Module 6: Analysis of Algorithms

Reading from the Textbook: Chapter 4 Algorithms

Introduction

The focus of this module is the 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, used to characterize algorithms into different classes. And we present examples of asymptotic analysis.

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 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.)

```
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, usually 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 of "expected value", suppose a random variable has the possible values $\{1,2,3\}$ with the following probabilities.

Value of the random variable r	Probability P_r
1	0.1
2	0.1
3	0.8

Then the *expected value* of r is

$$1 * 0.1 + 2 * 0.1 + 3 * 0.8 = 2.7$$

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

$P =$ Probability that the key is found somewhere in the array

and let

$P_i =$ Probability that the key is found in position i 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:

$$
\begin{aligned}
f(n) &= \sum_{i=1}^{n} P_i \cdot i + Q \cdot n \\
&= \sum_{i=1}^{n} \frac{P}{n} \cdot i + (1-P) \cdot n \\
&= \frac{P}{n} \cdot \sum_{i=1}^{n} i + (1-P) \cdot n \qquad \text{(Use arithmetic sum formula)} \\
&= \frac{P}{n} \cdot \frac{n(n+1)}{2} + (1-P) \cdot n \\
&= P \cdot \frac{n+1}{2} + (1-P) \cdot n
\end{aligned}
$$

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. For example, if $P = {}^{1}/_{2}$, then the expected number of comparisons becomes $\frac{3n+1}{4}$.

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, $A[1{:}n]$. (*Max* and *Min* are the returned parameters.)

```
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 Max. If $A[i]$ is greater, then Max is updated. Otherwise, a second comparison is made against Min, and if $A[i]$ is smaller, then 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.

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

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 $\ln n$. (Here, the logarithm is the natural log.)

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \cong \ln n$$

Therefore,

$$f(n) \cong 2n - 1 - \ln n.$$

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.

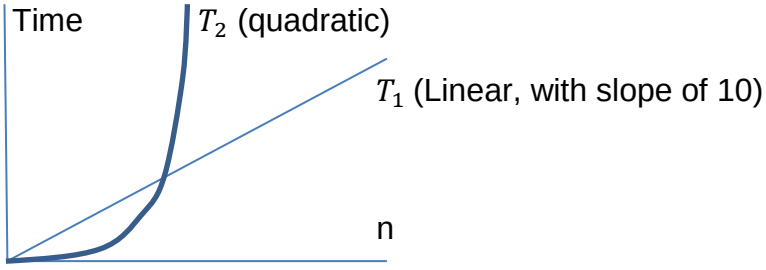
Suppose there are two algorithms for a problem of size *n* with the running times, respectively

$$T_1(n) = 10\ n,$$
$$T_2(n) = 2\ n^2$$

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

n	$10n$	$2n^2$
1	10	2
2	20	8
5	50	50
10	100	200
100	1,000	20,000
1,000	10,000	2,000,000
10,000	100,000	200,000,000
100,000	1×10^6	2×10^{10}

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*).



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:

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

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, we want the asymptotic complexity definitions to incorporate two issues:

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

Before making the formal definitions, we consider one more example.

$$T(n) = 2\, n^2 + 10\, n + 20$$

Let us see how this function behaves as n gets large, by tabulating the function for some increasing values of n.

n	$2\, n^2$	$10\, n$	20	$T(n)$	$T(n)/n^2$
1	2	10	20	32	32
10	200	100	20	320	3.20000
100	20,000	1,000	20	21,020	2.10200
1,000	2,000,000	10,000	20	2,010,020	2.01002
10,000	200,000,000	100,000	20	200,100,020	2.00100
100,000	20,000,000,000	1,000,000	20	20,001,000,020	2.00010

From the last column, observe that as n gets larger, the value of $T(n)$ gets closer to $2\, n^2$. But we cannot find any constant C where

$$T(n) = 2\, n^2 + 10\, n + 20 = Cn^2.$$

That is, the ratio $T(n)/n^2$ is not a constant, but a function of n.

$$\frac{T(n)}{n^2} = \frac{2\, n^2 + 10\, n + 20}{n^2}$$

However, we can express an upper bound for $T(n)$. For example, for all $n \geq 100$,

$$T(n) \leq 2.102\, n^2.$$

We are now ready to make the formal definitions.

Definition: O( ) Upper Bound
Suppose there are positive constants C and n_0 such that
$$T(n) \le C \cdot f(n), \qquad \forall n \ge n_0$$
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).

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

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, we formally prove that $T(n)$ is O(n^2) by finding positive constants C and n_0 such that $T(n) \le Cn^2, \forall n \ge n_0$.

$$\begin{aligned} T(n) &= 5\,n^2 + 10\,n + 100 \\ &\le 5n^2 + 10n\,(n) + 100\,(n^2), \quad n \ge 1 \\ &\le 115n^2, \qquad n \ge 1 \end{aligned}$$

This satisfies the definition and proves $T(n)$ is O(n^2). (Here $C = 115$ and $n_0 = 1$.)
But, to satisfy our intuition, let us find the constant *C* closer to 5 by picking a larger n_0.
Let's arbitrarily pick $n \ge 100$, so $\left(\frac{n}{100}\right) \ge 1$. Then,

$$\begin{aligned} T(n) &= 5\,n^2 + 10\,n + 100 \\ &\le 5n^2 + 10n\,\left(\frac{n}{100}\right) + 100\,\left(\frac{n}{100}\right)^2, \qquad n \ge 100 \\ &\le 5.11\,n^2, \quad n \ge 100. \end{aligned}$$

Example: Prove the following polynomial is O(n^4).

$$T(n) = 5n^4 - 10n^3 + 20n^2 - 50\,n + 100$$

Proof: First we need to get rid of the negative terms.

$$\begin{aligned} T(n) &= 5n^4 - 10n^3 + 20n^2 - 50\,n + 100 \\ &\le 5n^4 + 20n^2 + 100, \qquad n \ge 0. \end{aligned}$$

Now, since we have only positive terms, we may multiply the smaller positive terms by anything ≥ 1, as we did in our earlier example. Suppose we pick $n \ge 10$, so $\left(\frac{n}{10}\right) \ge 1$.

$$\begin{aligned} T(n) &\le 5n^4 + 20n^2 + 100 \\ &\le 5n^4 + 20\,n^2\left(\frac{n}{10}\right)^2 + 100\left(\frac{n}{10}\right)^4 \\ &\le 5.21\,n^4, \qquad n \ge 10. \end{aligned}$$

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

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

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

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

Example: Prove the following polynomial is $\Omega(n^4)$.

$$T(n) = 5n^4 - 10n^3 + 20n^2 - 50\,n + 100$$

Proof: We must show that $T(n) \geq Cn^4, \forall n \geq n_0$ for some positive constants C, n_0. Here, we need to pick n_0 carefully so that the constant C becomes positive.

$$\begin{aligned} &T(n) = 5n^4 - 10n^3 + 20n^2 - 50\,n + 100 \quad (\text{discard smaller positive terms}) \\ &\geq 5n^4 - 10n^3 - 50n \\ &\geq 5n^3 - 10n^3\left(\frac{n}{100}\right) - 50n\left(\frac{n}{100}\right)^3, \qquad n \geq 100 \\ &\geq 4.89995\,n^4, \quad n \geq 100. \end{aligned}$$

■

Definition: $\Theta(\,)$ 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 $\Omega\big(f(n)\big)$.
Then, we say that $T(n)$ is $\Theta(f(n))$.

Example: We proved the following polynomial is both $\mathrm{O}(n^4)$and $\Omega(n^4)$.
Therefore, $T(n)$ is $\Theta(n^4)$.

$$T(n) = 5n^4 - 10n^3 + 20n^2 - 50\,n + 100$$

Note: For the upper bound, we proved

$$T(n) \leq 5.21\,n^4, \qquad n \geq 10.$$

And for the lower bound, we proved

$$T(n) \geq 4.89995\,n^4, \quad n \geq 100.$$

The upper bound holds for $n \geq 10$, and the lower bound holds for $n \geq 100$.
Therefore, for $n \geq 100$, they both hold.

■

Example: Suppose

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

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

Example: Prove the following summation is $\Theta(n^2)$, without using the 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 $O(n^2)$

$$\begin{aligned} S(n) &= 1 + 2 + \cdots + n \\ &\leq n + n + \cdots + n \\ &\leq n^2. \end{aligned}$$

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.

$$\begin{aligned} S(n) = 1 + 2 + \cdots + n &= \sum_{i=1}^{n} i \\ &\geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} i \\ &\geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \left\lceil \frac{n}{2} \right\rceil \\ &\geq \left\lceil \frac{n}{2} \right\rceil \cdot \left\lceil \frac{n}{2} \right\rceil \geq \frac{n^2}{4} \end{aligned}$$

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

Sum-Rule and Product-Rule

Sum Rule for O( )
Suppose $T_1(n)$ is O($f(n)$) and $T_2(n)$ is O($g(n)$).
Then,

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

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

$$T_1(n) \le C_1 f(n), \quad \forall n \ge n_1$$
$$T_2(n) \le C_2 g(n), \quad \forall n \ge n_2$$

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

$$T_1(n) + T_2(n) \le C_1 f(n) + C_2 g(n), \quad n \ge \max\{n_1, n_2\}$$
$$\le C f(n) + C g(n), \quad C = \max\{C_1, C_2\}$$
$$\le C\big(f(n) + g(n)\big).$$

(**Note**: The sum rule similarly applies to Ω and Θ.)

Application of sum rule: Suppose a program has two parts: Part 1, which is executed first, and then Part 2.

Part 1 : Time $T_1(n)$ is $O(n)$
Part 2: Time $T_2(n)$ is $O(n^2)$

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

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

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

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

(**Note**: The product rule similarly applies to Ω and Θ.)

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        Inner loop number of times = (3n − 1) − (i) + 1 = 3n − i
        C = C + 1
```

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 for i is arithmetic sum, so we apply the formula for it.

$$F(n) = \sum_{i=1}^{n+1} \sum_{j=i}^{3n-1} 1 = \sum_{i=1}^{n+1} (3n - i) = (num\ of\ terms) * \frac{(first + last)}{2}$$

$$= (n + 1)\ \frac{(3n - 1) + (3n - n - 1)}{2} = (n + 1)\frac{(5n - 2)}{2}$$
$$= \frac{5n^2 + 3n - 2}{2}$$

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

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.

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

2	4	5	6	[7	3] Cm/Swp
2	4	5	[6	3] Cm/Swp	7
2	4	[5	3] Cm/Swp	6	7
2	[4	3] Cm/Swp	5	6	7
[2	3] Compare	4	5	6	7
2	3	4	5	6	7

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

```
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.

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.) 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.

Event	If element $A[i]$ is:	Number of key comparisons made by the while-loop
1	Largest	1
2	Second largest	2
3	Third Largest	3
⋮	⋮	⋮
$i-1$	Third smallest	$i-1$
i	Second smallest	i
$i+1$	Smallest	i

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

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

(Since $\frac{i}{i+1}$ is smaller than 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

$$F(n) = \sum_{i=1}^{n-1}\left(\frac{i}{2}+\frac{i}{i+1}\right) = \sum_{i=1}^{n-1}\frac{i}{2}+\sum_{i=1}^{n-1}\left(1-\frac{1}{i+1}\right) = \frac{n\,(n-1)}{4}+n-1-\sum_{i=1}^{n-1}\frac{1}{i+1}$$
$$= \frac{n\,(n-1)}{4}+n-\left(1+\sum_{i=1}^{n-1}\frac{1}{i+1}\right) = \frac{n\,(n-1)}{4}+n-\sum_{i=1}^{n}\frac{1}{i}$$

Recall that the latter summation is the harmonic series, $H_n = \sum_{i=1}^{n}\frac{1}{i} \cong \ln n$. So the expected number of key-comparisons for the entire algorithm is

$$F(n) \cong \frac{n\,(n-1)}{4}+n-\ln n = \frac{n^2}{4}+\frac{3n}{4}-\ln n$$

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.

Event	If element $A[i]$ is:	Number of SWAPS made by the while-loop
1	Largest	0
2	Second largest	1
3	Third Largest	2
⋮	⋮	3
$i-1$	Third smallest	⋮
i	Second smallest	$i-1$
$i+1$	Smallest	i

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

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

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, counts the number of swaps for each case, and averages it over all possible cases. (This analysis provides additional insight on how the expected performance is 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.

$$\underbrace{(\text{Elements} < A[i]) \quad \text{followed by} \quad (k \text{ elements} > A[i])}_{Sorted\ Part} \qquad \underbrace{A[i]}_{Element\ to\ be\ inserted}$$

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.

	Input Sequence			Number of Inversions
1	1	2	3	0
2	1	3	2	1
3	2	1	3	1
4	2	3	1	2
5	3	1	2	2
6	3	2	1	3
				Overall Average $= 9/6 = 1.5$

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.

	Pairs of input sequences which are reverse of each other			Number of inversions	Average number of inversions for each pair
	1	2	3	0	1.5
	3	2	1	3	
	1	3	2	1	1.5
	2	3	1	2	
	2	1	3	1	1.5
	3	1	2	2	
					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

$$S(n) = \frac{n(n-1)}{4}$$

The overall average number of inversions is also $S(n)$. Therefore, the expected number of swaps in the algorithm is this exact number.

Deriving the Expected Number of Comparisons from the Number of Swaps

Earlier, we derived the expected number of comparisons. As an alternative approach, we now show how we may derive the same results rather quickly from the expected number of swaps. 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. That is, if the while loop terminates with $(j > 0 \text{ and } A[j] \geq A[j-1])$, then this last comparison is not followed by a swap. And this happens when the element being inserted is not the smallest in its prefix sequence, which has the probability $i/(i+1)$, as indicted 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 equals the expected number of swaps, $S(n)$, plus $\sum_{i=1}^{n-1} \frac{i}{i+1}$.

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

$$= \frac{n(n-1)}{4} + n - \sum_{i=1}^{n} \frac{1}{i} = \frac{n(n-1)}{4} + n - H_n \cong \frac{n(n-1)}{4} + n - \ln n$$

(This is the same result which we derived earlier by a different method.)