

2.1 Algorithm efficiency

Algorithm efficiency

An algorithm describes the method to solve a computational problem. Programmers and computer scientists should use or write efficient algorithms. **Algorithm efficiency** is typically measured by the algorithm's computational complexity. **Computational complexity** is the amount of resources used by the algorithm. The most common resources considered are the runtime and memory usage.

PARTICIPATION ACTIVITY

2.1.1: Computational complexity.



Animation captions:

1. An algorithm's computational complexity includes runtime and memory usage.
2. Measuring runtime and memory usage allows different algorithms to be compared.
3. Complexity analysis is used to identify and avoid using algorithms with long runtimes or high memory usage.

PARTICIPATION ACTIVITY

2.1.2: Algorithm efficiency and computational complexity.



- 1) Computational complexity analysis allows the efficiency of algorithms to be compared.

- True
- False



- 2) Two different algorithms that produce the same result have the same computational complexity.

- True
- False



- 3) Runtime and memory usage are the only two resources making up computational complexity.

- True
- False

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021



Runtime complexity, best case, and worst case

An algorithm's **runtime complexity** is a function, $T(N)$, that represents the number of constant time operations performed by the algorithm on an input of size N . Runtime complexity is discussed in more detail elsewhere.

Because an algorithm's runtime may vary significantly based on the input data, a common approach is to identify best and worst case scenarios. An algorithm's **best case** is the scenario where the algorithm does the minimum possible number of operations. An algorithm's **worst case** is the scenario where the algorithm does the maximum possible number of operations.

Input data size must remain a variable

A best case or worst case scenario describes contents of the algorithm's input data only. The input data size must remain a variable, N . Otherwise, the overwhelming majority of algorithms would have a best case of $N=0$, since no input data would be processed. In both theory and practice, saying "the best case is when the algorithm doesn't process any data" is not useful. Complexity analysis always treats the input data size as a variable.

PARTICIPATION ACTIVITY

2.1.3: Linear search best and worst cases.



Animation captions:

1. LinearSearch searches through array elements until finding the key. Searching for 26 requires iterating through the first 3 elements.
2. The search for 26 is neither the best nor the worst case.
3. Searching for 54 only requires one comparison and is the best case: The key is found at the start of the array. No other search could perform fewer operations.
4. Searching for 82 compares against all array items and is the worst case: The number is not found in the array. No other search could perform more operations.

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021



PARTICIPATION ACTIVITY

2.1.4: FindFirstLessThan algorithm best and worst case.

Consider the following function that returns the first value in a list that is less than the specified value. If no list items are less than the specified value, the specified value is returned.

```
FindFirstLessThan(list, listSize, value) {
    for (i = 0; i < listSize; i++) {
        if (list[i] < value)
            return list[i]
    }
    return value // no lesser value found
}
```

Best case**Worst case****Neither best nor worst case**

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

No items in the list are less than value.

The first half of the list has elements greater than value and the second half has elements less than value.

The first item in the list is less than value.

Reset**PARTICIPATION ACTIVITY**

2.1.5: Best and worst case concepts.



1) The linear search algorithm's best case scenario is when N = 0.

- True
- False



2) An algorithm's best and worst case scenarios are always different.

- True
- False

**Space complexity**©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

An algorithm's **space complexity** is a function, S(N), that represents the number of fixed-size memory units used by the algorithm for an input of size N. Ex: The space complexity of an algorithm that duplicates a list of numbers is $S(N) = 2N + k$, where k is a constant representing memory used for things like the loop counter and list pointers.

Space complexity includes the input data and additional memory allocated by the algorithm. An algorithm's **auxiliary space complexity** is the space complexity not including the input data. Ex: An algorithm to find the maximum number in a list will have a space complexity of $S(N) = N + k$, but an auxiliary space complexity of $S(N) = k$, where k is a constant.

PARTICIPATION ACTIVITY

2.1.6: FindMax space complexity and auxiliary space complexity.



©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

Animation content:

undefined

Animation captions:

1. FindMax's arguments represent input data. Non-input data includes variables allocated in the function body: maximum and i .
2. The list's size is a variable, N . Three integers are also used, making the space complexity $S(N) = N + 3$.
3. The auxiliary space complexity includes only the non-input data, which does not increase for larger input lists.
4. The function's auxiliary space complexity is $S(N) = 2$.

PARTICIPATION ACTIVITY

2.1.7: Space complexity of GetEvens function.



Consider the following function, which builds and returns a list of even numbers from the input list.

```
GetEvens(list, listSize) {
    i = 0
    evensList = Create new, empty list
    while (i < listSize) {
        if (list[i] % 2 == 0)
            Add list[i] to evensList
        i = i + 1
    }
    return evensList
}
```

- 1) What is the maximum possible size of the returned list?

- listSize
- listSize / 2

- 2) What is the minimum possible size of the returned list?

- listSize / 2

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

1 0

- 3) What is the worst case auxiliary space complexity of GetEvens if N is the list's size and k is a constant?

 $S(N) = N + k$ $S(N) = k$

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

- 4) What is the best case auxiliary space complexity of GetEvens if N is the list's size and k is a constant?

 $S(N) = N + k$ $S(N) = k$

2.2 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

2.2.1: 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.

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

Figure 2.2.1: 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 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

```

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

2.2.2: Linear search algorithm execution.

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

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

- 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?

[Check](#)[Show answer](#)

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

- 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 μs (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

2.2.3: Linear search runtime.



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

 μs [Check](#)[Show answer](#)

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021



- 2) Given a list of 10,000 elements, and if each comparison takes 2 μs , what is

the longest possible runtime for linear search?

μs

Check

Show answer

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

2.3 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

2.3.1: 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 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

PARTICIPATION
ACTIVITY

2.3.2: 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?



Show answer

- 2) What is the second contact searched?

Check**Show answer**

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021



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

2.3.3: 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.

Figure 2.3.1: Binary search algorithm.

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

```

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.

```

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

PARTICIPATION ACTIVITY

2.3.4: Binary search algorithm execution.

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



©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021



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

Check**Show answer**

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021



- 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 $\lfloor \log_2 N \rfloor + 1$. Ex: $\lfloor \log_2 32 \rfloor + 1 = 6$.

**PARTICIPATION
ACTIVITY**

2.3.5: Speed of linear search versus binary search to find a number within a sorted list.

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021



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.

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

2.3.6: Linear and binary search efficiency.



©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021



- 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****CHALLENGE ACTIVITY**

2.3.1: Binary search.

**Start**

A flowers list is searched for Azalea using binary search.

Flowers list: (Azalea, Daisy, Iris, Magnolia, Peony, Poppy, Rose, Violet)

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

What is the first flower searched?

 Ex: Violet

What is the second flower searched?

1

2

3

4

5

[Check](#)[Next](#)

©zyBooks 03/13/21 13:32 026027

Eric Knapp

STEVENSCS570Spring2021

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.



Animation content:

undefined

Animation captions:

1. $x = 10$, $y = 20$, $a = 1000$, and $b = 2000$ are assigning variables with values, and are constant time operations.
2. A CPU multiplies values 10 and 20 at the same speed as 1000 and 2000. Multiplication 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.

Eric Knapp
STEVENSCS570Spring2021**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



©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

- 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 2.4.1: 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.	<pre>x = 1000 y = x a = true b = a</pre>
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
```

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

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



©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

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 \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 $N \geq 1$.

Ex: An algorithm with best case runtime $T(N) = 7N + 36$ and worst case runtime $T(N) = 3N^2 + 10N + 17$, has a lower bound $f(N) = 7N$ and an upper bound $f(N) = 30N^2$. These lower and upper bounds provide a general picture of the runtime, while using simpler functions than the exact runtime.

Eric Knapp
STEVENSCS570Spring2021

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

2.5.1: Upper and lower bounds.



Animation content:

undefined

Animation captions:

1. An algorithm's worst and best case runtimes are represented by the blue and purple curves, respectively.
2. $2N^2$, shown in yellow, is a lower bound. The lower bound is less than or equal to both runtime functions for all $N \geq 1$.
3. $30N^2$, shown in orange, is an upper bound. The upper bound is greater than or equal to both runtime functions for all $N \geq 1$.
4. Together, the upper and lower bounds enclose all possible runtimes for this algorithm.

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

PARTICIPATION ACTIVITY

2.5.2: Upper and lower bounds.



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) Which function is a lower bound for the



algorithm?

- $5N^2$
- $5N$
- $3N$

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

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

3) $5N^2 + 7N$ is an upper bound for the algorithm.

- True
- False

4) $3N + 6$ is a lower bound for the algorithm.

- True
- False



©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021



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** 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.

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

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 $N \geq 1$, $T(N) \leq c * f(N)$.

Ω	$T(N) = \Omega(f(N))$	A positive constant c exists such that, for all $N \geq 1$, $T(N) \geq c * f(N)$.
Θ	$T(N) = \Theta(f(N))$	$T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$.

PARTICIPATION ACTIVITY

2.5.3: Asymptotic notations.

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

Suppose $T(N) = 2N^2 + N + 9$.

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

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021



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:

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

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.



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 $O(734 \cdot N)$?

- $O(N)$
- $O(734)$
- $O(734 \cdot N^2)$



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

- $O(1000)$
-

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021



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

Big O notation of composite functions

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

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

Figure 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($3 \cdot N^2$)

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

$O(N^3)$ 4) $2 \cdot N^3 + O(N^2)$  $O(N^2)$ $O(N^3)$ $O(N^2 + N^3)$ 5) $\log_2 N$

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

 $O(\log_2 N)$ $O(\log N)$ $O(N)$

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 2.6.1: Growth rates for different input sizes.

Function	$N = 10$	$N = 50$	$N = 100$	$N = 1000$	$N = 10000$	$N = 100000$
$\log_2 N$	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	1000 μs	10 ms	100 ms
$N \log_2 N$.03 ms	.28 ms	.66 ms	.099 s	.132 s	1.66 s
N^2	.1 ms	2.5 ms	10 ms	1 s	100 s	2.7 hours
N^3	1 ms	.125 s	1 s	16.7 min	11.57 days	31.7 years
2^N	.001 s	35.7 years	©zyBooks 03/13/21 13:32 926027			
			Eric Knapp			
			STEVENSCS570Spring2021			

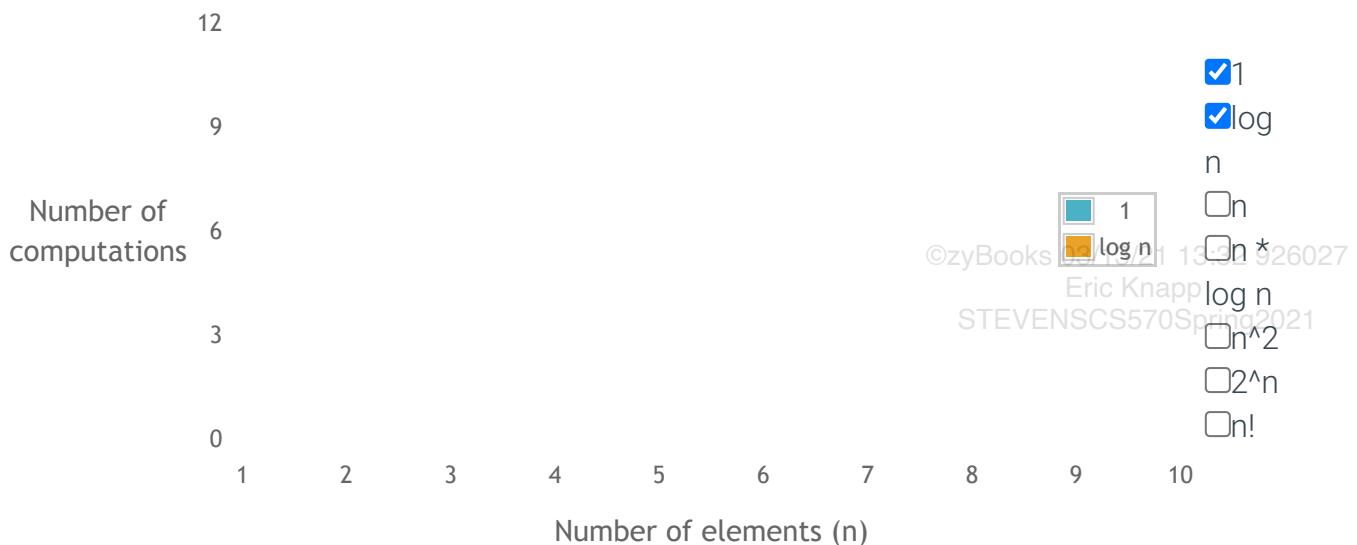
The interactive tool below illustrates graphically the growth rate of commonly encountered functions.

PARTICIPATION ACTIVITY

2.6.4: Computational complexity graphing tool.



Number of computations vs number of elements



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

Figure 2.6.2: 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>
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
}
```

zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

 $O(N)$

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
    }
}
```

 $O(N^2)$

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 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

 $O(c^N)$

Exponential

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

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

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.



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

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

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

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

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

2.7.1: 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.

PARTICIPATION ACTIVITY

2.7.2: Worst-case runtime analysis.



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

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

```
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 big-O notation for the worst-case runtime?



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

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

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

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
    }
}
```

- $O(1)$
- $O(\frac{N}{2})$
- $O(N)$

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
}
```

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

- $O(\log N)$
- $O(\frac{N}{2})$
- $O(N)$

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



```

i = 0
belowMinSum = 0.0
belowMinCount = 0
while (i < N && numbers[i] <=
maxVal) {
    belowMinCount = belowMinCount + 1
    belowMinSum = numbers[i]
    ++i
}
avgBelow = belowMinSum /
belowMinCount

```

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

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).

PARTICIPATION ACTIVITY

2.7.3: 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

2.7.4: Constant time operations.

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021



- 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
}
```

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

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.

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

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.

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021

Figure 2.7.1: Common summation: Summation of consecutive numbers.

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

PARTICIPATION ACTIVITY

2.7.6: Nested loops.



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

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


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

2)

©zyBooks 03/13/21 13:32 926027
Eric Knapp
STEVENSCS570Spring2021

```

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)

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021



```

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

```

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



```

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^2)
- O(N^3)

©zyBooks 03/13/21 13:32 926027

Eric Knapp

STEVENSCS570Spring2021