout << ", " << numbers[i];
   }
   cout << "]" << endl;

   // Prompt for an integer to search for
   cout << "Enter an integer value: ";
   int key = 0;
   cin >> key;
   cout << endl;

   int comparisons = 0;
   int keyIndex = LinearSearch(numbers, numbersSize, key, comparisons);
   if (keyIndex == -1) {
      cout << "Linear search: " << key << " was not found with ";
      cout << comparisons << " comparisons." << endl;
   }
   else {
      cout << "Linear search: Found " << key << " at index " << keyIndex;
      cout << " with " << comparisons << " comparisons." << endl;
   }

   keyIndex = BinarySearch(numbers, numbersSize, key, comparisons);
   if (keyIndex == -1) {
      cout << "Binary search: " << key << " was not found with ";
      cout << comparisons << " comparisons." << endl;
   }
   else {
      cout << "Binary search: Found " << key << " at index " << keyIndex;
      cout << " with " << comparisons << " comparisons." << endl;
   }
}
```

```
45
```

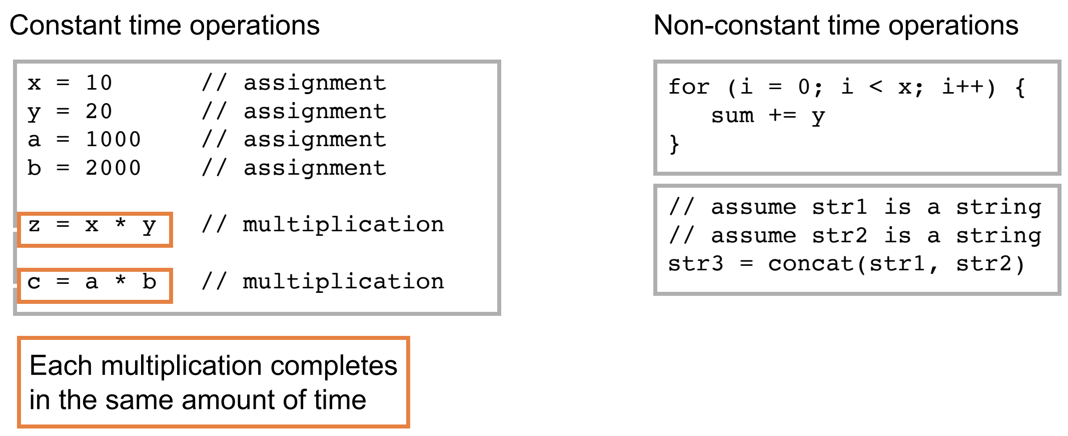
Run

2.4 Constant time operations

Constant time operations

In practice, designing an efficient algorithm aims to lower the amount of time that an algorithm runs. However, a single algorithm can always execute more quickly on a faster processor. Therefore, the theoretical analysis of an algorithm describes runtime in terms of number of constant time operations, not nanoseconds. A ***constant time operation*** is an operation that, for a given processor, always operates in the same amount of time, regardless of input values.

PARTICIPATION ACTIVITY 2.4.1: Constant time vs. non-constant time operations.


Constant time operations
Non-constant time operations
Each multiplication completes in the same amount of time

```
x = 10      // assignment
y = 20      // assignment
a = 1000    // assignment
b = 2000    // assignment

z = x * y   // multiplication

c = a * b   // multiplication
```

```
for (i = 0; i < x; i++) {
   sum += y
}
```

```
// assume str1 is a string
// assume str2 is a string
str3 = concat(str1, str2)
```

Animation content:

Static figure: A code block labeled Constant time operations and two code blocks labeled Non-constant time operations are displayed.
Constant time operations:
Begin pseudocode:
x = 10 // assignment
y = 20 // assignment
a = 1000 // assignment

```
b = 2000   // assignment

z = x * y   // multiplication

c = a * b   // multiplication
End pseudocode.
```

First non-constant time operations code block:

```
Begin pseudocode:
for (i = 0; i < x; i++) {
  sum += y
}
End pseudocode.
```

Second non-constant time operations code block:

```
Begin pseudocode:
// assume str1 is a string
// assume str2 is a string
str3 = concat(str1, str2)
End pseudocode.
```

Step 1: Statements x = 10, y = 20, a = 1000, and b = 2000 assign values to fixed-size integer variables. Each assignment is a constant time operation.
Step 2: A CPU multiplies values 10 and 20 at the same speed as 1000 and 2000. Multiplication of fixed-size integers is a constant time operation. The lines of code, z = x * y, c = a * b, are highlighted and the text, each multiplication completes in the same amount of time, is displayed.
Step 3: A loop that iterates x times, adding y to a sum each iteration, will take longer if x is larger. The loop is not constant time.
Step 4: String concatenation is another common operation that is not constant time, because more characters must be copied for larger strings.

Animation captions:

1. Statements x = 10, y = 20, a = 1000, and b = 2000 assign values to fixed-size integer variables. Each assignment is a constant time operation.
2. A CPU multiplies values 10 and 20 at the same speed as 1000 and 2000. Multiplication of fixed-size integers is a constant time operation.
3. A loop that iterates x times, adding y to a sum each iteration, will take longer if x is larger. The loop is not constant time.
4. String concatenation is another common operation that is not constant time, because more characters must be copied for larger strings.

PARTICIPATION ACTIVITY 2.4.2: Constant time operations.

1) The statement below that assigns x with y is a constant time operation.

```
y = 10
x = y
```

- True
- False

2) A loop is never a constant time operation.

- True
- False

3) The three constant time operations in the code below can collectively be considered one constant time operation.

```
x = 26.5
y = 15.5
z = x + y
```

- True
- False

Identifying constant time operations

The programming language being used, as well as the hardware running the code, both affect what is and what is not a constant time operation. Ex: Most modern processors perform arithmetic operations on integers and floating point values at a fixed rate that is unaffected by operand values. Part of the reason for this is that the floating point and integer values have a fixed size. The table below summarizes operations that are generally considered constant time operations.

Table 2.4.1: Common constant time operations.

Operation	Example

Addition, subtraction, multiplication, and division of fixed size integer or floating point values.	`w = 10.4`
	`x = 3.4`
	`y = 2.0`
	`z = (w - x) / y`

Assignment of a reference, pointer, or other fixed size data value.	`x = 1000`
	`y = x`
	`a = true`
	`b = a`

Comparison of two fixed size data values.	`a = 100`
	`b = 200`
	`if (b > a) {`
	`   ...`
	`}`

Read or write an array element at a particular index.	`x = arr[index]`
	`arr[index + 1] = x + 1`

PARTICIPATION ACTIVITY 2.4.3: Identifying constant time operations.

1) In the code below, suppose str1 is a pointer or reference to a string. The code only executes in constant time if the assignment copies the pointer/reference, and not all the characters in the string.

```
str2 = str1
```

- True
- False

2) Certain hardware may execute division more slowly than multiplication, but both may still be constant time operations.

- True
- False

3) The hardware running the code is the

only thing that affects what is and what is not a constant time operation.

- True
- False

2.5 Growth of functions and complexity

Upper and lower bounds

An algorithm with runtime complexity T(N) has a lower bound and an upper bound.

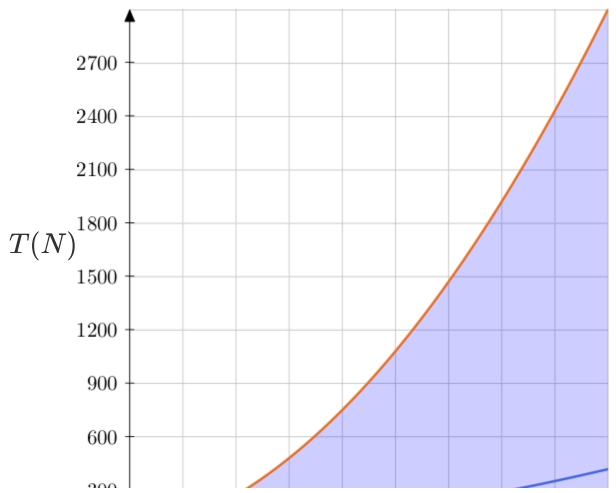
- ***Lower bound***: A function f(N) that is ≤ the best case T(N), for all values of N ≥ N_0.
- ***Upper bound***: A function f(N) that is ≥ the worst case T(N), for all values of N ≥ N_0.

Since N is an algorithm's input data size, N_0 is a nonnegative integer. In this material N_0 is usually 1, thus making an upper or lower bound hold for any non-empty input data size.

Given a function T(N), an infinite number of lower bounds and upper bounds exist. Ex: If an algorithm's best case runtime is T(N) = 5N + 4, then subtracting any nonnegative integer yields a lower bound that holds for all N: 5N + 3, 5N + 2, and so on. So two additional criteria are commonly used to choose a preferred upper or lower bound. The preferred bound:

1. is a single-term polynomial and
2. bounds T(N) as tightly as possible.

PARTICIPATION ACTIVITY 2.5.1: Upper and lower bounds with N_0 = 1.



Algorithm runtime complexities

$3N^2 + 10N + 17$	(worst case)
$2N^2 + 5N + 5$	(best case)

Algorithm runtime bounds

$2N^2$	(lower)
$30N^2$	(upper)

Animation content:

Static figure:
A graph is shown with N as the independent x-axis and T(N) as the dependent y-axis. Window is from 1 to 9 on the x-axis and from 0 to 3000 on the y-axis. 4 lines are shown on the graph representing the functions T(N) = 3N^2 + 10N + 17 (worst case), T(N) = 2N^2 + 5N + 5 (best case), 2N^2 (lower), and 30 N^2 (upper). The first two formulas are shown under the title "Algorithm runtime complexities". The second two formulas are shown under the title "Algorithm runtime bounds".

Step 1: A graph is shown with N as the independent x-axis and T(N) as the dependent y-axis. Window is from 1 to 9 on the x-axis and from 0 to 3000 on the y-axis. Functions T(N) = 3N^2 + 10N + 17 (worst case) and T(N) = 2N^2 + 5N + 5 (best case) are graphed and the formulas are revealed under the title "Algorithm runtime complexities".

Step 2: The label "Three-term polynomial" appears beneath the best case T(N). Three arrows appear to point out the polynomial's three terms: 2N^2, 5N, and 5.

Step 3: The lower bound, 2N^2, is graphed. The lower bound is below both T(N)s for all N values in the window.

Step 4: Text appears: "Upper bound: CN^2 ≥ 3N^2 + 10N + 17 for all N ≥ 1".

Step 5: Text appears, describing that if N = 1 then 3N^2 = 3 and 3N^2 + 10N + 17 = 30. Since 3 is not ≥ 30, 3N^2 does not work as an upper bound.

Animation captions:

1. An algorithm's worst and best case runtimes are represented by the blue and purple curves, respectively.
2. The best case expression itself is a lower bound, but is a polynomial with three terms: $2N^2$, $5N$, and 5. A single-term polynomial would provide a simpler picture.
3. $2N^2$, shown in yellow, is a lower bound. The lower bound is less than or equal to the best case T(N) for all N ≥ 1.
4. The worst case's highest power of N is N^2. So the upper bound must be some constant C times N^2 such that $CN^2 \geq 3N^2 + 10N + 17$ for all N ≥ 1.

5. $3N^2$ does not work. Ex: When $N = 1$, $3N^2 = 3$ and $3N^2 + 10N + 17 = 30$.
6. The lowest C that satisfies requirements is 30. $30N^2$ is greater than or equal to the worst case T(N) for all N ≥ 1. So $30N^2$, shown in orange, is an upper bound.
7. Together, the upper and lower bounds enclose all possible runtimes for this algorithm.

PARTICIPATION ACTIVITY 2.5.2: Upper and lower bounds with N_0 = 1.

Suppose an algorithm's best case runtime complexity is $T(N) = 3N + 6$, and the algorithm's worst case runtime is $T(N) = 5N^2 + 7N$.

1) The algorithm has _____.

- only one possible lower bound
- multiple, but finite, lower bounds
- an infinite number of lower bounds

2) Which is the preferred lower bound?

- $3N + 6$
- $3N$
- N

3) $7N^2$ is _____ for the algorithm.

- an upper bound
- a lower bound
- neither a lower bound nor an upper bound

4) Which function is an upper bound for the algorithm?

- $12N^2$
- $5N^2$
- $7N$

Growth rates and asymptotic notations

An additional simplification can factor out the constant from a bounding function, leaving a function that categorizes the algorithm's growth rate. Ex: Instead of saying that an algorithm's runtime function has an upper bound of $30N^2$, the algorithm could be described as having a worst case growth rate of N^2. ***Asymptotic notation*** is the classification of runtime complexity that uses functions that indicate only the growth rate of a bounding function. Three asymptotic notations are commonly used in complexity analysis:

- ***O notation*** (***Big O notation***) provides a growth rate for an algorithm's upper bound.
- ***Ω notation*** (***Big Omega notation***) provides a growth rate for an algorithm's lower bound.
- ***Θ notation*** (***Big Theta notation***) provides a growth rate that is both an upper and lower bound.

Table 2.5.1: Notations for algorithm complexity analysis.

Notation	General form	Meaning
O	$T(N) = O(f(N))$	A positive constant c exists such that, for all N ≥ N_0, $T(N) \leq c * f(N)$.
Ω	$T(N) = \Omega(f(N))$	A positive constant c exists such that, for all N ≥ N_0, $T(N) \geq c * f(N)$.
Θ	$T(N) = \Theta(f(N))$	$T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$.

PARTICIPATION ACTIVITY 2.5.3: Asymptotic notations.

$T(N) = 2N^2 + N + 9$ and $N_0 = 1$.

1) $T(N) = O(N^2)$

- True
- False

2) $T(N) = \Omega(N^2)$

- True
- False

3) $T(N) = \Theta(N^2)$

- True
- False

4) $T(N) = O(N^3)$

- True
- False

5) $T(N) = \Omega(N^3)$

- True
- False

PARTICIPATION ACTIVITY | 2.5.4: Exploring N_0 with logarithms.

$T(N) = \log(N) + 42$.

Note: In this book $\log(N)$ means $\log_2(N)$.

1) If N_0 = 0, $T(N) = O(\log N)$.

- True
- False

2) If N_0 = 1, $T(N) = O(\log N)$.

- True
- False

3) If N_0 = 2, $T(N) = O(\log N)$.

- True
- False

2.6 O notation

Big O notation

Big O notation is a mathematical way of describing how a function (running time of an algorithm) generally behaves in relation to the input size. In Big O notation, all functions that have the same growth rate (as determined by the highest order term of the function) are characterized using the same Big O notation. In essence, all functions that have the same growth rate are considered equivalent in Big O notation.

Given a function that describes the running time of an algorithm, the Big O notation for that function can be determined using the following rules:

1. If f(N) is a sum of several terms, the highest order term (the one with the fastest growth rate) is kept and others are discarded.
2. If f(N) has a term that is a product of several factors, all constants (those that are not in terms of N) are omitted.

PARTICIPATION ACTIVITY 2.6.1: Determining Big O notation of a function.

Algorithm steps: $5 + 13 \cdot N + 7 \cdot N^2$

Big O notation: $O(5 + 13 \cdot N + 7 \cdot N^2) = O(7 \cdot N^2)$

$= O(N^2)$ Pronounced: "Oh N squared"

Rule 1: If f(N) is a sum of several terms, the highest order term is kept and others are discarded.

Rule 2: If f(N) has a term that is a product of several factors, all constants are omitted.

Animation content:

Static figure: The following text is displayed:
Algorithm steps: 5 + 13·N + 7·N sup 2
Big O notation: O(5 + 13·N + 7·N sup 2)

Step 1: Determine a function that describes the running time of the algorithm, and then compute the Big O notation of that function.
Step 2: Apply rules to obtain the Big O notation of the function. The text, Rule 1: If f(N) is a sum of several terms, the highest order term
is kept and others are discarded., is displayed. The text, 7·N sup 2, is highlighted and Big O notation: now shows O(5 + 13·N + 7·N sup 2) = O(7·N sup 2). The text, Rule 2: If f(N) has a term

that is a product of several factors, all constants are omitted., is displayed and the equation, = O(N sup 2), appears under = O(7·N sup 2).
Step 3: All functions with the same growth rate are considered equivalent in Big O notation.

Animation captions:

1. Determine a function that describes the running time of the algorithm, and then compute the Big O notation of that function.
2. Apply rules to obtain the Big O notation of the function.
3. All functions with the same growth rate are considered equivalent in Big O notation.

PARTICIPATION ACTIVITY 2.6.2: Big O notation.

1) Which of the following Big O notations is equivalent to O(N + 9999)?

- O(1)
- O(N)
- O(9999)

2) Which of the following Big O notations is equivalent to O(734·N)?

- O(N)
- O(734)
- O(734·N^2)

3) Which of the following Big O notations is equivalent to O(12·N + 6·N^3 + 1000)?

- O(1000)
- O(N)
- O(N^3)

Big O notation of composite functions

The rules in the table below are used to determine the Big O notation of composite functions, where c denotes a constant.

Table 2.6.1: Rules for determining Big O notation of composite functions.

Composite function	Big O notation
$c \cdot O(f(N))$	$O(f(N))$
$c + O(f(N))$	$O(f(N))$
$g(N) \cdot O(f(N))$	$O(g(N) \cdot f(N))$
$g(N) + O(f(N))$	$O(g(N) + f(N))$

PARTICIPATION ACTIVITY 2.6.3: Big O notation for composite functions.

Determine the simplified Big O notation.

1) $10 \cdot O(N^2)$

- ○ $O(10)$
- ○ $O(N^2)$
- ○ $O(10 \cdot N^2)$

2) $10 + O(N^2)$

- ○ $O(10)$
- ○ $O(N^2)$
- ○ $O(10 + N^2)$

3) $3 \cdot N \cdot O(N^2)$

- ○ $O(N^2)$
- ○ $O(3 \cdot N^2)$
- ○ $O(N^3)$

4) $2 \cdot N^3 + O(N^2)$

- ○ $O(N^2)$
- ○ $O(N^3)$
- ○ $O(N^2 + N^3)$

5) $5 \cdot log_{10} N$

- ○ $O(5 \cdot log N)$
- ○ $O(log_{10} N)$
- ○ $O(log\, N)$

Runtime growth rate

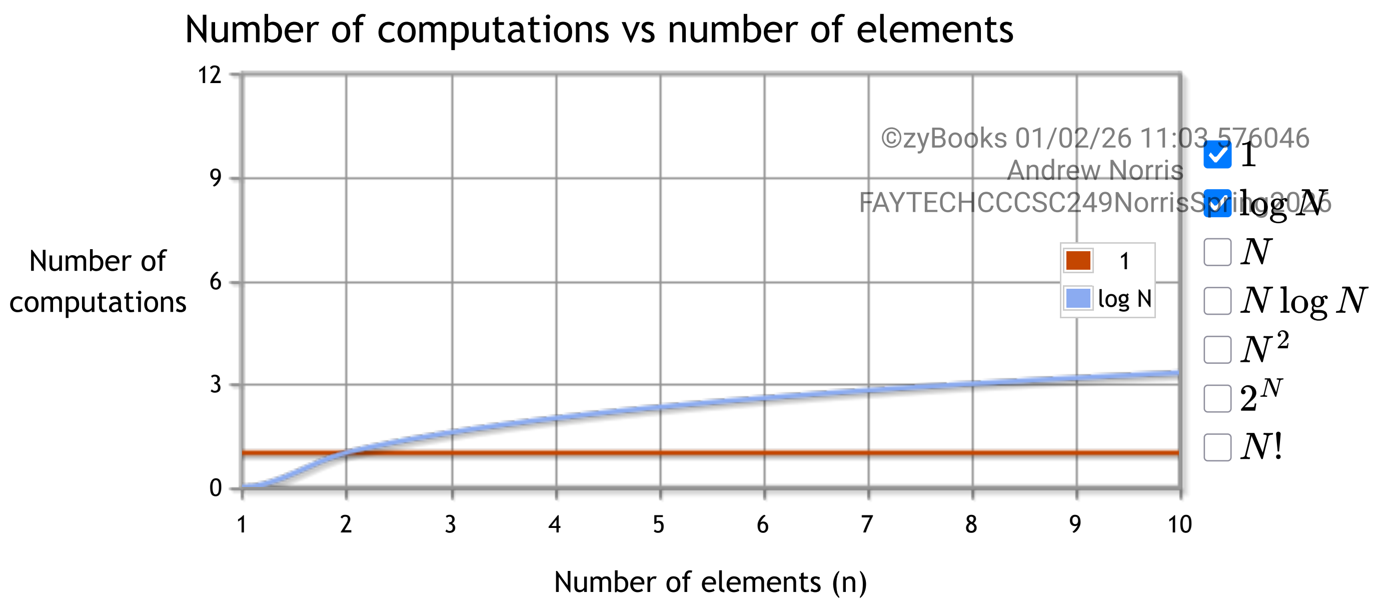
One consideration in evaluating algorithms is that the efficiency of the algorithm is most critical for large input sizes. Small inputs are likely to result in fast running times because N is small, so efficiency is less of a concern. The table below shows the runtime to perform f(N) instructions for different functions f and different values of N. For large N, the difference in computation time varies greatly with the rate of growth of the function f. The data assumes that a single instruction takes 1 μs to execute.

Table 2.6.2: Growth rates for different input sizes.

Function	N = 10	N = 50	N = 100	N = 1,000	N = 10,000	N = 100,000
$\log_2 N$	3.3 μs	5.65 μs	6.6 μs	9.9 μs	13.3 μs	16.6 μs
N	10 μs	50 μs	100 μs	1,000 μs	10 ms	100 ms
$N \log_2 N$	0.03 ms	0.28 ms	0.66 ms	0.0099 s	0.132 s	1.66 s
N^2	0.1 ms	2.5 ms	10 ms	1 s	100 s	2.7 hours
N^3	1 ms	0.125 s	1 s	16.7 min	11.57 days	31.7 years
2^N	0.001 s	35.7 years	> 1,000 years			

The interactive tool below illustrates graphically the growth rate of commonly encountered functions.

PARTICIPATION ACTIVITY 2.6.4: Computational complexity graphing tool.



Common Big O complexities

Many commonly used algorithms have running time functions that belong to one of a handful of growth functions. These common Big O notations are summarized in the following table. The table shows the Big O notation, the common word used to describe algorithms that belong to that notation, and an example with source code. Clearly, the best algorithm is one that has constant time complexity. Unfortunately, not all problems can be solved using constant complexity algorithms. In many cases, computer scientists have proven that certain types of problems can only be solved using quadratic or exponential algorithms.

Table 2.6.3: Runtime complexities for various code examples.

Notation	Name	Example pseudocode

$O(1)$	Constant	`GetMin(x, y) {`
		`   if (x < y) {`
		`      return x`
		`   }`
		`   else {`
		`      return y`
		`   }`
		`}`

$O(\log N)$	Logarithmic	`BinarySearch(numbers, N, key) {`
		`   mid = 0`
		`   low = 0`
		`   high = N - 1`
		
		`   while (high >= low) {`
		`      mid = (high + low) / 2`
		`      if (numbers[mid] < key) {`
		`         low = mid + 1`
		`      }`
		`      else if (numbers[mid] > key) {`
		`         high = mid - 1`
		`      }`
		`      else {`
		`         return mid`
		`      }`
		`   }`
		
		`   return -1   // not found`
		`}`

$O(N)$	Linear	`LinearSearch(numbers, numbersSize, key) {`
		`  for (i = 0; i < numbersSize; ++i) {`
		`      if (numbers[i] == key) {`
		`         return i`
		`      }`
		`   }`
		
		`   return -1 // not found`
		`}`

$O(N \log N)$	Linearithmic	`MergeSort(numbers, i, k) {`
		`   j = 0`
		`   if (i < k) {`
		`      j = (i + k) / 2              // Find midpoint`
		
		`      MergeSort(numbers, i, j)     // Sort left part`
		`      MergeSort(numbers, j + 1, k) // Sort right part`
		`      Merge(numbers, i, j, k)      // Merge parts`
		`   }`
		`}`

		`SelectionSort(numbers, numbersSize) {`

$O(c^N)$	Quadratic	`for (i = 0; i < numbersSize; ++i) { indexSmallest = i for (j = i + 1; j < numbersSize; ++j) { if (numbers[j] < numbers[indexSmallest]) { indexSmallest = j } } temp = numbers[i] numbers[i] = numbers[indexSmallest] numbers[indexSmallest] = temp } }`
	Exponential	`FibonacciNumber(n) { if (n == 0) { return 0 } else if (n == 1) { return 1 } else { return FibonacciNumber(n - 1) + FibonacciNumber(n - 2) } }`

PARTICIPATION ACTIVITY 2.6.5: Big O notation and growth rates.

1) O(5) has a _____ runtime complexity.

- constant
- linear
- exponential

2) O(N log N) has a _____ runtime complexity.

- constant
- linearithmic
- logarithmic

3) O(N + N^2) has a _____ runtime complexity.

- linear-quadratic

- exponential
- quadratic

4) A linear search has a _____ runtime complexity.

- O(log N)
- O(N)
- O(N^2)

5) A selection sort has a _____ runtime complexity.

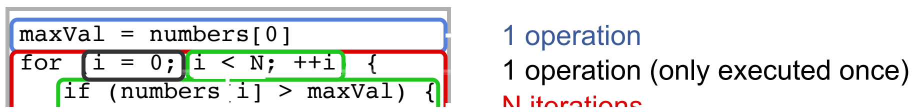
- O(N)
- O(N log N)
- O(N^2)

2.7 Algorithm analysis

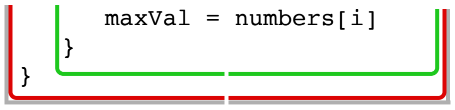
Worst-case algorithm analysis

To analyze how an algorithm's runtime scales as the input size increases, one first determines how many operations the algorithm executes for a specific input size, N. Then, the Big O notation for that function is determined. Algorithm runtime analysis often focuses on the worst-case runtime complexity. The ***worst-case runtime*** of an algorithm is the runtime complexity for an input that results in the longest execution. Other runtime analyses include best-case runtime and average-case runtime. Determining the average-case runtime requires knowledge of the statistical properties of the expected data inputs.

PARTICIPATION ACTIVITY 2.7.1: Runtime analysis: Finding the max value.



```
maxVal = numbers[0]
for (i = 0; i < N; ++i) {
   if (numbers[i] > maxVal) {
```


```
            maxVal = numbers[i]
        }
    }
```

Animation content:

Static figure:
Begin pseudo code:

```
maxVal = numbers[0]
for (i = 0; i < N; ++i) {
  if (numbers[i] > maxVal) {
    maxVal = numbers[i]
  }
}
```

End pseudo code.
The pseudo code block is shown with different parts highlighted by various boxes. The first line of code, "maxVal = numbers[0] " is highlighted in a blue box labeled "1 operation". Entire for loop, "for (i = 0; I < N; ++i) {
  if (numbers[i] > maxVal) {
    maxVal = numbers[i]
  }
}"
is highlighted in a red box labeled "N iterations". The For loop's "i = 0;" statement is highlighted in a black box labeled "1 operation (only executed once)". The For loop's other two statements, "i < N;" and "++i" are highlighted in a green box. The For loop’s "i < N;" statement also has a purple underline with corresponding purple label "1 final comparison operation". The For loop's body, "if (numbers[i] > maxVal) {
    maxVal = numbers[i]
  }" is also highlighted in a green box, labeled "2 operations" for the first two statements and "2 operations (worst-case)" for the body's statements.

Text labels below code block shows the time complexity computation:

$f(N) = 1 + 1 + 1 + N(2 + 2)$
$= 3 + 4N$
$= O(N)$

Step 1: Runtime analysis determines the total number of operations. Operations include assignment, addition, comparison, etc.
The pseudo code block appears.
Begin pseudo code:
```
maxVal = numbers[0]
for (i = 0; i < N; ++i) {
  if (numbers[i] > maxVal) {
    maxVal = numbers[i]
  }
}
```
End pseudo code.
The first line of code, "maxVal = numbers[0] " is highlighted in a blue box labeled "1 operation".

Step 2: The for loop iterates N times, but the for loop's initial expression i = 0 is executed once.
Entire For loop, "for (i = 0; i < N; ++i) {
```
  if (numbers[i] > maxVal) {
    maxVal = numbers[i]
  }
}"
```
is highlighted in a red box labeled "N iterations". Next, the For loop's "i = 0;" statement is highlighted in a black box labeled "1 operation (only executed once)".

Step 3: For each loop iteration, the increment and comparison expressions are each executed once. In the worst case, the if's expression is true, resulting in 2 operations.
The For loop's other two statements, "i < N;" and "++i" are highlighted in a green box, "2 operations".
Next, the For loop's body, "if (numbers[i] > maxVal) {
```
    maxVal = numbers[i]
  }" is highlighted in a green box, labeled "2 operations (worst-case)".
```

Step 4: One additional comparison is made before the loop ends.
A purple underline with corresponding purple label "1 final comparison operation" is shown under the For loop’s "i < N;" statement.

Step 5: The function f(N) specifies the number of operations executed for input size N. The Big O notation for the function is the algorithm's worst-case runtime complexity.
Text labels are revealed below code block which shows the time complexity computation:
```
"f(N) = 1 + 1 + 1 + N (2 + 2 )
   = 3 + 4N
   = O(N) "
```

Animation captions:

1. Runtime analysis determines the total number of operations. Operations include assignment,

addition, comparison, etc.
2. The for loop iterates N times, but the for loop's initial expression i = 0 is executed once.
3. For each loop iteration, the increment and comparison expressions are each executed once. In the worst case, the if's expression is true, resulting in 2 operations.
4. One additional comparison is made before the loop ends.
5. The function f(N) specifies the number of operations executed for input size N. The Big O notation for the function is the algorithm's worst-case runtime complexity.

PARTICIPATION ACTIVITY 2.7.2: Worst-case runtime analysis.

1) Which function best represents the number of operations in the worst case?

```
i = 0
sum = 0
while (i < N) {
   sum = sum + numbers[i]
   ++i
}
```

- ○ f(N) = 3N + 2
- ○ f(N) = 3N + 3
- ○ f(N) = 2 + N (N + 1)

2) What is the simplest Big O notation for the worst-case runtime?

```
negCount = 0
for(i = 0; i < N; ++i) {
   if (numbers[i] < 0) {
      ++negCount
   }
}
```

- ○ f(N) = 2 + 4N + 1
- ○ O(4N + 3)
- ○ O(N)

3) What is the Big O notation for the worst-case runtime?

```
for (i = 0; i < N; ++i) {
   if ((i % 2) == 0) {
      outVal[i] = inVals[i] * i
   }
}
```

- O(1)
- O(log N)
- O(N)

4) Assuming nVal is an integer, what is the best Big O notation for the worst-case runtime?

```
nVal = N
steps = 0
while (nVal > 0) {
   nVal = nVal / 2
   steps = steps + 1
}
```

- O(1)
- O(log N)
- O(N)

5) What is the Big O notation for the *best-case* runtime?

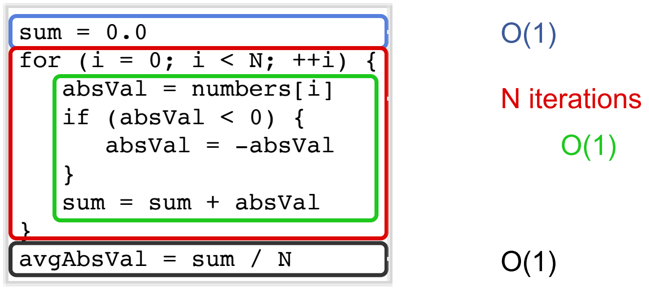
```
i = 0
belowThresholdSum = 0.0
belowThresholdCount = 0
while (i < N && numbers[i] <=
threshold) {
   belowThresholdCount += 1
   belowThresholdSum +=
numbers[i]
   i += 1
}
avgBelow = belowThresholdSum /
belowThresholdCount
```

- O(1)
- O(N)

Counting constant time operations

For algorithm analysis, the definition of a single operation does not need to be precise. An operation can be any statement (or constant number of statements) that has a constant runtime complexity, O(1). Since constants are omitted in Big O notation, any constant number of constant time operations is O(1). So, precisely counting the number of constant time operations in a finite sequence is not needed. Ex: An algorithm with a single loop that executes 5 operations before the loop, 3 operations each loop iteration, and 6 operations after the loop has a runtime of f(N) = 5 + 3N + 6, which can be written as O(1) + O(N) + O(1) = O(N). If 100 operations occur before the loop, the Big O notation for those operations is still O(1).

PARTICIPATION ACTIVITY 2.7.3: Simplified runtime analysis: A constant number of constant time operations is O(1).



```
sum = 0.0
for (i = 0; i < N; ++i) {
   absVal = numbers[i]
   if (absVal < 0) {
      absVal = -absVal
   }
   sum = sum + absVal
}
avgAbsVal = sum / N
```

O(1) + NO(1) + O(1) = O(N)

Animation content:

Static figure: Blue box around first line of code with label "O(1)". Red box around entire loop with label "N iterations". Green box around loop's body with label "O(1)". Black box around last line of code with label "O(1)". Final statement beneath the code indicates the time complexity calculation: O(1) + N * O(1) + O(1) = O(N).

Animation captions:

1. Constants are omitted in Big O notation, so any constant number of constant time operations is O(1).
2. The for loop iterates N times. Big O complexity can be written as a composite function and simplified.

PARTICIPATION ACTIVITY 2.7.4: Constant time operations.

1) A for loop of the form `for (i = 0; i < N; ++i) {}` that does not have nested loops or function calls, and does not modify i in the loop has runtime complexity O(N).

○ True

○ False

2) The complexity of the code below is O(1).

```
if (timeHour < 6) {
   tollAmount = 1.55
}
else if (timeHour < 10) {
   tollAmount = 4.65
}
else if (timeHour < 18) {
   tollAmount = 2.35
}
else {
   tollAmount = 1.55
}
```

○ True

○ False

3) The complexity of the algorithm below is O(1).

```
for (i = 0; i < 24; ++i) {
   if (timeHour < 6) {
      tollSchedule[i] = 1.55
   }
   else if (timeHour < 10) {
      tollSchedule[i] = 4.65
   }
   else if (timeHour < 18) {
      tollSchedule[i] = 2.35
   }
   else {
      tollSchedule[i] = 1.55
   }
}
```

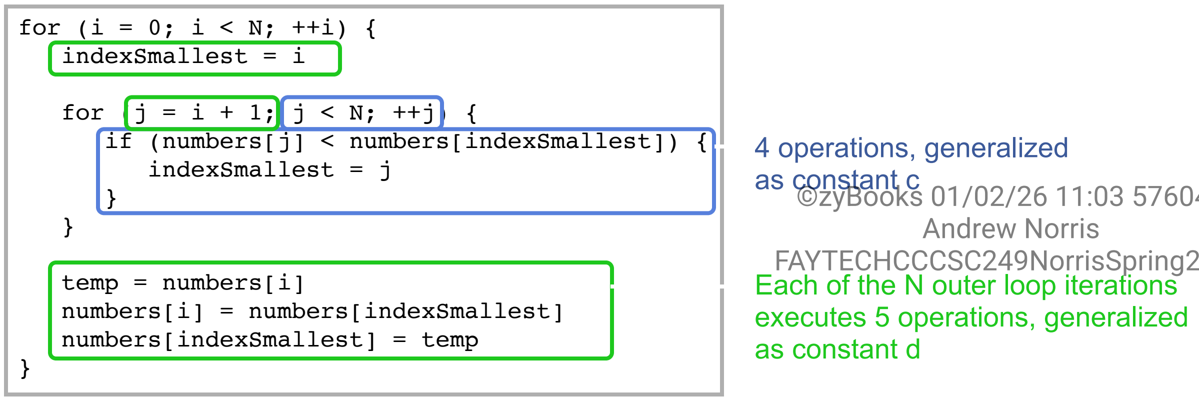
○ True

○ False

Runtime analysis of nested loops

Runtime analysis for nested loops requires summing the runtime of the inner loop over each outer loop iteration. The resulting summation can be simplified to determine the Big O notation.

PARTICIPATION ACTIVITY | 2.7.5: Runtime analysis of nested loop: Selection sort algorithm.



```
for (i = 0; i < N; ++i) {
   indexSmallest = i

   for (j = i + 1; j < N; ++j) {
      if (numbers[j] < numbers[indexSmallest]) {
         indexSmallest = j
      }
   }

   temp = numbers[i]
   numbers[i] = numbers[indexSmallest]
   numbers[indexSmallest] = temp
}
```

4 operations, generalized as constant c
Each of the N outer loop iterations executes 5 operations, generalized as constant d

$$f(N) = c\Big((N - 1) + (N - 2) + \cdots + 2 + 1 + 0\Big) = c\left(\frac{N(N-1)}{2}\right) + d \cdot N$$

$$O(f(N)) = O\left(\frac{c}{2}(N^2 - N) + d \cdot N\right) = O(N^2 + N) = O(N^2)$$

Animation content:

Static figure: Inside for loop's body, three parts are enclosed in three green boxes: indexSmallest = i, j = i + 1, and last three lines of loop body that perform the swap. Last green box's label is the "Each of the N outer..." label. Two statements in loop body enclosed in two blue boxes: j < N; ++j and entire if statement. If statement box's label is the "4 operations..." label.
Two formulas below code box, indicating computations for time complexity determination:
f(N) = c((N-1) + (N-2) + ... + 2 + 1 + 0) = c((N(N-1))/2) + d*N
O(f(N)) = O((c/2)(N^2 - N) + d*N) = O(N^2 + N) = O(N^2)

Animation captions:

1. For each iteration of the outer loop, the runtime of the inner loop is determined and added together to form a summation. For iteration i = 0, the inner loop executes N - 1 iterations.
2. For i = 1, the inner loop iterates N - 2 times: iterating from j = 2 to N - 1.
3. For i = N - 3, the inner loop iterates twice: iterating from j = N - 2 to N - 1. For i = N - 2, the inner loop iterates once: iterating from j = N - 1 to N - 1.

4. For i = N - 1, the inner loop iterates 0 times. The summation is the sum of a consecutive sequence of numbers from N - 1 to 0.
5. The sequence contains N / 2 pairs, each summing to N - 1, and can be simplified.
6. Each iteration of the loops requires a constant number of operations, which is defined as the constant c.
7. Additionally, each iteration of the outer loop requires a constant number of operations, which

is defined as the constant d.

8. Big O notation omits the constant values, and the runtime is equal to the summation of the total inner loop iterations.

Figure 2.7.1: Common summation: Summation of consecutive numbers.

$$(N-1)+(N-2)+\cdots+2+1=\frac{N(N-1)}{2}=O(N^2)$$

PARTICIPATION ACTIVITY 2.7.6: Nested loops.

Determine the Big O worst-case runtime for each algorithm.

1)
```
for (i = 0; i < N; i += 1)  {
   for (j = 0; j < N; j += 1) {
      if (numbers[i] <
numbers[j]) {
         eqPerms += 1
      }
      else {
         neqPerms += 1
      }
   }
}
```

○ O(N)

○ O(N^2)

2)
```
for (i = 0; i < N; i += 1)  {
   for (j = 0; j < (N - 1); j +=
1) {
      if (numbers[j + 1] <
numbers[j]) {
         temp = numbers[j]
         numbers[j] = numbers[j
+ 1]
         numbers[j + 1] = temp
      }
   }
}
```

○ O(N)

○ O(N^2)

3)
```
for (i = 0; i < N; i = i + 2) {
   for (j = 0; j < N; j = j + 2)
{
      cVals[i][j] = inVals[i] *
j
   }
}
```

- O(N)
- O(N^2)

4)
```
for (i = 0; i < N; i += 1)  {
   for (j = i; j < N - 1; j +=
1) {
      cVals[i][j] = inVals[i] *
j
   }
}
```

- O(N^2)
- O(N^3)

5)
```
for (i = 0; i < N; i += 1) {
   sum = 0
   for (j = 0; j < N; j += 1) {
      for (k = 0; k < N; k += 1)
{
         sum = sum + aVals[i][k]
* bVals[k][j]
      }

      cVals[i][j] = sum
   }
}
```

- O(N)
- O(N^2)
- O(N^3)

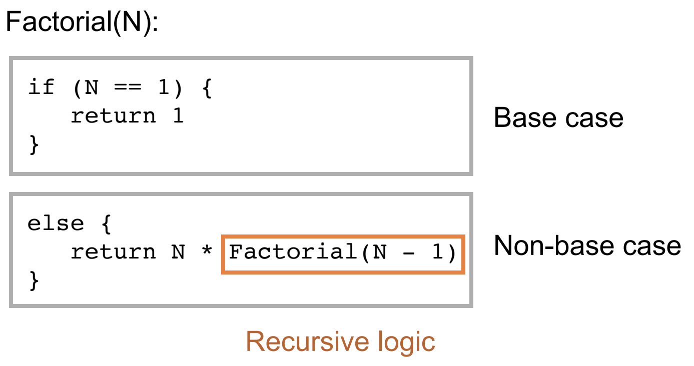
2.8 Recursive definitions

Recursive algorithms

An ***algorithm*** is a sequence of steps, including at least one terminating step, for solving a problem. A ***recursive algorithm*** is an algorithm that breaks the problem into smaller subproblems and applies the algorithm itself to solve the smaller subproblems.

Because a problem cannot be endlessly divided into smaller subproblems, a recursive algorithm must have a ***base case***: A case where a recursive algorithm completes without applying itself to a smaller subproblem.

PARTICIPATION ACTIVITY 2.8.1: Recursive factorial algorithm for positive numbers.



Factorial(N):

```
if (N == 1) {
   return 1
}
```

```
else {
   return N * Factorial(N - 1)
}
```

Animation content:

Static figure: A recursive algorithm labeled Factorial(N): containing two code blocks is displayed. The first code block is labeled Base case and the second code block is labeled Non-base case.
Base case:
Begin pseudocode:

```
if (N == 1) {
  return 1
}
```

End pseudocode.
Non-base case:
Begin pseudocode:

```
else {
  return N * Factorial(N - 1)
}
```

End pseudocode.

Step 1: A recursive algorithm to compute N factorial has two parts: the base case and the non-base case.
Step 2: N is assumed to be a positive integer. The base case is when N equals 1, wherein a result of 1 is returned.
Step 3: The non-base case computes the result by multiplying N by (N - 1) factorial.
Step 4: The algorithm applying itself to a smaller subproblem is what makes the algorithm recursive. The code, Factorial(N - 1), is highlighted and the text, Recursive logic, appears below the code.

Animation captions:

1. A recursive algorithm to compute N factorial has two parts: the base case and the non-base case.
2. N is assumed to be a positive integer. The base case is when N equals 1, wherein a result of 1 is returned.
3. The non-base case computes the result by multiplying N by (N - 1) factorial.
4. The algorithm applying itself to a smaller subproblem is what makes the algorithm recursive.

PARTICIPATION ACTIVITY 2.8.2: Recursive algorithms.

1) A recursive algorithm applies itself to a smaller subproblem in all cases.

- ○ True
- ○ False

2) The base case is what ensures that a recursive algorithm eventually terminates.

- ○ True
- ○ False

3) The presence of a base case is what identifies an algorithm as being recursive.

- ○ True
- ○ False

Recursive functions

A ***recursive function*** is a function that calls itself. Recursive functions are commonly used to implement recursive algorithms.

Table 2.8.1: Sample recursive functions: Factorial(), CumulativeSum(), and ReverseArray().

```
Factorial(N) {
   if (N == 1)
      return 1
   else
      return N * Factorial(N - 1)
}
```

```
CumulativeSum(N) {
   if (N == 0)
      return 0
   else
      return N + CumulativeSum(N - 1)
}
```

```
ReverseArray(array, startIndex, endIndex) {
   if (startIndex >= endIndex)
      return
   else {
      Swap array elements at startIndex and endIndex
      ReverseArray(list, startIndex + 1, endIndex - 1)
   }
}
```

PARTICIPATION ACTIVITY 2.8.3: CumulativeSum() recursive function.

1) What is the condition for the base case in the CumulativeSum() function?

- ○ N equals 0
- ○ N does not equal 0

2) If `Factorial(6)` is called, how many additional calls are made to Factorial() to compute the result of 720?

- ○ 7
- ○ 5
- ○ 3

3) Suppose ReverseArray() is called with an array of length 3, a start index of 0, and an out-of-bounds end index of 3. The base case ensures that the function

still properly reverses the array.

- True
- False

2.9 Recursive algorithms

Fibonacci numbers

The ***Fibonacci sequence*** is a numerical sequence where each term is the sum of the previous two terms in the sequence, except the first two terms, which are 0 and 1. A recursive function can be used to calculate a ***Fibonacci number***: A term in the Fibonacci sequence.

Figure 2.9.1: FibonacciNumber() recursive function.

```
FibonacciNumber(termIndex) {
   if (termIndex == 0) {
      return 0
   }
   else if (termIndex == 1) {
      return 1
   }
   else {
      return FibonacciNumber(termIndex - 1) + FibonacciNumber(termIndex - 2)
   }
}
```

PARTICIPATION ACTIVITY 2.9.1: FibonacciNumber recursive function.

1) What does `FibonacciNumber(2)` return?

Check **Show answer**

2) What does `FibonacciNumber(4)` return?

Check Show answer

3) What does `FibonacciNumber(8)` return?

Check Show answer

Recursive binary search

Binary search is an algorithm that searches a sorted array for a key by first comparing the key to the array's middle element. If a match occurs, the key's index is returned. Otherwise, half of the remaining array is recursively searched until the key is found or the remaining subarray is empty.

Figure 2.9.2: BinarySearch() recursive function.

```
BinarySearch(numbers, low, high, key) {
   if (low > high) {
      return -1
   }

   mid = (low + high) / 2
   if (numbers[mid] < key) {
      return BinarySearch(numbers, mid + 1, high, key)
   }
   else if (numbers[mid] > key) {
      return BinarySearch(numbers, low, mid - 1, key)
   }
   return mid
}
```

PARTICIPATION ACTIVITY 2.9.2: Recursive binary search.

Suppose `BinarySearch(numbers, 0, 6, 42)` is used to search the array [14, 26, 42, 59, 71, 88, 92] for key 42.

1) What is the first middle element that is compared against 42?

- 42
- 59

○ 71

2) What will the low and high argument values be for the first recursive call?

○ low = 0
high = 2

○ low = 0
high = 3

○ low = 4
high = 6

3) How many calls to BinarySearch() will be made by the time 42 is found?

○ 2

○ 3

○ 4

PARTICIPATION ACTIVITY 2.9.3: Recursive binary search base case.

Which does not describe a base case for BinarySearch()?

○ The low argument is greater than the high argument.

○ The element at index `mid` equals the key.

○ The element at index `mid` is less than the key.

2.10 Analyzing the time complexity of recursive algorithms

Recurrence relations

The runtime complexity T(N) of a recursive function will have function T on both sides of the equation.

Ex: Binary search performs constant time operations, then a recursive call that operates on half of the input, making the runtime complexity T(N) = O(1) + T(N / 2). Such a function is known as a ***recurrence relation***: A function f(N) that is defined in terms of the same function operating on a value < N.

Using O-notation to express runtime complexity of a recursive function requires solving the recurrence relation. For simpler recursive functions such as binary search, runtime complexity can be determined by expressing the number of function calls as a function of N.

PARTICIPATION ACTIVITY 2.10.1: Worst case binary search runtime complexity.

```
BinarySearch(numbers, low, high, key) {
   if (low > high) {
      return -1
   }
   mid = (low + high) / 2
   if (numbers[mid] < key) {
      return BinarySearch(numbers, mid + 1, high, key)
   }
   else if (numbers[mid] > key) {
      return BinarySearch(numbers, low, mid - 1, key)
   }
   return mid
}
```

Input size	Number of function calls
1	1
2	2
4	3
8	4
16	5
32	6
...	...
N	$\log_2 N + 1$

```
  If: T(N) = O(1) + T(N / 2)
Then: T(N) = O(log N)
```

Animation content:

Static figure: A code block and a table with two columns, left is "Input size" and right is "Number of function calls", are displayed.
Begin pseudocode:
BinarySearch(numbers, low, high, key) {
  if (low > high) {

```
    return -1
  }
  mid = (low + high) / 2
  if (numbers[mid] < key) {
    return BinarySearch(numbers, mid + 1, high, key)
  }
  else if (numbers[mid] > key) {
    return BinarySearch(numbers, low, mid - 1, key)
  }
  return mid
}
End pseudocode.
```

Table data:
1,1
2,2
4,3
8,4
16,5
32,6
...,...
N,log_2(N) + 1
End table data.

Below the table is the final conclusion about time complexity:
If: T(N) = O(1) + T(N/2)
Then: T(N) = O(log N)

Animation captions:

1. In the non-base case, BinarySearch() does some O(1) operations plus a recursive call on half the input array.
2. The maximum number of recursive calls can be computed for any known input size. For size 1, 1 recursive call is made.
3. Additional entries in the table can be filled. An array of size 32 is split in half 6 times before encountering the base case.
4. By analyzing the pattern, the total number of function calls can be expressed as a function of N.
5. The number of function calls corresponds to the runtime complexity.

PARTICIPATION ACTIVITY 2.10.2: Binary search and recurrence relations.

1) When the low and high arguments are equal, BinarySearch() has 0 items to search and so immediately returns -1.

   ○ True

   ○ False

2) Suppose BinarySearch() is used to search for a key within an array with 64 numbers. If the key is not found, how many recursive calls to BinarySearch() are made?

   ○ 1

   ○ 7

   ○ 64

3) Which function is a recurrence relation?

   ○ $T(N) = N^2 + 6N + 2$

   ○ $T(N) = 6N + T(N/4)$

   ○ $T(N) = \log_2 N$

Recursion trees

The runtime complexity of any recursive function can be split into two parts: operations done directly by the function and operations done by recursive calls made by the function. Ex: For binary search's T(N) = O(1) + T(N / 2), O(1) represents operations directly done by the function, and T(N / 2) represents operations done by a recursive call. A useful tool for solving recurrences is a ***recursion tree***: A visual diagram of an operation done by a recursive function that separates operations done directly by the function and operations done by recursive calls.

Recursion trees consist of nodes. A ***node*** is a structure that stores data and has links to other nodes. Nodes are commonly shown as an ellipse. The node's data is inside the ellipse, and lines extending from the ellipse indicate links to other nodes. For a recursion tree, the node's data represents the number of operations performed by a function, not including recursive calls. Links to other nodes represent recursive calls.

PARTICIPATION ACTIVITY 2.10.3: Recursion trees.

Animation content:

Static figure:
On the left, equation at top: T(N) = k + T(N / 2). Below the equation is a visual structure for the recursion tree. Atop the tree is the text "k" in a circle. The circle is known as a "node". A line below connects to another "k" node. A line below heads toward "..." text, then another line that connects to another "k" node at the bottom. Height bar appears to the side of the recursion tree indicating the height: log_2(N).

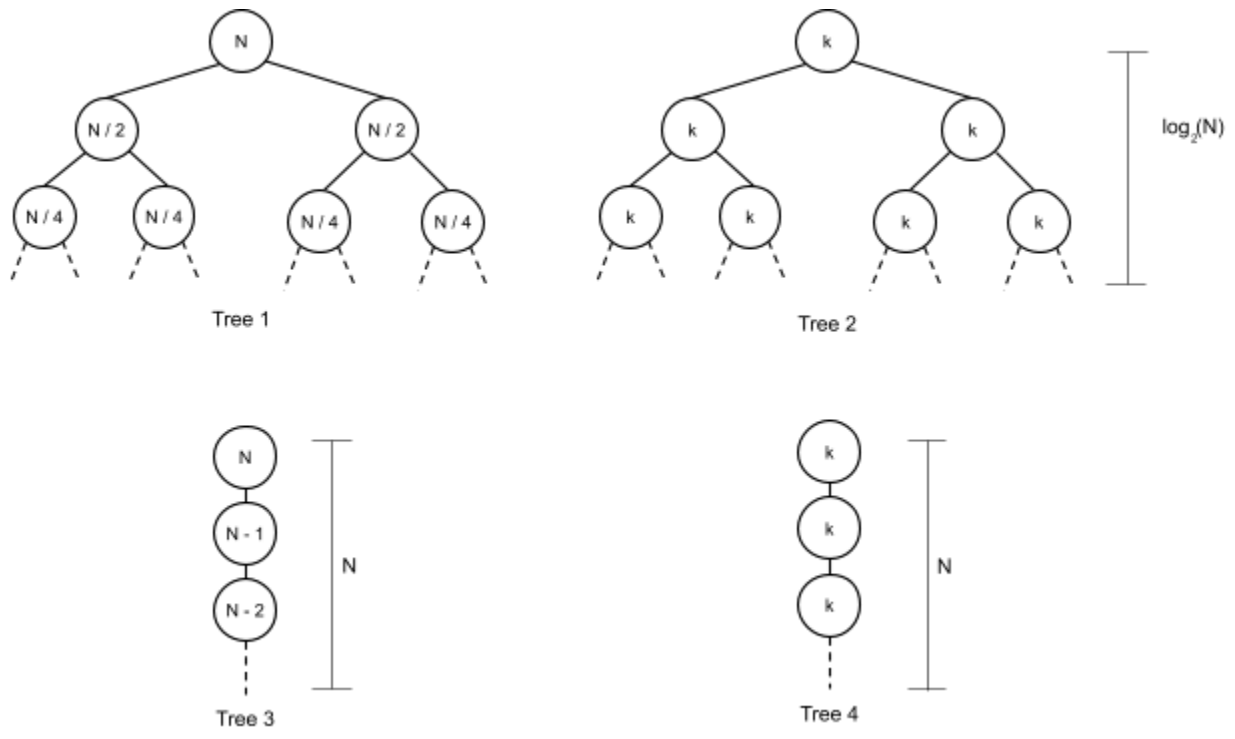
On the right, equation at top: T(N) = T(N / 2) + T(N / 2). Below the equation is a visual structure for the recursion tree. Atop the tree is a node with text "N". Two lines below the "N" node connect to "N / 2" nodes. Each of those has two lines below connecting to "N / 4" nodes. "..." labels below each "N / 4" node indicate that the process continues. Height bar appears to the side of the recursion tree indicating the height: log_2(N). Each level of nodes has a calculation to the side that equates to N: top level is "N", next level is "(N / 2) * 2 = N", and the last level is "(N / 4) * 4 = N".

Animation captions:

1. An algorithm like binary search does a constant number of operations, k, followed by a recursive call on half the array.
2. The root node in the recursion tree represents k operations inside the first function call.
3. Recursive operations are represented below the node. The first recursive call also does k operations.
4. The tree's height corresponds to the number of recursive calls. Splitting the input in half each time results in $\log_2 N$ recursive calls. $O(\log_2 N)$ = $O(\log N)$.

5. Another algorithm may perform N operations then two recursive calls, each on N / 2 items. The root node represents N operations.
6. The initial call makes two recursive calls, each of which has a local N value of the initial N value / 2.
7. N operations are done per level.
8. The tree has $O(\log_2 N)$ levels. $O(N * \log_2 N)$ = $O(N \log N)$ operations are done in total.

PARTICIPATION ACTIVITY 2.10.4: Matching recursion trees with runtime complexities.



How to use this tool

Mouse: Drag/drop
Keyboard: Grab/release `Spacebar` (or `Enter`). Move `↑` `↓` `←` `→`. Cancel `Esc`

Tree 4 **Tree 3** **Tree 2** **Tree 1**

T(N) = k + T(N / 2) + T(N / 2)

T(N) = k + T(N - 1)

T(N) = N + T(N - 1)

T(N) = N + T(N / 2) + T(N / 2)

Reset

PARTICIPATION ACTIVITY 2.10.5: Recursion trees.

Suppose a recursive function's runtime is $T(N) = 7 + T(N-1)$.

1) How many levels will the recursion tree have?

○ 7

○ $\log_2 N$

○ N

2) What is the runtime complexity of the function using Big O notation?

○ $O(1)$

○ $O(\log N)$

○ $O(N)$

PARTICIPATION ACTIVITY 2.10.6: Recursion trees.

Suppose a recursive function's runtime is $T(N) = N + T(N-1)$.

1) How many levels will the recursion tree have?

○ $\log_2 N$

○ N

○ N^2

2) The runtime can be expressed by the

series N + (N - 1) + (N - 2) + ... + 3 + 2 + 1. Which expression is mathematically equivalent?

- $N * \log_2 N$
- $(N/2) * (N + 1)$

3) What is the runtime complexity of the function using Big O notation?

- $O(N)$
- $O(N^2)$

2.11 LAB: Binary search with custom comparer.

LAB ACTIVITY | 2.11.1: LAB: Binary search with custom comparer. Full screen 0 / 10

Binary search with non-integer arrays

This book provides an implementation of binary search that operates on an integer array. But an implementation that can operate on arrays of *any* type has more practical value. This lab focuses on a binary search implementation that uses a template type for array elements and a custom comparison function.

Step 1: Inspect Comparer.h, IntComparer.h, and StringComparer.h

File Comparer.h contains the Comparer abstract base class definition. Comparer is a template class that has one member function named Compare(). Compare() takes two arguments for two items to compare. `Compare(a, b)` returns an integer:

- greater than 0 if a > b,
- less than 0 if a < b, or
- equal to 0 if a == b.

Comparer<T> is a template class. Class IntComparer inherits from Comparer<int> and so implements Compare() to compare two integers. Class StringComparer inherits from

Comparer<std::string> and so implements Compare() to compare two strings.

Step 2: Inspect Searcher.h

File Searcher.h contains the Searcher class definition. Searcher has one member function named BinarySearch(). BinarySearch() has parameters for an array to search, the array's size, a key to search for, and a Comparer object. Searcher is also a template class, and Searcher's template type T is used for BinarySearch()'s array, search key, and Comparer template type. So BinarySearch() can be implemented to search arrays of any type, provided a Comparer exists to compare elements of that type.

Step 3: Implement the Searcher class's BinarySearch() function

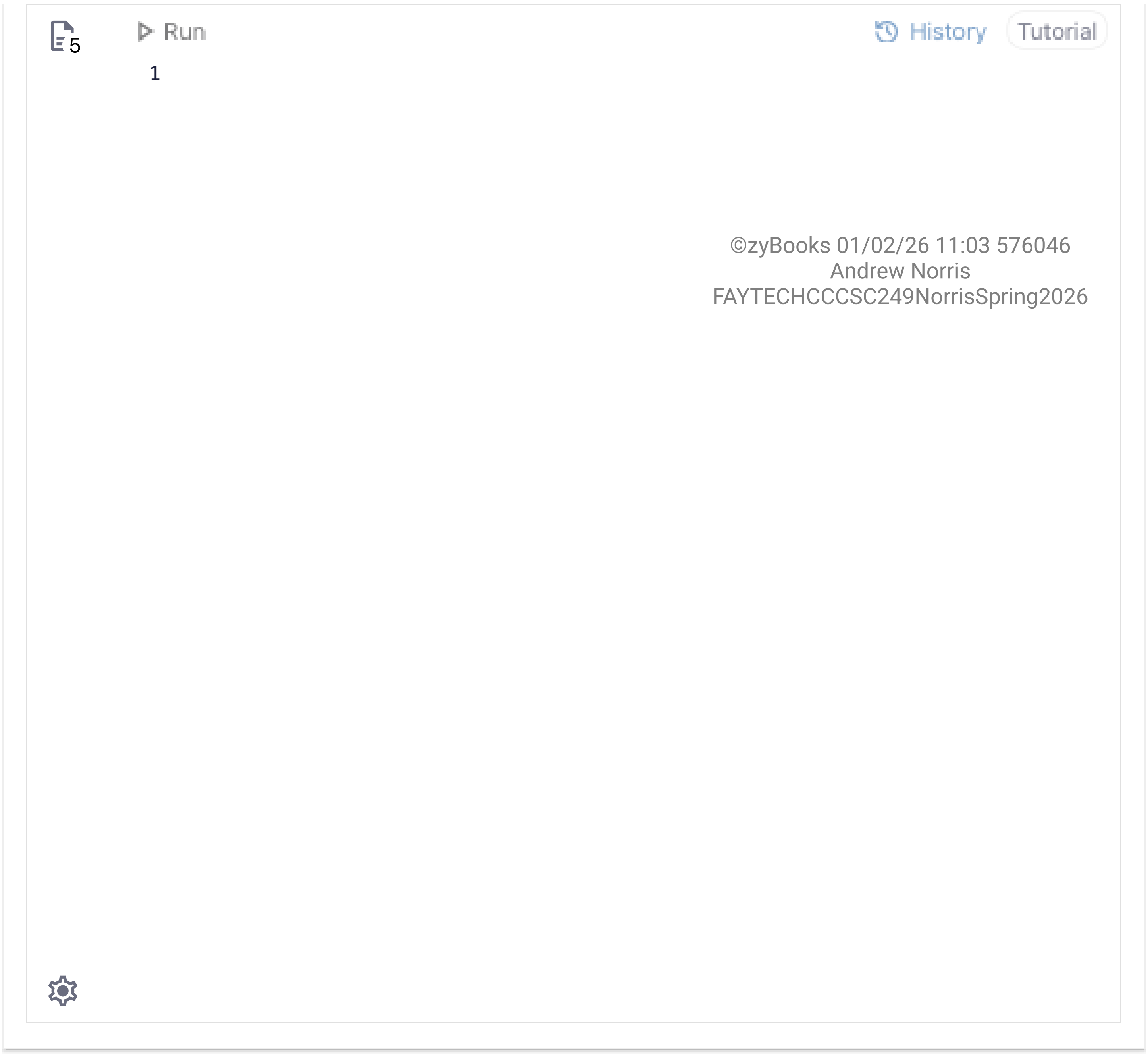
Implement the Searcher class's BinarySearch() template function in the Searcher.h file. The function should perform a binary search on the sorted array (first parameter) for the key (third parameter). BinarySearch() returns the key's index if found, -1 if not found. Compare an array element to the key using the Compare() member function of the comparer object passed as BinarySearch()'s last argument.

Some test cases exist in main() to test BinarySearch() with both string searches and integer searches. Clicking the Run button will display test case results, each starting with "PASS" or "FAIL". Ensure that all tests are passing before submitting code.

Each test in main() only checks that BinarySearch() returns the correct result, but does not check the number of comparisons performed. The unit tests that grade submitted code check both BinarySearch()'s return value *and* the number of comparisons performed. The last unit test also uses arrays with element types other than int and string.

Open new tab Dock

Model Solution


5
Run
History
Tutorial