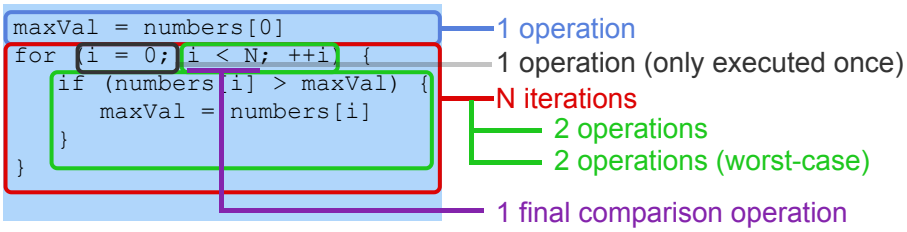
2.6 Algorithm analysis

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 2.6.1: Runtime analysis: Finding the max value.

Start 2x speed



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

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.

Feedback?

PARTICIPATION ACTIVITY 2.6.2: 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(\(\frac{N}{2}\))
- O(N)

4) 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(\(\frac{N}{2}\))

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
```

O(1)

O(N)

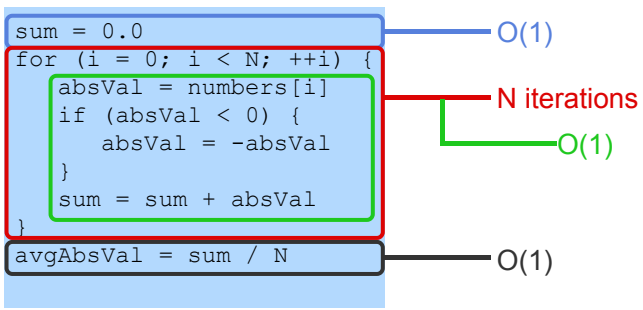
Feedback?

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

Start 2x speed



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

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.

Feedback?

PARTICIPATION ACTIVITY 2.6.4: 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

Feedback?

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

Start 2x speed

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

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

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

4 operations, generalized as constant c

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

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

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

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.

Feedback?

Figure 2.6.1: Common summation: Summation of consecutive numbers.

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

Feedback?

PARTICIPATION ACTIVITY 2.6.6: 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\(^{2}\))

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\(^{3}\))

Feedback?

How was this section?

Provide feedback