Analysis of Algorithms

Introduction

The focus of this module is mathematical aspects of algorithms. Our main focus is analysis of algorithms, which means evaluating efficiency of algorithms by analytical and mathematical methods. We start by some simple examples of worst-case and average-case analysis. We then discuss the formal definitions for asymptotic complexity of algorithms, used to characterize algorithms into different classes. And we present examples of asymptotic analysis of algorithms.

The next module deals with recursive algorithms, their correctness proofs, analysis of algorithms by recurrence equations, and algorithmic divide-and-conquer technique.

Contents

Worst-case and Average-Case Analysis:
Introductory Examples

Sequential Search
Finding Max and Min

Definitions of Asymptotic Complexities

$O()$, $\Omega()$, and $\Theta()$

Sum Rule and Product Rule

Example of Analysis: A Nested Loop

Insertion Sort Algorithm

Worst-Case Analysis of Insertion Sort

Average-Case Analysis of Insertion Sort
Worst-case and Average-Case Analysis: Introductory Examples

Sequential Search Algorithm

Suppose we have an array of \( n \) elements \( A[1:n] \), and a key element, \( KEY \). The following program is a simple loop which goes through the array sequentially until it either finds the key, or determines that it is not found. (dtype is the type declaration for the array.)

```c
int SequentialSearch (dtype A[ ], int n, dtype KEY) {
    for (i = 1 to n) {
        if (KEY == A[i])
            return (i);
    }
    return (-1); //The for-loop ended and the key was not found.
}
```

Let us analyze the number of key-comparisons (highlighted) in this algorithm. By counting this dominant operation, basically we are counting the number of times the loop is executed. There are two common ways of doing the analysis:

- **Worst-case analysis:** This determines the maximum amount of time the algorithm would ever take. This analysis is usually easier than the average-case analysis. At the same time, it is hoped that it is a good reflection of overall performance.
- **Average-case analysis:** This method determines the overall average performance. It considers all possible cases, assigns a probability to each case, and computes the weighted average (called expected value) for the random variable (in this case, the number of key-comparisons).

Worst-Case Analysis of Sequential Search

The worst-case number of key-comparison in this algorithm is obviously \( n \). This happens if the key is found in the last position of the array, or if it is not found anywhere.

Since each iteration of the for-loop takes at most some constant amount of time, \( C \), then the total worst-case time of the algorithm is

\[
T(n) \leq Cn + D.
\]

(The constant \( D \) represents the maximum amount of time for all statements that are executed only once, independent of the variable \( n \).) This total time is characterized as “order of” \( n \), denoted as \( O(n) \). (We will shortly see the formal definition for the order.)
Average-Case Analysis of Sequential Search

Now let us compute the average number of key-comparisons. As a first estimate, one may think that since the worst-case number is \( n \), and the best-case is 1 (found right away), then the average must be about \( n/2 \). Let us do a careful analysis and see how good this estimate is.

First, as a quick review the "expected value", suppose a random variable has the possible values \{1,2,3\} with the following probabilities.

<table>
<thead>
<tr>
<th>Value of the random variable</th>
<th>Probability ( p_r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
</tr>
<tr>
<td>3</td>
<td>0.8</td>
</tr>
</tbody>
</table>

Then the expected value of \( r \) is

\[
1 \times 0.1 + 2 \times 0.1 + 3 \times 0.8 = 2.7
\]

We may also refer to this as the weighted average. Note that a straight average (when there is no probability involved) would be simply \((1 + 2 + 3)/3 = 2\).

Now, to compute the expected value of the number of key-comparisons for the algorithm, let

\[
P = \text{Probability that the key is found somewhere in the array}
\]

and let

\[
P_i = \text{Probability that the key is found in position } i \text{ of the array, } 1 \leq i \leq n.
\]

Assuming that the array is random, a common assumption is that when the key is found, then it is equally likely that it is found in any of the \( n \) positions. So,

\[
P_i = \frac{p}{n}, \quad \forall i.
\]

Finally, the probability that the key is not found is

\[
Q = 1 - P
\]
So, the expected number of key-comparisons in this algorithm is:

\[
 f(n) = \sum_{i=1}^{n} P_i \cdot i + Q \cdot n
\]

\[
 = \sum_{i=1}^{n} \frac{P}{n} \cdot i + (1 - P) \cdot n
\]

\[
 = \frac{P}{n} \cdot \sum_{i=1}^{n} i + (1 - P) \cdot n
\]

\[
 = \frac{P}{n} \cdot \frac{n(n + 1)}{2} + (1 - P) \cdot n
\]

\[
 = P \cdot \frac{n + 1}{2} + (1 - P) \cdot n
\]

In the special case when \( P = 1 \), the expected number of comparisons is \( \frac{n+1}{2} \), which agrees with our initial estimate. Otherwise, there is an additional term that takes into account the additional event when the key is not found anywhere in the array.

**Finding Max and Min of an Array**

The following pseudocode is a simple program loop that finds the maximum and minimum elements in an array of \( n \) elements, \( A[1 : n] \). (\text{Max} and \text{Min} are the returned parameters.)

```plaintext
FindMaxMin (dtype A[], int n, dtype Max, dtype Min) {
    Max = A[1]; Min = A[1];
    for i = 2 to n {
        if [A[i] > Max]
            Max = A[i];
        else if [A[i] < Min]
            Min = A[i];
    }
}
```

In iteration \( i \) of the for-loop, \( A[i] \) is first compared against \( \text{Max} \). If \( A[i] \) is greater, then \( \text{Max} \) is updated. Otherwise, a second comparison is made against \( \text{Min} \), and if \( A[i] \) is smaller, then \( \text{Min} \) is updated.

Let us analyze the worst-case, best-case, and average-case number of key-comparisons. (The key comparisons are highlighted.)
**Worst-Case and Best-Case Analysis**

In the worst-case, every iteration of the loop makes two comparisons. (This happens if the first element of the array has the largest value.) So the worst-case number of comparisons is $2(n - 1)$. In the best-case, every iteration makes only one comparison, so the best-case number of comparisons is $(n - 1)$. This happens if the input is in sorted order.

**Average-Case Analysis**

The number of comparisons in each iteration of the loop is 2 in the worst-case, and 1 in the best-case. So is it a good estimate to figure that on the average, the number is 1.5 per iteration?! No, the average is not always half-way between the worst and the best. (If I buy a lottery ticket, do I have a 50-50 chance of winning the jackpot?!) We will prove that it is much more likely to make two comparisons per iteration. As a result, the expected number of comparisons is very close to the worst-case number.

Let us assume the array is random and the elements are all distinct. Iteration $i$ of the loop compares element $A[i]$ against the current $Max$ and possibly $Min$. Let us define the *prefix sequence* of $A[i]$ as the sequence of elements in the array starting with $A[1]$ and ending with $A[i]$ itself.

$$A[1], A[2], \ldots, A[i]$$

Since the array is random, element $A[i]$ is equally likely to be the smallest in its prefix, or the second smallest, or third smallest, ⋯, or the largest. So, the probability that $A[i]$ is the largest in its prefix sequence is

$$\frac{1}{i}$$

and the probability that $A[i]$ is not the largest in its prefix sequence is

$$\frac{i - 1}{i}$$

If $A[i]$ is the largest in its prefix, iteration $i$ makes only one comparison. Otherwise, iteration $i$ makes two comparisons. Therefore, the expected number of comparisons is

$$f(n) = \sum_{i=2}^{n} \left( \frac{1}{i} \cdot 1 + \frac{i - 1}{i} \cdot 2 \right) = \sum_{i=2}^{n} \left( 2 - \frac{1}{i} \right) = 2(n - 1) - \sum_{i=2}^{n} \frac{1}{i} = 2n - 1 - \sum_{i=1}^{n} \frac{1}{i}$$

The latter summation is known as the Harmonic series $H_n$, and the value of the sum is approximately $\ln n$. (\(\ln\) is the natural log.)
\[H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n\]

Therefore,

\[f(n) \approx 2n - 1 - \ln n.\]

Since the value of the log is negligible compared to \(2n\), the expected number of comparisons is indeed very close to the worst-case value, as stated earlier. For example, if \(n = 1000\), the expected number of key-comparison is about 1992, and the worst-case number \((2n - 2)\) is 1998.

(Note: Homework problems explore a technique for finding approximate value of a summation by converting the summation into integral. This technique may be used to find the approximate value for the Harmonic sum.)

**Definitions of Asymptotic Complexity**

When we study the running time of an algorithm, our focus is on the performance when the problem size gets large. This is because the performance for small problem sizes hardly matters since the running time will be very small anyway.

Suppose there are two algorithms for a problem of size \(n\) with the running times, respectively

\[T_1(n) = 10n,\]
\[T_2(n) = 2n^2\]

Which one of the two is faster (smaller) running time? Let’s tabulate these two functions for some values of \(n\).

<table>
<thead>
<tr>
<th>(n)</th>
<th>(10n)</th>
<th>(2n^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>100</td>
<td>1,000</td>
<td>20,000</td>
</tr>
<tr>
<td>1,000</td>
<td>10,000</td>
<td>2,000,000</td>
</tr>
<tr>
<td>10,000</td>
<td>100,000</td>
<td>200,000,000</td>
</tr>
<tr>
<td>100,000</td>
<td>(1 \times 10^6)</td>
<td>(2 \times 10^{10})</td>
</tr>
</tbody>
</table>
We observe that initially, for $n < 5$, $T_1$ is larger than $T_2$. The two equal at $n = 5$. And after that, as $n$ gets larger, $T_2$ gets much larger than $T_1$. This may also be observed pictorially, by looking at the graphs of these functions (time-versus-$n$).

![Graph showing $T_1$ (Linear, with slope of 10) and $T_2$ (quadratic)](image)

The quadratic function $T_2$ starts smaller than the linear one $T_1$. The two cross at $n = 5$. After that, $T_2$ starts growing much faster than $T_1$. As $n$ gets larger and larger, $T_2$ gets much larger than $T_1$. We say that $T_2$ has a faster growth rate than $T_1$, or that $T_2$ is asymptotically larger than $T_1$.

The fact that $T_1$ has a slower growth rate than $T_2$ is due to the fact that $T_1$ is a linear function $n$ and $T_2$ is a quadratic function $n^2$. The coefficients (also called constant factors) are not as critical in this comparison. For example, suppose we have a different pair of coefficients:

$T_1(n) = 50n,$
$T_2(n) = n^2$

The cross point between the two functions now is $n = 50$, and after that $T_2$ starts growing much faster than $T_1$ again.

So, the asymptotic complexity definitions incorporate two issues:

1. Focus on large problem size (large $n$), and ignore small values of $n$.
2. Ignore the constant coefficients (constant factors).

We are now ready to make the formal definition.

**Definition 1: $O(\ )$ Upper Bound**

Suppose there are positive constants $C$ and $n_0$ such that

$T(n) \leq C \cdot f(n), \quad \forall n \geq n_0$

Then we say $T(n)$ is $O(f(n))$.

The $O(\ )$ is read as “order of”, or “big oh” of.
**Example**: Prove the following function is $O(n^2)$.

$$T(n) = 5n^2 + 10n + 100$$

**Proof**: Intuitively, when $n$ gets large, the total value of this polynomial is close to $5n^2$ because the remaining terms become negligible in comparison. Now, let us formally prove that $T(n)$ is $O(n^2)$ by finding constants $C$ and $n_0$ such that $T(n) \leq Cn^2, \forall n \geq n_0$.

$$T(n) = 5n^2 + 10n + 100$$
$$\leq 5n^2 + 10n\left(\frac{n}{100}\right) + 100\left(\frac{n}{100}\right)^2, \quad n \geq 1$$
$$\leq 115n^2, \quad n \geq 1$$

This satisfies the definition and proves $T(n)$ is $O(n^2)$. (Here $C = 115$ and $n_0 = 1$.) But, to satisfy our intuition, how do we find the constant $C$ closer to 5? By picking a larger $n_0$. Let’s arbitrarily pick $n \geq 100$, so $\left(\frac{n}{100}\right) \geq 1$.

$$T(n) = 5n^2 + 10n + 100$$
$$\leq 5n^2 + 10n \left(\frac{n}{100}\right) + 100 \left(\frac{n}{100}\right)^2, \quad n \geq 100$$
$$\leq 5.11n^2, \quad n \geq 100.$$ 

**Example**: Prove the following polynomial is $O(n^3)$.

$$T(n) = 5n^3 - 10n^2 + 20n - 100$$

**Proof**: Let’s arbitrarily pick $n \geq 10$, so that $\left(\frac{n}{10}\right) \geq 1$.

$$T(n) = 5n^3 - 10n^2 + 20n - 100 \quad (\text{discard negative terms})$$
$$\leq 5n^3 + 20n,$$
$$\leq 5n^3 + 20n \left(\frac{n}{10}\right)^2, \quad n \geq 10$$
$$\leq 5.2n^3, \quad n \geq 10.$$ 

Next, we make the following definition for the lower bound, which is symmetrical to $O(\cdot)$.

**Definition 2**: $\Omega(\cdot)$ **Lower Bound**
Suppose there are positive constants $C$ and $n_0$ such that

$$T(n) \geq C \cdot f(n), \quad \forall n \geq n_0$$

Then we say $T(n)$ is $\Omega(f(n))$. 

---

8
**Example:** Prove the following polynomial is $\Omega(n^3)$.

$$T(n) = 5n^3 - 10n^2 + 20n - 100$$

**Proof:** We must show that $T(n) \geq Cn^3, \forall n \geq n_0$ for some positive constants $C, n_0$. Here, we need to pick $n_0$ carefully so that $C$ becomes positive.

$$T(n) = 5n^3 - 10n^2 + 20n - 100 \quad \text{(discard smaller positive terms)}$$

$$\geq 5n^3 - 10n^2 - 100$$

$$\geq 5n^3 - 10n^2 \left(\frac{n}{100}\right) - 100 \left(\frac{n}{100}\right)^3, \quad n \geq 100$$

$$\geq 4.8999 \, n^3, \quad n \geq 100.$$

---

**Definition 3:** $\Theta(\cdot)$ Tight Bound

Suppose there are positive constants $C_1, C_2, n_0$ such that

$$C_1 f(n) \leq T(n) \leq C_2 f(n), \quad \forall n \geq n_0$$

That is, $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$. Then, we say that $T(n)$ is $\Theta(f(n))$.

---

**Example:** We proved the following polynomial is both $O(n^3)$ and $\Omega(n^3)$.

$$T(n) = 5n^3 - 10n^2 + 20n - 100$$

Therefore, $T(n)$ is $\Theta(n^3)$.

**Example:** Suppose

$$2n \leq T(n) \leq 5n^2$$

In this case, $T(n)$ is $\Omega(n)$ and $O(n^2)$. But this function does not have a tight bound.

**Example:** Prove the following summation is $\Theta(n^2)$, without the use of arithmetic sum formula, but rather by manipulating the terms to find the needed upper bound and lower bound.

$$S(n) = 1 + 2 + 3 + \cdots + n$$

1. **Prove $O(n^2)$:**

$$S(n) = 1 + 2 + \cdots + n$$

$$\leq n + n + \cdots + n$$

$$\leq n^2.$$
2. Prove $\Omega(n^2)$:

In the above proof for the upper bound, we raised all terms to the largest term. If we try to mimic that approach and lower all terms to the smallest term, we get $S(n) \geq 1 + 1 + \cdots + 1 = n$, which will not give the desired lower bound. Instead, we will first discard the first half of the terms, and then lower all terms to the smallest.

\[
S(n) = 1 + 2 + \cdots + n = \sum_{i=1}^{n} i \\
\geq \sum_{i=\left\lfloor n/2 \right\rfloor}^{n} i \\
\geq \sum_{i=\left\lfloor n/2 \right\rfloor}^{n} \left\lfloor n/2 \right\rfloor \\
\geq \left\lfloor n/2 \right\rfloor \cdot \left\lfloor n/2 \right\rfloor \geq \frac{n^2}{4}
\]

We proved $\frac{n^2}{4} \leq S(n) \leq n^2$. Therefore, $S(n)$ is $\Theta(n^2)$.

**Sum-Rule and Product-Rule**

<table>
<thead>
<tr>
<th>Sum Rule for $O()$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suppose $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$.</td>
</tr>
<tr>
<td>Then,</td>
</tr>
<tr>
<td>$T_1(n) + T_2(n)$ is $O(f(n) + g(n))$.</td>
</tr>
</tbody>
</table>

**Proof:** we first express $T_1(n)$ and $T_2(n)$ in terms of the definitions of $O()$:

\[
T_1(n) \leq C_1 f(n), \quad \forall n \geq n_1 \\
T_2(n) \leq C_2 g(n), \quad \forall n \geq n_2
\]

For $n \geq \max\{n_1, n_2\}$, the two inequalities will both hold. So,

\[
T_1(n) + T_2(n) \leq C_1 f(n) + C_2 g(n), \quad n \geq \max\{n_1, n_2\} \\
\leq Cf(n) + Cg(n), \quad C = \max\{C_1, C_2\} \\
\leq C(f(n) + g(n)).
\]

(Note: The sum rule similarly applies to $\Omega$ and to $\Theta$.)
**Application of sum rule:** Suppose a program has two parts: Part 1, which is executed first, and then Part 2.

<table>
<thead>
<tr>
<th>Part 1 : Time $T_1(n)$ suppose $O(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part 2: Time $T_2(n)$ suppose $O(n^2)$</td>
</tr>
</tbody>
</table>

Then, the total time is $O(n + n^2)$, which is $O(n^2)$.

**Product Rule for $O(\cdot)$**

If $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$, then,

$$T_1(n) \times T_2(n) \text{ is } O(f(n) \times g(n)).$$

**Proof:** Similar to the above proof; left to the student.

*(Note: The product rule similarly applies to $\Omega$ and to $\Theta$.)*

An application of the product rule is in **nested loops**, as in the following example.

**Example of Analysis: A Nested Loop**

Let’s analyze the running time of the following program (nested loops).

```
C = 0
for i = 1 to n + 1
    for j = i to 3n - 1
        C = C + 1
Inner loop number of times = (3n - 1) - (i) + 1 = 3n - i
```

**Method 1 (Detailed Analysis):** Let us find the total number of times the innermost statement ($C = C + 1$) is executed. That is, find the final value for $C$. Let $F(n)$ denote this final count. We find this count by the following double summation.

$$F(n) = \sum_{i=1}^{n+1} \sum_{j=i}^{3n-1} 1$$
The inner summation for \( j \) is \( 1 + 1 + \cdots + 1 \), so we find the count by:

\[
\text{Upper limit of summation} - \text{lower limit of summation} + 1 = (3n - 1) - (i) + 1 = 3n - i.
\]

And the outer summation is arithmetic sum, so we apply the formula for it.

\[
F(n) = \sum_{i=1}^{n+1} \sum_{j=i}^{3n-1} 1
\]

\[
= \sum_{i=1}^{n+1} (3n - i) \quad \text{(This is arithmetic sum)}
\]

\[
= (n + 1) \left( \frac{(3n - 1) + (3n - n - 1)}{2} \right)
\]

\[
= (n + 1) \left( \frac{5n - 2}{2} \right)
\]

\[
= \frac{5n^2 + 3n - 2}{2}
\]

The total number of times the innermost statement is executed is \( O(n^2) \), which means the total running time of the program is also \( O(n^2) \).

**Method 2 (Loop Analysis):** In this method, we don’t bother to find the exact total number of times the innermost statement is executed. Rather, we analyze the loops by using the sum-rule and product-rule.

- The inner loop is executed \( 3n - i \) times. Since the range of \( i \) values is 1 to \( n + 1 \), then we know \( 3n - i \) is \( O(n) \).
- The outer loop is executed \( n + 1 \) times, which is \( O(n) \).
- Therefore, by the product rule, the total running time is \( O(n^2) \).
Insertion Sort Algorithm

Suppose we have an array of \( n \) elements, \( A[0: n - 1] \). Insertion sort starts by sorting the first two elements. Then the third element is inserted into the sorted part, so that the first 3 elements are sorted. Then, the next element is inserted into the sorted portion, so that the first 4 elements become sorted, and so on. Let us first illustrate by a numerical example.

<table>
<thead>
<tr>
<th>7</th>
<th>5</th>
<th>6</th>
<th>2</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Sort the first two
Insert 6 into [5 7]
Insert 2 into [5 6 7]
Insert 4 into [2 5 6 7]
Insert 3 into [2 4 5 6 7]

Let's look at the details of each insertion phase. For example, let us see how the last insertion is carried out. At the start of this phase, the sorted part is [2 4 5 6 7] and element 3 needs to be inserted into the sorted part. This is done by a sequence of compare/swap operations between pairs, starting at the end with the pair [7 3].

<table>
<thead>
<tr>
<th>2</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>[7</th>
<th>3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cm/Swp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>[6</td>
<td>3]</td>
<td>7</td>
</tr>
<tr>
<td>Cm/Swp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>[5</td>
<td>3]</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Cm/Swp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[4</td>
<td>3]</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Cm/Swp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2</td>
<td>3]</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Compare</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

We present the pseudocode next, and then analyze both the worst-case and average-case time complexity.

```plaintext
Insertion Sort (datatype A[ ], int n) { //Input array is A[0: n − 1]
  for i = 1 to n − 1
    { // Insert A[i]. Everything to the left of A[i] is already sorted.
      j = i;
      while (j > 0 and A[j] < A[j - 1])
        {swap(A[j], A[j - 1]);
         j = j - 1;
        }
    }
}
```
Worst-Case Analysis of Insertion Sort

Method 1 (Find the worst-Case Exact Number of Operations): One way to analyze an algorithm is in terms of some dominant operation, in this case a key comparison. This is a comparison between a pair of elements in the array, which is highlighted in the above algorithm. By counting this operation, we are basically counting the number of times the inner loop (while loop) is executed. So, let \( f(n) \) be the worst-case number of key comparisons in this algorithm.

The worst-case number of key comparisons in the while loop is exactly \( i \), because the loop starts with \( j = i \) and in the worst case goes down to \( j = 1 \). (When \( j = 0 \), the comparison is skipped.) Therefore, the worst-case number of key-comparisons is:

\[
f(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2-n}{2}
\]

(Arithmetic sum formula was used to find the sum.) So, we conclude that the worst-case total time of the algorithm is \( O(n^2) \).

Method 2 (Analyze the Loops):

- The worst-case time of the inner while-loop is \( O(i) \), thus \( O(n) \).
  This is because the range of \( i \) values is 1 to \( n-1 \).
- The outer loop (for loop) is executed \( O(n) \) times.
- By the product rule, the total worst-case time of the algorithm becomes \( O(n^2) \).

Average-Case Analysis of Insertion Sort

We carry out the average case analysis in two ways, one in terms of the number of key comparisons, and the other in terms of the number of swaps.

Expected Number of Key-Comparisons: Consider the while loop, where element \( A[i] \) is inserted into the sorted part. If we assume the array is random, then \( A[i] \) has equal probability of being the largest, second largest, third largest, and so on. There are \( i + 1 \) cases, as listed in the table below. The table also shows the number of key-comparisons made by the while loop for each case. Note that the last two cases both make the worst-case number of key comparisons, which is \( i \).
If element $A[i]$ is:

<table>
<thead>
<tr>
<th>Event</th>
<th>Number of key comparisons made by the while-loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Largest</td>
</tr>
<tr>
<td>2</td>
<td>Second largest</td>
</tr>
<tr>
<td>3</td>
<td>Third Largest</td>
</tr>
<tr>
<td>$i - 1$</td>
<td>Third smallest</td>
</tr>
<tr>
<td>$i$</td>
<td>Second smallest</td>
</tr>
<tr>
<td>$i + 1$</td>
<td>Smallest</td>
</tr>
</tbody>
</table>

Since these $(i + 1)$ cases all have equal probabilities, the expected number of key comparisons made by the while loop is:

$$
\frac{(1 + 2 + 3 + \cdots + i) + i}{i + 1} = \frac{i(i + 1)}{2} + \frac{i}{i + 1} = \frac{i}{2} + \frac{i}{i + 1}
$$

(Since the value of last term is small, $\frac{i}{i+1} < 1$, the expected number of key comparisons made by the while loop is about $i/2$, which is half of the worst-case.) Therefore, the expected number of key-comparisons for the entire algorithm is

$$
\sum_{i=1}^{n-1} \frac{i}{2} + \frac{i}{i + 1} = \sum_{i=1}^{n-1} \frac{i}{2} + \sum_{i=1}^{n-1} \frac{i}{i + 1} = \frac{n(n - 1)}{4} + \sum_{i=1}^{n-1} \frac{i}{i + 1} < \frac{n(n - 1)}{4} + (n - 1)
$$

(The upper bound for the summation is a fairly good approximation. For example, for $n = 100$, the value of the summation becomes $\sum_{i=1}^{99} \frac{i}{i+1} \approx 95$.)

### Expected Number of Swaps

In the above table, we saw the last two cases both have $i$ comparisons. This non-uniformity resulted in a slight complication in the analysis. This complication is avoided by analyzing the expected number of swaps. Let us again look at the table of possible events.

<table>
<thead>
<tr>
<th>Event</th>
<th>Number of SWAPS made by the while-loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Largest</td>
</tr>
<tr>
<td>2</td>
<td>Second largest</td>
</tr>
<tr>
<td>3</td>
<td>Third Largest</td>
</tr>
<tr>
<td>$i - 1$</td>
<td>Third smallest</td>
</tr>
<tr>
<td>$i$</td>
<td>Second smallest</td>
</tr>
<tr>
<td>$i + 1$</td>
<td>Smallest</td>
</tr>
</tbody>
</table>
Since all \( i + 1 \) events have equal probability, the expected number of swaps for the while loop is

\[
\frac{0 + 1 + 2 + \cdots + i}{i + 1} = \frac{i}{2}
\]

So the expected number of swaps for the entire algorithm is

\[
\sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n - 1)}{2}
\]

**Alternative Analysis for the Expected Number of Swaps:**

Suppose the input sequence is a random ordering of integers 1 to \( n \). This analysis considers all \( n! \) possible input orderings, count the number of swaps for each case, and average it over all possible cases. (We hope this analysis provides additional insight on how the expected performance may be computed.)

Define a pair of elements \((A[i], A[j])\) in the array to be inverted (or out-of-order) if

\[(i < j) \text{ and } A[i] > A[j].\]

And define the total number of inversions in an array as the number of inverted pairs. One way to count the number of inversions is:

\[
\sum_{i=1}^{n-1} \text{Number of elements to the left of } A[i] \text{ which are greater than } A[i].
\]

From this formulation, it should be obvious that the number of inversions in a sequence is exactly equal to the number of swap operations made by the insertion-sort algorithm for that input sequence. Suppose there are \( k \) elements in the original input sequence to the left of \( A[i] \) with values greater than \( A[i] \). Then, at the start of the while loop for inserting \( A[i] \), these \( k \) elements will all be on the rightmost part of the sorted part, immediately to the left of \( A[i] \), as depicted below.

\[
\begin{array}{c}
\text{(Elements < } A[i]) \hspace{1cm} \text{followed by} \hspace{1cm} \text{(k elements > } A[i]) \\
\text{Sorted Part} \hspace{2cm} A[i] \hspace{2cm} \text{Element to be inserted}
\end{array}
\]

Then \( A[i] \) has to hop over the \( k \) elements in order to get to where it needs to be in the sorted part, which means exactly \( k \) swaps.
The following table shows an example for \( n = 3 \). There are \( n! = 6 \) possible input sequences. The number of inversions for each sequence is shown, as well as the overall average number of inversions.

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Number of Inversions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 2 3</td>
<td>0</td>
</tr>
<tr>
<td>2 1 3 2</td>
<td>1</td>
</tr>
<tr>
<td>3 2 1 3</td>
<td>1</td>
</tr>
<tr>
<td>4 2 3 1</td>
<td>2</td>
</tr>
<tr>
<td>5 3 1 2</td>
<td>2</td>
</tr>
<tr>
<td>6 3 2 1</td>
<td>3</td>
</tr>
</tbody>
</table>

Overall Average = \( \frac{9}{6} = 1.5 \)

The input sequences may be partitioned into pairs, such that each pair of sequences are reverse of each other. For example, the reverse of \([1\ 3\ 2]\) is the sequence \([2\ 3\ 1]\). The following table shows the pairing of the 6 input sequences.

<table>
<thead>
<tr>
<th>Pairs of input sequences which are reverse of each other</th>
<th>Number of inversions</th>
<th>Average number of inversions for each pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
<td>0</td>
<td>1.5</td>
</tr>
<tr>
<td>3 2 1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1 3 2</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>2 3 1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2 1 3</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>3 1 2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Overall Average: 1.5

In general, if a pair of elements is inverted in one sequence, then the pair is not inverted in the reverse sequence, and vice versa. For example:

- In the sequence \([1\ 3\ 2]\), the pair \((3, 2)\) is inverted
- In the reverse sequence, \([2\ 3\ 1]\), the pair \((2, 3)\) is not inverted.

This means that each pair of values is inverted in only one of the two sequences. So each pair contributes 1 to the sum of inversions in the two sequences. Therefore, the sum of inversions in each pair of reverse sequences is the number of pairs in a sequence of \( n \) elements, which is \( \frac{n(n-1)}{2} \). So, the average number of inversions for each pair of sequences is

\[
W(n) = \frac{n(n - 1)}{4}
\]
The overall average number of inversions is also $W(n)$. Therefore, the expected number of swaps in the algorithm is this exact number.

**Deducing the Expected Number of Comparisons**

It is interesting to note that from the expected number of swaps, we may arrive at the expected number of comparisons rather quickly. Every key-comparison made by the while loop is followed by a swap, except possibly the last comparison before the termination of the while loop. This happens if $(j > 0$ and $A[j] \geq A[j - 1])$, which means that the element being inserted is not the smallest in its prefix sequence. This has the probability $i/(i + 1)$, as indicated earlier. So, with this probability, there is a last comparison in the while loop which is not followed by a swap. Therefore, the expected number of comparisons is

$$W(n) + \sum_{i=1}^{n-1} \frac{i}{i + 1} = \frac{n(n - 1)}{4} + \sum_{i=1}^{n-1} \frac{i}{i + 1} < \frac{n(n - 1)}{4} + n - 1$$

which is the same as earlier.