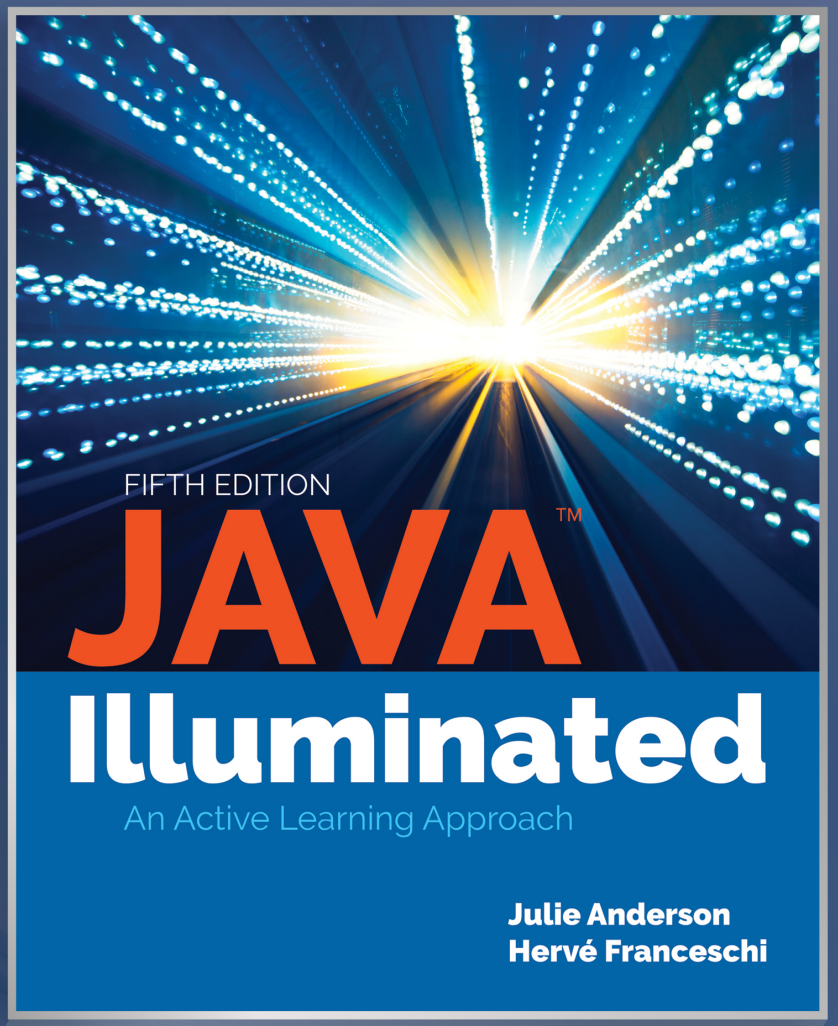
Chapter 15

Running Time Analysis



Topics

- Orders of Magnitude and Big-Oh Notation
- Running Time Analysis of Algorithms
  - Counting Statements
  - Evaluating Recursive Methods
  - Tracking the Number of Statements Executed
  - Searching and Sorting Algorithms

Introduction

- An ever-increasing amount of data requires efficient processing.
- Execution time depends on the algorithm used.
- Algorithms must be well-designed and efficient.
- To measure the performance of an algorithm, we use the term running time.
- Because the absolute running time of an algorithm depends on the amount of data it processes, we express running time as a function of the number of inputs.
- This allows us to compare the relative efficiency of multiple algorithms.

Inputs and *n*

- We use *n* to represent the number of inputs or the input value (the size of the problem).
- We are interested in relative time.
- We look for orders of magnitude.
  - For 1 million inputs, *n* and *n* + 17 will yield similar performance results.
  - However, a running time of n^2 (as compared to *n*) will have a significant impact on performance as the number of inputs increases.

Orders of Magnitude

Various orders of magnitude as a function of the number of inputs (n)

Order of Magnitude	Number of Statements Executed			
	$n = 10$	$n = 20$	$n = 1{,}000$	$n = 1$ million
$\log n$	2.23	3.23	Approx. 10	Approx. 20
n	10	20	1000	10^6
$n \log n$	22.3	64.6	Approx. 10,000	Approx. $20*10^6$
n^2	100	400	10^6	10^{12}
n^3	1,000	8,000	10^9	10^{18}
2^n	1,024	Approx. 10^6	Approx. 10^{300}	Approx. 10^{300000}

**As you can see, algorithms with running times where n is an exponent should be avoided.

The Big-Oh Notation

- Running times are often represented using the Big-Oh notation ($O(n)$, for example).
- To estimate the Big-Oh of a function representing a running time, follow these rules:
  - Keep only the dominant term, that is, the term that grows the fastest as n increases.
  - Ignore the coefficient of the dominant term.

Examples of Running Times and Their Big-Oh

$f(n)$	Dominant Term	Big-Oh
$2 * n + 19$	$2 * n$	$O(n)$
$3 * n^2 + 6 * n + 12$	$3 * n^2$	$O(n^2)$
$n^3 + 9 * n^2 + 5 * n + 2$	n^3	$O(n^3)$
$3 * 2^n + 5 * n^3 + 3 * n + 7$	$3 * 2^n$	$O(2^n)$
$n + 7 * \log n$	n	$O(n)$
$2 * n * \log n + 8 * n + \log n + 8$	$2 * n * \log n$	$O(n * \log n)$
$3 * \log n + 35$	$3 * \log n$	$O(\log n)$

Counting Statements

One simple method to estimate the running time of an algorithm is to count the number of times each statement is executed.

This method calculates the sum of all elements in an array of size *n*. The number of times each statement is executed is shown as comments:

```
public static int addElements ( int [ ] arr )
{
  int sum = 0;                      // 1
  int i = 0;                           // 1
  while ( i < arr.length )          // n + 1
  {
     sum += arr[ i ];             // n
     i++;                              // n
  }
  return sum;                          // 1
}
```

Calculate Total Statements Executed

The total number of statements executed is found by adding the number of times each statement is executed.

$$T(n) = 1 + 1 + (n + 1) + n + n + 1$$
$$= 3n + 4$$
$$= O(n)$$

So we can say that the running time of the *addElements* method is $O(n)$.

Note that in the end, we do not need an exact count of the statements executed. The Big-Oh running time can be used to compare this method to other algorithms.

Another Example

This method determines the maximum value in a two-dimensional array of *ints*. Here, the comments label each statement for an analysis on the next slide:

```
public static int calculateMaximum( int [ ][ ] arr )
{
  int maximum  = arr[0][0];                    // ( 1 )
  for( int i = 0; i < arr.length; i++ )        // ( 2 )
  {
   for( int j = 0; j < arr[i].length; j++ ) // ( 3 )
   {
    if ( maximum < arr[i][j] )                 // ( 4 )
         maximum  = arr[i][j];                 // ( 5 )
   }
  }
  return maximum;                              // ( 6 )
}
```

Analysis

With an array having n rows with each row having n columns, we can develop the following analysis:

Statement	# Times Executed
(1)	1
(2)	$1 + (n + 1) + n = 2 * n + 2$
(3)	$n * (1 + (n + 1) + n) = 2 * n^2 + 2 * n$
(4)	$n * n = n^2$
(5)	between 0 and $n * n$
(6)	1

For statements 2 and 3, the *for* loop headers consist of 3 statements.

For statement 4, we enter the outer *for* loop n times, and for each entry, the inner *for* loop executes n times.

For statement 5, the number of executions depends on the data. It will be a minimum of 0 and a maximum of n. We will call this unknown x.

Upper and Lower Bounds

Thus, the total number of statements executed, $T(n)$, is equal to:

$$T(n) = 1 + (2 * n + 2) + (2 * n^2 + 2 * n) + (n^2) + x + 1$$
$$= 3 * n^2 + 4 * n + 4 + x$$

with $x <= n * n$

Furthermore, because the value of x is between 0 and n^2, we can calculate the lower bounds of $T(n)$ by substituting 0 for x and the upper bounds by substituting n^2 for x.

$$3 * n^2 + 4 * n + 4 <= T(n) <= 3 * n^2 + 4 * n + 4 + n^2$$
$$3 * n^2 + 4 * n + 4 <= T(n) <= 4 * n^2 + 4 * n + 4$$

Both lower and upper bounds of $T(n)$ are $O(n^2)$.
So $T(n)$ is $O(n^2)$.

Sequential Search Running Time (1 of 2)

This algorithm searches an array of *n* elements for a *key* value:

```
public int sequentialSearch( int [ ] array,
                             int key )
{
  for ( int i = 0; i < array.length; i++ )    // ( 1 )
    if ( array[ i ] == key )                  // ( 2 )
       return i;                              // ( 3 )
  return -1;                                  // ( 4 )
}
```

Statement	# Times Executed
(1)	1 + (between 1 and $(n + 1)$) + (between 0 and n)
(2)	between 1 and n
(3)	0 or 1
(4)	1 or 0

Sequential Search Continued (2 of 2)

Taking the lower and upper bounds, we calculate the total number of statements executed as $T(n)$:

```
1 + (1) + (0) + 1 + 1 <=
                T(n)
<= 1 + (n + 1) + n + n + 1

          4 <= T(n) <= 3*n + 3
```

`T(n) <= 3*n + 3` shows that `T(n)` is `O(n)`

However, we cannot really tell from the coding of the function how many statements will be executed as a function of *n*. In these situations, it is interesting to consider three running times:

- The worst-case running time
- The best-case running time
- The average-case running time

Best, Worst, and Average Cases

- In the worst case, the search key is not found in the array or it is found in the last element, $T(n) = 3 * n + 3$, and therefore $T(n)$ is $O(n)$.
- In the best case, the element we are looking for is at index 0 of the array, and only four statements will be executed, independent of the value of n. Thus, the best-case running time is $O(1)$ because we do not take the multiplying constant into consideration when we compute a Big-Oh.
- In the average case, we find the element we are looking for in the middle of the array, and the value of $T(n)$ will be

$$T(n) = 1 + (n + 1)/2 + n/2 + n/2 + 1$$
$$= 3 * n/2 + 2 \quad 1/2$$
$$= O(n)$$

Evaluating Recursive Methods

Consider a recursive method that takes a parameter, *n*, and returns 2^n. We evaluate two ways of writing the method.

The first method: *powerOf2A* is designed using:

Base case: when $n = 0$, $2^0 = 1$

General case: $2^n = 2 * 2^{n-1}$

The code becomes:

```
public static int powerOf2A( int n ) // n >= 0
{
    if ( n == 0 )
        return 1;
    else
        return 2 * powerOf2A( n - 1 );
}
```

Defining a Recurrence Relation

Let's compute the running time of *powerOf2A* as a function of the input *n*; we will call it *T*1(*n*).

- In the base case (*n* == 0), *powerOf2A* makes only one comparison.

  Thus, *T*1(0) = 1.

- Because it takes T1(*n*) to compute and return *powerOf2A(n)*, then it takes *T*1(*n* – 1) to compute and return *powerOf2A(n – 1)*. Thus, in the general case,
  - The *if* statement will cost 1 instruction
  - Computing and returning *powerOf2A(n – 1)* will cost *T*1(*n* – 1)
  - Multiplying that result by 2 will cost 1 instruction

  Thus, the total time *T*1(*n*) can be expressed as:

  $$T1(n) = 1 + T1(n-1) + 1 = T1(n-1) + 2 \quad \text{// Equation 15.1}$$

The preceding equation is called a recurrence relation between *T*1(*n*) and *T*1(*n* – 1) because T1(*n*) is expressed as a function of *T*1(*n* – 1).

Computing $T(n)$ from a Recurrence Relation

- From there, we can use a number of techniques to compute the value of $T1(n)$ as a function of n.
- Three such techniques are:
  - Handwaving method
  - Iterative method
  - Proof by induction

Handwaving Method

- This is more of an estimation method rather than a method based on strict mathematics.
- From the preceding recurrence relation, we can say that it costs us two instructions to go down one step (from n to $n - 1$). Therefore, to go down n steps will cost us $2 * n$ instructions. We then add one instruction for $T(0)$, and get:

$$T1(n) = 2 * n + 1$$

Iterative Method (1 of 3)

This method involves iterating several times, starting with the recurrence relation until we can identify a pattern.

Generally, we can say:

$T1(x) = T1(x - 1) + 2$, where x is some integer.

(This is the recurrence relation we developed with x substituted for n.)

We now want to express $T(n)$ as a function of $T(n - 2)$; thus, we replace $T(n - 1)$ by an expression using $T(n - 2)$. Substituting $n - 1$ for x, we get

$$T1(n - 1) = T1(n - 2) + 2$$

Iterative Method (2 of 3)

Plugging in the value of $T1(n-1)$, we get

$$T1(n) = T1(n-2) + 2 + 2$$
$$= T1(n-2) + 2 * 2 \quad \text{// Equation 15.3}$$

- Note that we do not simplify *2 * 2*. In this way, we are trying to let a pattern develop so we can easily identify it.

Using $x = n - 2$ in our original equation, we get

$$T1(n-2) = T1(n-3) + 2$$

Plugging the value of $T1(n-2)$ into Equation 15.3, we get

$$T1(n) = T1(n-3) + 2 + 2 * 2$$
$$= T1(n-3) + 2 * 3 \text{ // Equation 15.4}$$

Using $x = n - 3$ in our original equation, we get

$$T1(n-3) = T1(n-4) + 2$$

Iterative Method (3 of 3)

Plugging the value of $T1(n-3)$ into Equation 15.4, we get

$$T1(n) = T1(n-4) + 2 + 2 * 3$$
$$= T1(n-4) + 2 * 4$$

Now we can see the pattern as follows:

$T1(n) = T1(n-k) + 2 * k$, where k is an integer between 1 and n // Equation 15.5

Plugging $k = n$ into Equation 15.5, we get

$$T1(n) = T1(0) + 2 * n$$
$$= 1 + 2 * n$$
$$= 2 * n + 1$$

This is the same result as from our handwaving method.

SOFTWARE ENGINEERING TIP

When trying to develop and identify a pattern using iteration, do not precisely compute all the terms. Instead, leave them as patterns.

Proof by Induction (1 of 3)

If we can guess the value of $T1(n)$ as a function of n, then we can use a proof by induction to prove that our guess is correct.

We can use the preceding iteration method to come up with a guess for $T1(n)$.

Generally, a proof by induction works as follows:

- Verify that our statement is true for a base case.
- Assume that our statement is true up to n.
- Prove that it is true for $n + 1$.

Proof by Induction (2 of 3)

Our guess is $T1(n) = 2 * n + 1$

Step 1: Verify that our guess is correct for T1(0).

$$T1(0) = 2 * 0 + 1$$
$$= 1$$

Thus, our guess is correct for $T1(0)$.

Step 2: Assume that $T1(n) = 2 * n + 1$.

Proof by Induction (3 of 3)

Step 3: Prove that $T1(n + 1) = 2 * (n + 1) + 1$

Plugging $x = n + 1$ into Equation 15.2, we get

$$T1(n + 1) = T1(n) + 2$$

Then, using our assumption

and replacing $T1(n)$ by $2 * n + 1$, we get

$$T1(n + 1) = 2 * n + 1 + 2$$
$$= 2 * n + 2 + 1$$
$$= 2 * (n + 1) + 1$$

Thus, we just proved, by induction, that our guess $T1(n) = 2 * n + 1$ is correct.

So, *powerOf2A*'s running time is $O(n)$.

Second Recursive Method

The second recursive method is designed using:

Base case: when $n = 0$, $2^0 = 1$

General case: $2^n = 2^{n-1} + 2^{n-1}$

The code becomes:

```
public static int powerOf2B( int n ) // n >= 0
{
   if ( n == 0 )
     return 1;
   else
     return powerOf2B( n - 1 )
            + powerOf2B( n - 1 );
}
```

Define the Recurrence Relation

Let's compute the running time of *powerOf2B* as $T2(n)$.

- In the base case ($n = 0$), *powerOf2B* makes only one comparison.

  Thus, $T2(0) = 1$.

- Because it takes $T2(n)$ to compute and return *powerOf2B(n)*, it takes $T2(n - 1)$ to compute and return *powerOf2B(n – 1)*. Thus, in the general case:
  - The *if* statement will cost 1 instruction
  - Computing and returning *powerOf2B(n – 1)* will cost $T2(n - 1)$
  - Doing it a second time will also cost $T2(n - 1)$
  - Adding the two results will cost 1 instruction

  Thus, the total time $T2(n)$ can be expressed as:

$$T2(n) = 1 + T2(n - 1) + T2(n - 1) + 1$$
$$= 2 * T2(n - 1) + 2 \text{ // Equation 15.6}$$

We now have a recurrence relation between $T2(n)$ and $T2(n - 1)$.

Evaluation of *powerOf2B* Method (1 of 4)

Using the iterative method:

Substituting x for n, we can rewrite Equation 15.6 as follows:

$$T2(x) = 2 * T2(x - 1) + 2 \text{ // Equation 15.7}$$

Using $x = n - 1$ in Equation 15.7, we get

$$T2(n - 1) = 2 * T2(n - 2) + 2$$

Plugging the value of $T2(n - 1)$ into Equation 15.6, we get

$$T2(n) = 2 * (2 * T2(n - 2) + 2) + 2$$
$$= 2^2 * T2(n - 2) + 2^2 + 2 \text{ // Equation 15.8}$$

Again, we leave $2^2 + 2$ as an expression to try to let a pattern develop.

Evaluation of *powerOf2B* Method (2 of 4)

Using $x = n - 2$ in Equation 15.7, we get

$$T2(n-2) = 2 * T2(n-3) + 2$$

Plugging the value of $T2(n-2)$ into Equation 15.8, we get

$$T2(n) = 2^2 * (2 * T2(n-3) + 2) + 2^2 + 2$$
$$= 2^3 * T2(n-3) + 2^3 + 2^2 + 2 \text{ // Equation 15.9}$$

Using $x = n - 3$ in Equation 15.7, we get

$$T2(n-3) = 2 * T2(n-4) + 2$$

Plugging the value of $T2(n-3)$ into Equation 15.9, we get

$$T2(n) = 2^3 * (2 * T2(n-4) + 2) + 2^3 + 2^2 + 2$$
$$= 2^4 * T2(n-4) + 2^4 + 2^3 + 2^2 + 2 \text{ // Equation 15.10}$$

Evaluation of *powerOf2B* Method (3 of 4)

Now we can see the pattern as follows:

$T2(n) = 2^k * T2(n - k) + 2^k + 2^{k-1} + \ldots + 2^2 + 2,$

where k is an integer between 1 and n // Equation 15.11

Noting that

$$2^k + 2^{k-1} + \ldots + 2^2 + 2$$
$$= -1 + 2^k + 2^{k-1} + \ldots + 2^2 + 2 + 1$$
$$= -1 + (2^{k+1} - 1) / (2 - 1)$$
$$= 2^{k+1} - 2$$

Equation 15.11 becomes

$T2(n) = 2^k * T2(n - k) + 2^{k+1} - 2$, where k is an integer between 1 and n // Equation 15.12

Evaluation of *powerOf2B* Method (4 of 4)

Plugging $k = n$ into Equation 15.12 to reach the base case of $T2(0)$, we get

$$\begin{aligned} T2(n) &= 2^n * T2(0) + 2^{n+1} - 2 \\ &= 2^n * 1 + 2^{n+1} - 2 \\ &= 3 * 2^n - 2 \\ &= O(2^n) \end{aligned}$$

Thus, *powerOf2A* runs in $O(n)$ while *powerOf2B* runs in $O(2^n)$, although they perform the same function.

As a result, computing 2^{20} using *powerOf2A* will cost 20 statement executions, while computing 2^{20} using *powerOf2B* will cost 1 million statement executions!

This simple example shows that how we code a method can have a significant impact on its running time.

Searching and Sorting

- As we did for the sequential search algorithm earlier, for search and sorting algorithms, we look at three cases:
  - Best case, the shortest running time
  - Worst case, the longest running time
  - Average case, the average running time

Let's start with a recursive binary search algorithm.

Recursive Binary Search

```
public static int recursiveBinarySearch ( int [ ] arr,
                        int key, int start, int end )
{
   if ( start <= end )
   {
      // look at the middle element of the subarray
      int middle = ( start + end ) / 2;
      if ( arr[middle] == key ) // found key, base case
          return middle;
      else if ( arr[middle] > key ) // look lower
          return recursiveBinarySearch( arr, key,
                                        start, middle - 1 );
      else // look higher
          return recursiveBinarySearch( arr, key,
                                        middle + 1, end );
   }
   else // key not found, base case
      return -1;
}
```

Statement Execution Analysis (1 of 2)

General case:

- The comparison of the first *if* statement will cost one instruction.
- The assignment statement will cost two instructions.
- The comparison in the second *if* statement will cost one instruction.
- The comparison in the *else/if* statement will cost one instruction.
- Computing and returning *recursiveBinarySearch(arr, key, start, middle* – 1*)* or *recursiveBinarySearch(arr, key, middle* + 1*, end)* will cost $T(n/2 - 1)$ or $T(n/2)$ instructions.

Note that only one recursive call will be made.

Thus, the total time $T(n)$ can be expressed as follows:

$$T(n) = 1 + 2 + 1 + 1 + T(n / 2)$$
$$= T(n / 2) + 5 \text{ // Equation 15.13}$$

Statement Execution Analysis (2 of 2)

- Base case (*n* is equal to 1)

  recursiveBinarySearch makes only

  – The first comparison

  – One addition

  – One division

  – The second comparison

  and returns the index of the found element or –1.

  Thus, $T(1) = 5$.

Recursive Binary Search Analysis

- Best case: we find the key in the middle at our first comparison.
  - Best case running time is $O(1)$.

Worst Case: We Do Not Find the Key (1 of 4)

Using the iteration method to compute the value of $T(n)$ as a function of n:

Substituting x for n, we can rewrite Equation 15.13 as follows:

$$T(x) = T(x / 2) + 5 \qquad \text{// Equation 15.14}$$

Using $x = n / 2$ in Equation 15.14, we get

$$T(n / 2) = \mathrm{T}((n / 2) / 2) + 5$$
$$= \mathrm{T}(n / 2^2) + 5$$

Worst-Case Running Time (2 of 4)

Plugging the value of $T(n / 2)$ into Equation 15.13, we get

$$T(n) = (T(n / 2^2) + 5) + 5$$
$$= T(n / 2^2) + 5 * 2 \qquad \text{// Equation 15.15}$$

Using $x = n / 2^2$ in Equation 15.14, we get

$$T(n / 2^2) = T((n / 2^2) / 2) + 5$$
$$= T(n / 2^3) + 5$$

Plugging the value of $T(n / 2^2)$ into Equation 15.15, we get

$$T(n) = (T(n / 2^3) + 5) + 5 * 2$$
$$= T(n / 2^3) + 5 * 3 \qquad \text{// Equation 15.16}$$

Using $x = n / 2^3$ in Equation 15.14, we get

$$T(n / 2^3) = T((n / 2^3) / 2) + 5$$
$$= T(n / 2^4) + 5$$

Worst-Case Running Time (3 of 4)

Plugging the value of $T(n / 2^3)$ into Equation 15.16, we get

$$T(n) = (T(n / 2^4) + 5) + 5 * 3$$
$$= T(n / 2^4) + 5 * 4 \quad \text{// Equation 15.17}$$

Now we can see the pattern as follows:

$$T(n) = T(n / 2^k) + 5 * \text{k},$$

where k is an integer between 1 and log n // Equation 15.18

We now want to choose k such that $n / 2^k$ is equal to 1 in order to reach our base case. If $n / 2^k = 1$, then $n = 2^k$ and taking the log of each side:

$$\log n = \log 2^k$$
$$= k \log 2$$
$$= k * 1$$
$$= k$$

Worst-Case Running Time (4 of 4)

Plugging $k = \log n$ into Equation 15.18, we get

$$T(n) = T(1) + 5 * \log n$$
$$= 5 + 5 * \log n$$
$$= O(\log n)$$

Average Case

- We find the key after performing half the number of possible comparisons.
- Thus, the average case running time is also $O(\log n)$.

Insertion Sort Running Time

```
public static void insertionSort( int [] array )
{
   int j, temp;
   for ( int i = 0; i < array.length; i++ )
   {
      j = i;
      temp = array[i];
      while ( j != 0 && array[j - 1] > temp )
      {
         array[j] = array[j - 1];
         j--;
      }
      array[j] = temp;
   }
}
```

Analysis: Insertion Sort Best Case

- The *for* loop header executes $n + 1$ times.
- The body of the *for* loop executes n times.
- In the best case, the array is already sorted.
  - The *while* loop condition will always evaluate to *false*, and we never execute the *while* loop body.
  - Inside the *for* loop, the three statements and the loop condition will each execute once for each iteration of the *for* loop, thus executing a total of

    $4 * n$ times.

So, the best-case running time is $O(n)$.

Insertion Sort Worst Case

- In the worst case, the array is sorted in the opposite order.
- The *while* loop condition is *true* at each first evaluation, and we enter the *while* loop every time we iterate the *for* loop.
  - Thus, the two statements inside the *while* loop each execute

    $(1 + 2 + 3 + 4 + \ldots (n - 1))$

    times.

Because $(1 + 2 + 3 + 4 + \ldots + (n - 1)) = n * (n - 1) / 2$, the worst-case running time of insertion sort is $O(n^2)$.

Insertion Sort Average Case

- In the average case, we will enter the *while* loop half the times we try.
- The average case is still $O(n^2)$.

Merge Sort Recursive Pseudocode

If the array has only one element, it is already sorted. Thus, do nothing; otherwise:

- Merge sort the left half of the array.
- Merge sort the right half of the array.
- Merge the two sorted half-arrays into one sorted array.

The last operation involves looping through all the elements of the two half-arrays; making it $O(n)$.

Thus, we can derive the following recursive formulation for the running time of Merge Sort:

$$T(n) = T(n / 2) + T(n / 2) + n$$
$$= 2\ T(n / 2) + n$$

Using the Iterative Method

Using derivation, we get:

$$T(n) = 2\ T(n / 2) + n$$
$$= 2\ (2\ T(n / 2^2) + n / 2) + n$$
$$= 2^2\ T(n / 2^2) + 2\ n$$

Continuing to iterate:

$$T(n) = 2^2\ T(n / 2^2) + 2\ n$$
$$= 2^2\ (2\ T(n / 2^3) + n / 2^2) + 2\ n$$
$$= 2^3\ T(n / 2^3) + 3\ n$$
$$T(n) = 2^3\ T(n / 2^3) + 3\ n$$
$$= 2^3\ (2\ T(n / 2^3) + n / 2^3) + 3\ n$$
$$= 2^4\ T(n / 2^4) + 4\ n$$

Continuing the Iteration

Thus, we identify the general pattern:

$$T(n) = 2^k\, T(n / 2^k) + k\, n$$

Choosing k so that $n / 2^k = 1$ in order to reach the base case, that is, $n = 2^k$, that is,

$k = \log n$, we get

$$T(n) = n\, T(1) + n \log n$$

$$= O(n \log n)$$

So Merge Sort is $O(n \log n)$. It is the same for best-case, worst-case, and average-case scenarios.

Thus, Merge Sort is better than Insertion Sort, Bubble Sort, and Selection Sort.