# 9.1 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 | 9.1.1: Constant time vs. non-constant time operations.

### Animation content:

undefined

### 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 | 9.1.2: 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 9.1.1: Common constant time operations.

| Operation | Example |
|---|---|
| Addition, subtraction, multiplication, and division of fixed size integer or floating point values. | `w = 10.4`<br>`x = 3.4`<br>`y = 2.0`<br>`z = (w - x) / y` |
| Assignment of a reference, pointer, or other fixed size data value. | `x = 1000`<br>`y = x`<br>`a = true`<br>`b = a` |
| Comparison of two fixed size data values. | `a = 100`<br>`b = 200`<br>`if (b > a) {`<br>`   ...`<br>`}` |

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

PARTICIPATION ACTIVITY | 9.1.3: 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

# 9.2 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 ≥ 1.

- ***Upper bound***: A function f(N) that is ≥ the worst case T(N), for all values of N ≥ 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 ≥ 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** 9.2.1: 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. The best case expression itself is a lower bound, but is a polynomial with three terms: , , and . A single-term polynomial would provide a simpler picture.
3. , 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 is . So the upper bound must be some constant times such that for all N ≥ 1.
5. does not work. Ex: When , and .
6. The lowest that satisfies requirements is 30. is greater than or equal to the worst case T(N) for all N >= 1. So , shown in orange, is an upper bound.
7. Together, the upper and lower bounds enclose all possible runtimes for this algorithm.

PARTICIPATION ACTIVITY | 9.2.2: Upper and lower bounds.

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

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) 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

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 , the algorithm could be described as having a worst case growth rate of . ***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 9.2.1: Notations for algorithm complexity analysis.

| Notation | General form | Meaning |
|---|---|---|
| | | A positive constant exists such that, for all N ≥ 1, . |
| | | A positive constant exists such that, for all N ≥ 1, . |
| | | and . |



PARTICIPATION ACTIVITY | 9.2.3: Asymptotic notations.

Suppose .

1)

- True
- False

2)

- True
- False

3)

- True
- False

4)

- True
- False



5)

- True
- False

# 9.3 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** | 9.3.1: 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** | 9.3.2: 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 )

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



- O(1000)
- O(N)
- O(N )

## 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 9.3.1: Rules for determining Big O notation of composite functions.

| Composite function | Big O notation |
|---|---|
| c · O(f(N)) | O(f(N)) |
| c + O(f(N)) | O(f(N)) |
| g(N) · O(f(N)) | O(g(N) · f(N)) |
| g(N) + O(f(N)) | O(g(N) + f(N)) |

**PARTICIPATION ACTIVITY** 9.3.3: Big O notation for composite functions.



Determine the simplified Big O notation.

1) 10 · O(N )

- O(10)
- O(N )

- O(10 · N )

2) 10 + O(N )

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



3) 3·N · O(N )

- O(N )
- O(3·N )
- O(N )

4) 2·N + O(N )

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

5)

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

## 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 9.3.1: Growth rates for different input sizes.



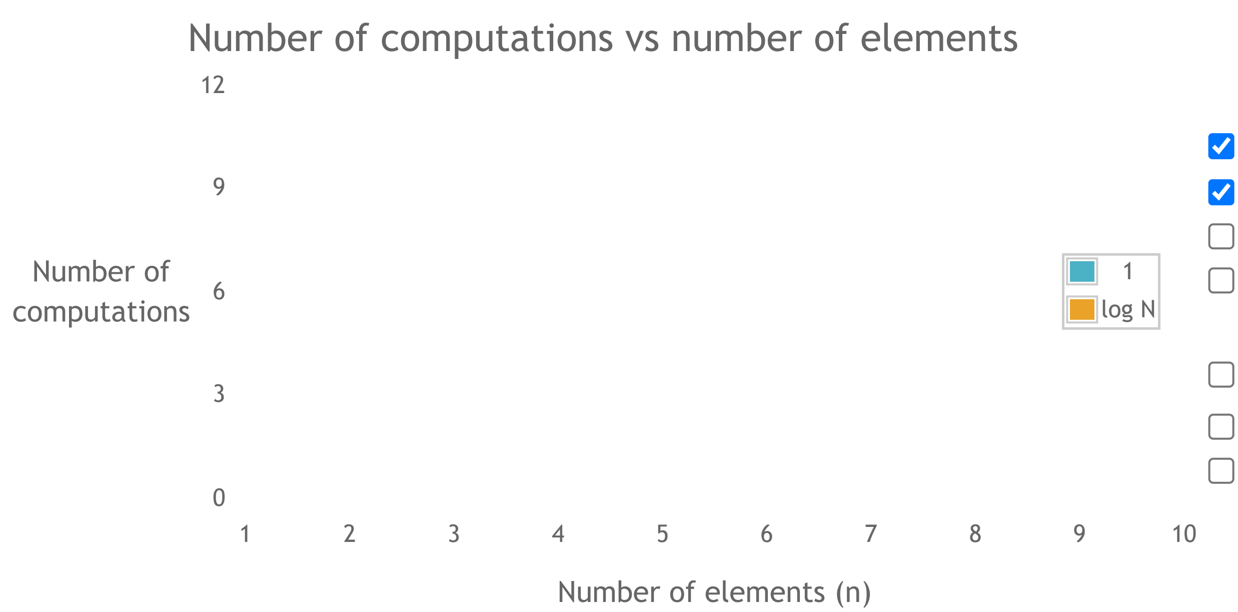
| Function | N = 10 | N = 50 | N = 100 | N = 1000 | N = 10000 | N = 100000 |
|---|---|---|---|---|---|---|
| | 3.3 μs | 5.65 μs | 6.6 μs | 9.9 μs | 13.3 μs | 16.6 μs |

| | | | | | | |
|---|---|---|---|---|---|---|
| | 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** 9.3.4: 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 fact, in many cases, computer scientists have proven that certain types of problems can only be solved using quadratic or exponential algorithms.

Figure 9.3.2: Runtime complexities for various code examples.

<table>
<tr><th>Notation</th><th>Name</th><th>Example pseudocode</th></tr>
<tr><td>O(1)</td><td>Constant</td><td><pre>
FindMin(x, y) {
   if (x < y) {
      return x
   }
   else {
      return y
   }
}
</pre></td></tr>
<tr><td>O(log N)</td><td>Logarithmic</td><td><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></td></tr>
<tr><td>O(N)</td><td>Linear</td><td><pre>
LinearSearch(numbers, numbersSize, key) {
  for (i = 0; i < numbersSize; ++i) {
      if (numbers[i] == key) {
         return i
      }
   }

   return -1 // not found
}
</pre></td></tr>
</table>

<table>
<tr><td>O(N log N)</td><td>Linearithmic</td><td><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></td></tr>
<tr><td>O(N )</td><td>Quadratic</td><td><pre>
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
   }
}
</pre></td></tr>
<tr><td>O(c )</td><td>Exponential</td><td><pre>
Fibonacci(N) {
  if ((1 == N) || (2 == N)) {
     return 1
  }
  return Fibonacci(N-1) + Fibonacci(N-2)
}
</pre></td></tr>
</table>



**PARTICIPATION ACTIVITY** | 9.3.5: 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.

- 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 )

# 9.4 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 | 9.4.1: 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 | 9.4.2: 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 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(—)
- 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)

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).



**PARTICIPATION ACTIVITY** | 9.4.3: 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** | 9.4.4: 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 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).



```
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** 9.4.5: 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 9.4.1: Common summation: Summation of consecutive numbers.

**PARTICIPATION ACTIVITY** 9.4.6: 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)
```
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 )

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 )

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 )



# 9.5 Searching and algorithms

This section has been set as optional by your instructor.

## 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 | 9.5.1: 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 9.5.1: 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** 9.5.2: 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?

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 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** 9.5.3: 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?

$\mu s$

Check Show answer



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

$\mu s$

Check Show answer

# 9.6 Analyzing the time complexity of recursive algorithms

This section has been set as optional by your instructor.

## 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 | 9.6.1: 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 | 9.6.2: 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?

-
-
-



## 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.

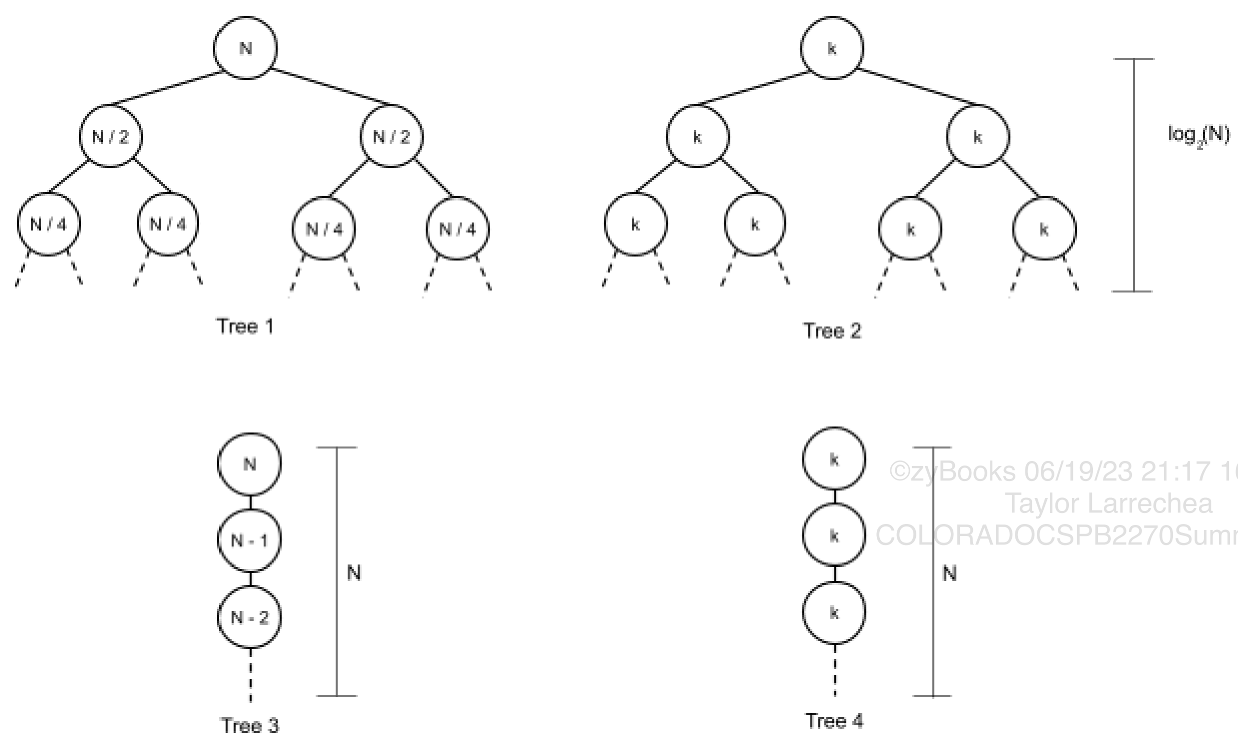
**PARTICIPATION ACTIVITY** 9.6.3: Recursion trees.

## 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( ) = O( ).
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( ) levels. O( ) = O( ) operations are done in total.

**PARTICIPATION ACTIVITY** 9.6.4: Matching recursion trees with runtime complexities.






If unable to drag and drop, refresh the page.

Tree 3 | Tree 4 | 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 | 9.6.5: Recursion trees.

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( )

PARTICIPATION ACTIVITY | 9.6.6: 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) + ... + 3 + 2 + 1. Which expression is mathematically equivalent?



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

- O( )
- O( )