

7.1 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 a list, and checks each element until the search key is found or the end of the list is reached.

PARTICIPATION
ACTIVITY

7.1.1: Linear search algorithm checks each element until key is found.



Animation content:

undefined

Animation captions:

1. Linear search starts at first element and searches elements one-by-one.
2. Linear search will compare all elements if the search key is not present.

Figure 7.1.1: 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 + ".")
    }
}

```

©zyBooks 04/06/23 14:20 1369565
 Gayoung Kim
 LEHIGHCSE017Spring2023

```

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.

```

©zyBooks 04/06/23 14:20 1369565
 Gayoung Kim
 LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

7.1.2: Linear search algorithm execution.

Given list: (20, 4, 114, 23, 34, 25, 45, 66, 77, 89, 11).

- 1) How many list elements will be compared to find 77 using linear search?



Check**Show answer**

- 2) How many list elements will be checked to find the value 114 using linear search?



©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

 Check**Show answer**

- 3) How many list elements will be checked if the search key is not found using linear search?

 **Check****Show answer**

Algorithm runtime

An algorithm's **runtime** is the time the algorithm takes to execute. If each comparison takes 1 μ s (1 microsecond), a linear search algorithm's runtime is up to 1 s to search a list with 1,000,000 elements, 10 s for 10,000,000 elements, and so on. Ex: Searching Amazon's online store, which has more than 200 million items, could require more than 3 minutes.

An algorithm typically uses a number of steps proportional to the size of the input. For a list with 32 elements, linear search requires at most 32 comparisons: 1 comparison if the search key is found at index 0, 2 if found at index 1, and so on, up to 32 comparisons if the search key is not found. For a list with N elements, linear search thus requires at most N comparisons. The algorithm is said to require "on the order" of N comparisons.

PARTICIPATION ACTIVITY

7.1.3: Linear search runtime.



©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

- 1) Given a list of 10,000 elements, and if each comparison takes 2 μ s, what is the fastest possible runtime for linear search?

 μ s**Check****Show answer**

- 2) Given a list of 10,000 elements, and if each comparison takes 2 μ s, what is the longest possible runtime for linear search?

μ s

Check

Show answer

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

7.2 Binary search

Linear search vs. binary search

Linear search may require searching all list 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 a contact list is 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 half.

PARTICIPATION ACTIVITY

7.2.1: Using binary search to search contacts on your phone.



Animation captions:

1. A contact list stores contacts 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. Pooja is before Sharod, so the search continues with only those contacts between Muhammad and Sharod, which is one fourth of the contacts.
4. The middle element between Muhammad and Sharod is Pooja. Each step reduces the number of contacts to search by half.

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

7.2.2: Using binary search to search a contact list.



A contact list is searched for Bob.

Assume the following contact list: (Amy, Bob, Chris, Holly, Ray, Sarah, Zoe)

- 1) What is the first contact searched?

Check

Show answer

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023



- 2) What is the second contact searched?

Check

Show answer



Binary search algorithm

Binary search is a faster algorithm for searching a list if the list's elements are sorted and directly accessible (such as an array). Binary search first checks the middle element of the list. 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 sublist (if the search key was less than the middle element) or the remaining right sublist (if the search key was greater than the middle element).

PARTICIPATION ACTIVITY

7.2.3: Binary search efficiently searches sorted list by reducing the search space by half each iteration.



Animation captions:

1. Elements with indices between low and high remain to be searched.
2. Search starts by checking the middle element.
3. If search key is greater than element, then only elements in right sublist need to be searched.
4. Each iteration reduces search space by half. Search continues until key found or search space is empty.

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

Figure 7.2.1: 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 + ".")  
    }  
}
```

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

```
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.
```

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



Given list: (4, 11, 17, 18, 25, 45, 63, 77, 89, 114).

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

Check**Show answer**

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



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

Check**Show answer**

- 3) Given an array with 32 elements, how many list elements will be checked if the key is less than all elements in the list, using binary search?

Check**Show answer**

Binary search efficiency

Binary search is incredibly efficient in finding an element within a sorted list. During each iteration or step of the algorithm, binary search reduces the search space (i.e., the remaining elements to search within) by half. The search terminates when the element is found or the search space is empty (element not found). For a 32 element list, if the search key is not found, the search space is halved to have 16 elements, then 8, 4, 2, 1, and finally none, requiring only 6 steps. For an N element list, the maximum number of steps required to reduce the search space to an empty sublist is

. Ex:

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



Animation captions:

1. A binary search begins with the middle element of the list. Each subsequent search reduces the search space by half. Using binary search, a match was found with only 3 comparisons.
2. Using linear search, a match was found after 6 comparisons. Compared to a linear search, binary search is incredibly efficient in finding an element within a sorted list.

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim
LEHIGHSE017Spring2023

If each comparison takes 1 μ s (1 microsecond), a binary search algorithm's runtime is at most 20 μ s to search a list with 1,000,000 elements, 21 μ s to search 2,000,000 elements, 22 μ s to search 4,000,000 elements, and so on. Ex: Searching Amazon's online store, which has more than 200 million items, requires less than 28 μ s; up to 7,000,000 times faster than linear search.

PARTICIPATION ACTIVITY

7.2.6: Linear and binary search efficiency.



- 1) Suppose a list of 1024 elements is searched with linear search. How many distinct list elements are compared against a search key that is less than all elements in the list?

 elements

Check

[Show answer](#)



- 2) Suppose a sorted list of 1024 elements is searched with binary search. How many distinct list elements are compared against a search key that is less than all elements in the list?

 elements

Check

[Show answer](#)



©zyBooks 04/06/23 14:20 1369565

Gayoung Kim
LEHIGHSE017Spring2023

CHALLENGE ACTIVITY

7.2.1: Binary search.



437612.2739130.qx3zqy7

Start

A colors list is searched for Orange using binary search.

Colors list: (Blue, Gold, Green, Indigo, Ochre, Orange, Red, Tan, Violet, Yellow)

What is the first color searched?

Ex: Yellow

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

What is the second color searched?

1

2

3

4

5

Check

Next

7.3 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

7.3.1: Constant time vs. non-constant time operations.



Animation content:

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

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.

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023



PARTICIPATION ACTIVITY

7.3.2: 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 3 constant time operations in the code below can collectively be considered 1 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 7.3.1: Common constant time operations.

Operation	Example
Addition, subtraction, multiplication, and division of fixed size integer or floating point values.	<pre>w = 10.4 x = 3.4 y = 2.0 z = (w - x) / y</pre>
Assignment of a reference, pointer, or other fixed size data value.	<p>©zyBooks 04/06/23 14:20 1369565 LEHIGHCSE017Spring2023</p> <pre>x = 1000 y = x a = true b = a</pre>
Comparison of two fixed size data values.	<pre>a = 100 b = 200 if (b > a) { ... }</pre>
Read or write an array element at a particular index.	<pre>x = arr[index] arr[index + 1] = x + 1</pre>

PARTICIPATION ACTIVITY

7.3.3: 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

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023





3) The hardware running the code is the only thing that affects what is and what is not a constant time operation.

- True
- False

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

7.4 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 \leq the best case $T(N)$, for all values of $N \geq 1$.
- **Upper bound:** A function $f(N)$ that is \geq the worst case $T(N)$, for all values of $N \geq 1$.

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: $5N + 3$, $5N + 2$, 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.

Upper and lower bounds in the context of runtime complexity

This section presents upper and lower bounds specifically in the context of algorithm complexity analysis. The constraint $N \geq 1$ is included because of the assumption that every algorithm presented in this book operates on a dataset with at least 1 item.

PARTICIPATION
ACTIVITY

7.4.1: Upper and lower bounds.

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023



Animation content:

undefined

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: \dots , and \dots . A single-term polynomial would provide a simpler picture.
3. \dots , shown in yellow, is a lower bound. The lower bound is less than or equal to the best case $T(N)$ for all $N \geq 1$.
4. The worst case's highest power of N is \dots . So the upper bound must be some constant times N^{\dots} such that $\dots \leq T(N) \leq \dots$ for all $N \geq 1$.
5. \dots does not work. Ex: When $N = 1$, \dots and \dots .
6. The lowest N that satisfies requirements is 30. \dots is greater than or equal to the worst case $T(N)$ for all $N \geq 1$. So \dots , shown in orange, is an upper bound.
7. Together, the upper and lower bounds enclose all possible runtimes for this algorithm.

PARTICIPATION ACTIVITY

7.4.2: Upper and lower bounds.



Suppose an algorithm's best case runtime complexity is \dots , and the algorithm's worst case runtime is \dots .

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?



-
-
-

3) \dots 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?



-
-
-

Growth rates and asymptotic notations

©zyBooks 04/06/23 14:20 1369565

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 $O(n^2)$, the algorithm could be described as having a worst case growth rate of $\Theta(n^2)$.

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** provides a growth rate for an algorithm's upper bound.
- **Ω notation** provides a growth rate for an algorithm's lower bound.
- **Θ notation** provides a growth rate that is both an upper and lower bound.

Table 7.4.1: Notations for algorithm complexity analysis.

Notation	General form	Meaning
		A positive constant c exists such that, for all $N \geq 1$,
		A positive constant c' exists such that, for all $N \geq 1$,
		and

PARTICIPATION ACTIVITY

7.4.3: Asymptotic notations.



Suppose

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

1)

- True
- False



2)



- True
- False

3)



- True
- False

4)

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



- True
- False

5)



- True
- False

7.5 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.

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION
ACTIVITY

7.5.1: Determining Big O notation of a function.



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**7.5.2: Big O notation.**

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023



- 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 $O(734 \cdot N)$?

- $O(N)$
- $O(734)$
- $O(734 \cdot N)$



- 3) Which of the following Big O notations is equivalent to $O(12 \cdot N + 6 \cdot N^2 + 1000)$?

- $O(1000)$
- $O(N)$
- $O(N^2)$



Big O notation of composite functions

The following rules are used to determine the Big O notation of composite functions: c denotes a constant

Figure 7.5.1: Rules for determining Big O notation of composite functions

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

$g(N) \cdot O(f(N))$	$O(g(N) \cdot f(N))$
$g(N) + O(f(N))$	$O(g(N) + f(N))$

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023



PARTICIPATION ACTIVITY

7.5.3: Big O notation for composite functions.

Determine the simplified Big O notation.

1) $10 \cdot O(N)$

- $O(10)$
- $O(N)$
- $O(10 \cdot N)$



2) $10 + O(N)$

- $O(10)$
- $O(N)$
- $O(10 + N)$



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

- $O(N)$
- $O(3 \cdot N)$
- $O(N)$



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

- $O(N)$
- $O(N)$
- $O(N + N)$



5)

- $O(\quad)$
- $O(\quad)$
- $O(\quad)$

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023



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.

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

Table 7.5.1: Growth rates for different input sizes.

Function	N = 10	N = 50	N = 100	N = 1000	N = 10000	N = 100000
	3.3 μs	5.65 μs	6.6 μs	9.9 μs	13.3 μs	16.6 μs
	10 μs	50 μs	100 μs	1000 μs	10 ms	100 ms
	.03 ms	.28 ms	.66 ms	.0099 s	.132 s	1.66 s
	.1 ms	2.5 ms	10 ms	1 s	100 s	2.7 hours
	1 ms	.125 s	1 s	16.7 min	11.57 days	31.7 years
	.001 s	35.7 years				> 1000 years

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

PARTICIPATION ACTIVITY

7.5.4: Computational complexity graphing tool.



Number of computations vs number of elements

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spr23

N

N

N *

log N



Number of computations

9

6

Common Big O complexities

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

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 fact, in many cases, computer scientists have proven that certain types of problems can only be solved using quadratic or exponential algorithms.

Figure 7.5.2: Runtime complexities for various code examples.

Notation	Name	Example pseudocode
O(1)	Constant	<pre>FindMin(x, y) { if (x < y) { return x } else { return y } }</pre>

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

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

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

O(N)

Quadratic

```
SelectionSort(numbers, numbersSize) {  
    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  
    }  
}
```

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

O(c)

Exponential

```
Fibonacci(N) {  
    if ((1 == N) || (2 == N)) {  
        return 1  
    }  
    return Fibonacci(N-1) + Fibonacci(N-2)  
}
```

PARTICIPATION ACTIVITY

7.5.5: 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) has a ____ runtime complexity.

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



- linear-quadratic
- exponential
- quadratic

4) A linear search has a ____ runtime complexity.

- O($\log N$)
- O(N)
- O(N)

5) A selection sort has a ____ runtime complexity.

- O(N)
- O($N \log N$)
- O(N)

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

7.6 Algorithm analysis

Worst-case algorithm analysis

To analyze how runtime of an algorithm scales as the input size increases, we first determine 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

7.6.1: Runtime analysis: Finding the max value.

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.

©zyBooks 04/06/23 14:20 1369565
LEHIGHCSE017Spring2023



- 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
}
```

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

- 2) What is the 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
    }
}
```

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

- $O(1)$
- $O(-)$
- $O(N)$

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



```

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

```

- O(log N)
- O(—)
- O(N)

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

- 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 execute 5 operations before the loop, 3 operations each loop iteration, and 6 operations after the loop would have a runtime of $f(N) = 5 + 3N + 6$, which can be written as $O(1) + O(N) + O(1) = O(N)$. If the number of operations before the loop was 100, the big-O notation for those operations is still $O(1)$.

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION
ACTIVITY

7.6.3: Simplified runtime analysis: A constant number of constant time operations is $O(1)$. 

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

7.6.4: Constant time operations.

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023



- 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 will always have a complexity of $O(N)$.

- True
 False

- 2) The complexity of the algorithm 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 $O(1)$.



©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

```

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
    }
}

```

©zyBooks 04/06/23 14:20 1369565
 Gayoung Kim
 LEHIGHCSE017Spring2023

- 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

7.6.5: Runtime analysis of nested loop: Selection sort algorithm.



Animation content:

undefined

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 7.6.1: Common summation: Summation of consecutive numbers.

PARTICIPATION ACTIVITY

7.6.6: Nested loops.



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

1)

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



$O(N)$

$O(N^2)$

2)

```
for (i = 0; i < N; i++) {  
    for (j = 0; j < (N - 1);  
j++) {  
        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)

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



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

- O(N)
- O(N)



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

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

- O(N)
- O(N)
- O(N)

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



7.7 Recursive definitions

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Recursive algorithms

An **algorithm** is a sequence of steps, including at least 1 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

7.7.1: Recursive factorial algorithm for positive numbers.



Animation captions:

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

1. A recursive algorithm to compute N factorial has 2 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

7.7.2: 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

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Recursive functions

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

Table 7.7.1: Sample recursive functions: Factorial, CumulativeSum, and ReverseList.

<pre>Factorial(N) { if (N == 1) return 1 else return N * Factorial(N - 1) }</pre>	<pre>CumulativeSum(N) { if (N == 0) return 0 else return N + CumulativeSum(N - 1) }</pre>
<pre>ReverseList(list, startIndex, endIndex) { if (startIndex >= endIndex) return else { Swap elements at startIndex and endIndex ReverseList(list, startIndex + 1, endIndex - 1) } }</pre>	

PARTICIPATION
ACTIVITY

7.7.3: 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 ReverseList is called on a list of size 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 list.

- True



7.8 Recursive algorithms

Fibonacci numbers

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

The **Fibonacci sequence** is a numerical sequence where each term is the sum of the previous 2 terms in the sequence, except the first 2 terms, which are 0 and 1. A recursive function can be used to calculate a **Fibonacci number**: A term in the Fibonacci sequence.

Figure 7.8.1: 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

7.8.1: FibonacciNumber recursive function.



- 1) What does FibonacciNumber(2)
return?

Check

[Show answer](#)



- 2) What does FibonacciNumber(4)
return?

Check

[Show answer](#)

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



- 3) What does FibonacciNumber(8)
return?

/ /

Check

Show answer

©zyBooks 04/06/23 14:20 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

Recursive binary search

Binary search is an algorithm that searches a sorted list for a key by first comparing the key to the middle element in the list and recursively searching half of the remaining list so long as the key is not found.

Binary search first checks the middle element of the list. If the search key is found, the algorithm returns the index. If the search key is not found, the algorithm recursively searches the remaining left sublist (if the search key was less than the middle element) or the remaining right sublist (if the search key was greater than the middle element).

Figure 7.8.2: BinarySearch recursive algorithm.

```
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

7.8.2: Recursive binary search.

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Suppose $\text{BinarySearch}(\text{numbers}, 0, 6, 42)$ is used to search the list (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

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

7.8.3: Recursive binary search base case.



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



- The low argument is greater than the high argument.
- The list element at index `mid` equals the key.
- The list element at index `mid` is less than the key.

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

7.9 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 .

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

7.9.1: Worst case binary search runtime complexity.



Animation content:

undefined

Animation captions:

1. In the non-base case, `BinarySearch` does some $O(1)$ operations plus a recursive call on half the input list.
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. A list 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

7.9.2: 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

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



7

64

3) Which function is a recurrence relation?

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



Recursion trees

The runtime complexity of any recursive function can be split into 2 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 operation 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.

PARTICIPATION
ACTIVITY

7.9.3: Recursion trees.



Animation captions:

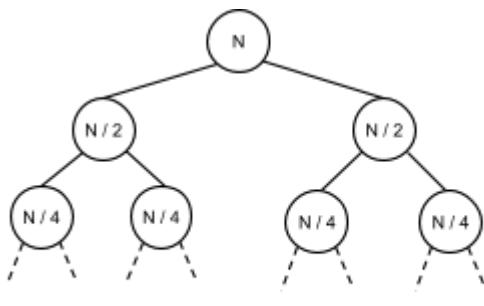
1. An algorithm like binary search does a constant number of operations, k , followed by a recursive call on half the list.
2. The root node in the recursion tree represents k operations inside the first function call.
3. Recursive operations are represented below the node. The first recursive call also does k operations.
4. The tree's height corresponds to the number of recursive calls. Splitting the input in half each time results in recursive calls. $O(\quad) = O(\quad)$.
5. Another algorithm may perform N operations then 2 recursive calls, each on $N / 2$ items. The root node represents N operations.
6. The initial call makes 2 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 $O(\quad)$ levels. $O(\quad) = O(\quad)$ operations are done in total.

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

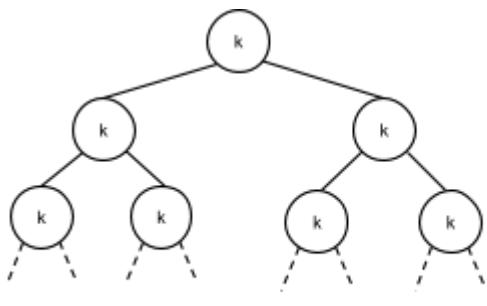
PARTICIPATION
ACTIVITY

7.9.4: Matching recursion trees with runtime complexities.





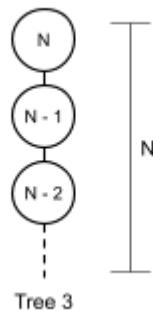
Tree 1



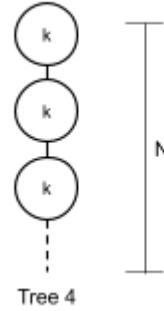
Tree 2

 $\log_2(N)$

©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



Tree 3



Tree 4

If unable to drag and drop, refresh the page.

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

7.9.5: Recursion trees.



©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Suppose a recursive function's runtime is

- 1) How many levels will the recursion tree have?

 7





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

- O(1)
- O()
- O()



©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION
ACTIVITY

7.9.6: Recursion trees.



Suppose a recursive function's runtime is

- 1) How many levels will the recursion tree have?

-
-
-



- 2) The runtime can be expressed by the series $N + (N - 1) + (N - 2) + \dots + 3 + 2 + 1$. Which expression is mathematically equivalent?

-
-



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

- O()
- O()



©zyBooks 04/06/23 14:20 1369565
Gayoung Kim
LEHIGHCSE017Spring2023