Chapter 2
Getting Started

We will cover some of the basics about design and analysis of algorithms in this unit, using the sorting problem as an example.

We start by checking out the insertion sort algorithm. In particular, we will present a “pseudocode” version of this algorithm, which is the way that we shall specify all the algorithms. We will also argue the correctness of this algorithm and analyze its running time.

We will also introduce the divide-and-conquer approach to the algorithm design and use it to present the merge sort algorithm.
Insertion sort

To review, the sorting problem is that, given the input: \((a_1, a_2, \cdots, a_n)\), we wish to find, as the output, a permutation \((a'_1, a'_2, \cdots, a'_n)\) of the input such that \(a'_1 \leq a'_2 \leq \cdots \leq a'_n\).

Those numbers, \(a_i\)'s, are often referred to as the keys, as in databases.

We choose to use \textit{pseudocode} to specify algorithms since it is clear and concise for us to specify a given algorithm. On the other hand, we use, e.g., Java, to implement them in executables when doing projects.
What is it?

*Insertion sort* is an efficient algorithm for sorting a given number of elements.

It works the same way many people sort a hand of playing cards: Start with an empty left hand while all the cards faced down on the table, we then take one card at a time, and insert it into the correct position in the left hand, until there is nothing left on the table.

To insert a card, we compare it with each card already in the left hand, which is already sorted, from right to left, until the correct position is found or we have no more cards to compare with.
What is really going on?

An important observation is that, at all times, the elements, card or not, held on the left side are sorted, and they are originally located on the left part in the list.

Thus, we really have cut the list into two parts, the left part and the right part. The left part, initially empty, is sorted; while the right part is not.

What the algorithm does is to repeatedly take one item from the right part and insert it into the correct place in the left part, until the right part is empty.

Check out the nine plus minute video on Insertion sort... .
The algorithm

Below is the pseudocode algorithm for the insertion sort.

\[
\text{INSERTION-SORT}(A) \\
1 \text{ for } j \leftarrow 2 \text{ to } \text{length}[A] \\
2 \quad \text{do } \text{key} \leftarrow A[j] \\
3 \quad//\text{Insert } A[j] \text{ into the sorted } A[1..j-1]. \\
4 \quad i \leftarrow j-1 \\
5 \quad\text{while } i>0 \text{ and } A[i]>\text{key} \\
6 \quad\quad\text{do } A[i+1] \leftarrow A[i] \\
7 \quad\quad i \leftarrow i-1 \\
8 \quad A[i+1] \leftarrow \text{key}
\]

A demo might help.

**Homework:** Complete Exercises 2.1-1 and 2.1-2.
An example

In Pass 2, \( j=2 \) and \( i=1 \), since \( A[i]=82 > key=42 \), 82 is moved one position to the right and settle in \( A[2] \), while \( i \) is decremented to 0. In the next round of the while loop in Line 5, the value of \( key \), i.e., 42, is placed in \( A[1] \).

It now goes back to Line 1 to increment \( j \) to 3, and repeats this process.

<table>
<thead>
<tr>
<th>Pass</th>
<th>82, 42, 49, 8, 92, 25, 59, 52</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass 1</td>
<td><strong>82</strong>, 42, 49, 8, 92, 25, 59, 52</td>
</tr>
<tr>
<td>Pass 2</td>
<td>82, <strong>42</strong>, 49, 8, 92, 25, 59, 52</td>
</tr>
<tr>
<td>Pass 3</td>
<td>42, 82, <strong>49</strong>, 8, 92, 25, 59, 52</td>
</tr>
<tr>
<td>Pass 4</td>
<td>42, 49, 82, <strong>8</strong>, 92, 25, 59, 52</td>
</tr>
<tr>
<td>Pass 5</td>
<td>8, 42, 49, 82, <strong>92</strong>, 25, 59, 52</td>
</tr>
<tr>
<td>Pass 6</td>
<td>8, 42, 49, 82, 92, <strong>25</strong>, 59, 52</td>
</tr>
<tr>
<td>Pass 7</td>
<td>8, 25, 42, 49, 82, 92, <strong>59</strong>, 52</td>
</tr>
<tr>
<td>Pass 8</td>
<td>8, 25, 42, 49, 59, 82, 92, <strong>52</strong></td>
</tr>
<tr>
<td>Sorted elements</td>
<td>8, 25, 42, 49, 59, 82, 92, 52</td>
</tr>
</tbody>
</table>
What should we do?

The first thing we want to do is to prove that this algorithm indeed puts things into order.

Before proving a piece of code does exactly what we want it to, we get to know precisely what that piece does, i.e., what will be the values of all the variables at the end of that piece.

It is relatively easy to figure out what a sequence of statements does; or even what a conditional statement does, but it is not so easy to know exactly what does a loop do. 😞

Let’s have a close look at what those loops do.
What does a for loop do?

A for loop is really implemented in a while loop. For example, if \( S(i) \) is any statement which changes the loop variable \( i \), then the following two pieces are equivalent to each other:

\[
\text{for } (i=1; i<=n; i++) \\
S(i);
\]

\[
i=1; \\
\text{while}(i<=n)\
S(i); i++; \\
\}
\]

The value of \( i \) in a while loop is its value \( i_0 \) when the condition at the very beginning of the loop is tested, \( i<=n \) in this case.

**Question:** What is the value of \( i \) when the for loop completes?

**Answer:** This is the value when the loop condition fails, i.e., \( n+1 \)
Simple stuff first

**Question:** What does the following loop do?

```plaintext
k=1;
for (i=1; i<=n; i++)
    k=1;
```

**Answer:** At the end of this loop, we have that \( k=1 \) and \( i=n+1 \).

**Question:** What does the following do?

```plaintext
k=1;
for (i=1; i<=n; i++)
    k=i+1;
```

**A short answer:** At the end of the above loop, \( k=n+1 \).
A longer answer

We convert the for structure into the equivalent while format:

\[
k=1; \ i=1;\]
\[
\text{while}(i<=n)\{
\quad k=i+1; \ i++;\}
\]

We found out earlier that the last value of \(i\) is \(n+1\), which must be the value that \(i\) got assigned, in \(i++\), when the loop body was executed the very last time. Thus, before this assignment, the value of \(i\) must be \(n\).

As a result, the value that \(k\) got during this last execution of the loop body must be \(n+1\). It will not be changed afterwards.
A deeper answer

We prove that, for all $i \geq 1$, at the beginning of every loop, i.e., before testing the loop condition, $k=i$.

\begin{verbatim}
 k=1; i=1;
 while(i<=n){
   k=i+1; i++; 
 }
\end{verbatim}

At the beginning of the very first loop, $k=1=i$.

Assume the statement holds at the beginning of a loop $k=i_0$, once we enter the loop, $k$ is set at $i_0+1$, and the very next statement sets $i$ to $i_0+1$, which is exactly the value of $i$ in the next loop. Hence, at the beginning of the next loop, we still have $k=i$.

By the induction principle, the statement holds.
Now what?

This statement, “For all \( i \geq 1 \), at the beginning of every loop, \( k=i \).”, certainly holds when \( i=n+1 \), at the very beginning of the loop in which the value of \( i \) is \( n+1 \), and \( k=n+1 \).

However, since \( i=n+1>n \), the loop condition fails, and the loop body will be bypassed, and the execution control gets to the next statement with the value of \( k \) equal to \( n+1 \).

*Therefore, this loop sets the value of \( k \) to \( n+1 \) once it is completed.*

Such a statement does not depend on the value of a loop variable and is often referred to as a *loop invariant.*
Why a loop variant?

We can use such a invariant to show the correctness of an algorithm, via the following three steps:

1. *Initialization*: The property holds prior to the first iteration.

2. *Maintenance*: If it holds before an iteration of the loop, it also holds before the next iteration.

3. *Termination*: When the loop terminates, the invariant tells us exactly what this loop does.
What is going on?

The loop invariant is essentially an inductive argument, when the first two pieces are shown, this loop invariant holds before every iteration of the loop.

The inductive argument shows that a property holds for every value of $n$, but a loop will not run forever. Thus, we need the third piece to wrap things up, which leads to a final instance of the loop invariant when the loop stops.

This final instance tells us what this loop does, which often helps us to establish the correctness of an algorithm.
Correctness of the insertion sort...

In the insertion algorithm, the index $j$ points to the current card being inserted into the hand.

At the beginning of each for loop, the subarray consisting of $A[1..j-1]$, i.e., the left part, is already sorted,; and elements in $A[j..n]$, i.e., the right part, may not be sorted.

We can characterize this property as a loop invariant: At the start of each iteration of the for loop, the subarray $A[1..j-1]$ consists of the elements in the original $A[1..j-1]$, but in sorted order.
... via the loop invariant

1. When \( j = 2 \), the subarray \( A[1..j-1] \) contains only one element, \( A[1] \), which is certainly sorted.

2. Assume the property holds before an iteration of the loop, where \( j \) holds the value of \( j_0 \), i.e., the subarray \( A[1..j_0-1] \) consists of the elements in the original \( A[1..j_0-1] \) but in sorted order. Now, the inner loop, lines 4-7 finds the correct position for \( A[j_0] \), which is then inserted into the list in line 8. Thus, \( A[1..j_0] \) consists of the elements in the original \( A[1..j_0] \) and sorted.

At the end of this loop, the value of the loop variable \( j \) is implicitly incremented to \( j_0 + 1 \), which is its value in the next loop.
Hence, at the beginning of the next loop, the statement that “the subarray A[1..j−1] consists of the elements in the original A[1..j−1] but in sorted order” really means that “the subarray A[1..j_0] consists of the elements in the original A[1..j_0] but in sorted order”, which does hold as we discussed before.

3. The above two steps show that the property holds before every iteration of the loop. At the end of the for loop, j has the value of n + 1. Hence, before the next iteration, which will not happen since the loop condition has failed, the subarray A[1..j−1] consists of the elements in the original A[1..n] but in sorted order, which is precisely what we want to show.

Therefore, the Insertion sort algorithm is indeed correct.

**Homework:** Self read the *Pseudocode conventions*, in pp. 19, then complete Exercises 2.1-3 and 2.1-4.
Algorithm analysis

Algorithm analysis is the process of predicting the resources that the algorithm requires.

Besides the need for such resources as memory space, communication bandwidth, or computer hardware, we mainly want to know how much time it takes the algorithm to complete its job.

We usually make such an analysis for several candidates to solve the same problem. At the end, we might not be able to pick up a clear winner, but we are often able to eliminate the inferior algorithms.
Our model

Before we carry out any analysis, we have to have a model for implementing the algorithms. We shall assume a generic uniprocessor model with a randomly accessible memory, and implement our algorithms as computer programs accordingly.

We should define precisely the instruction set for such a model as well. For example, we should make clear if our model supports a sort instruction. But, this would be tedious and difficult.

Thus, we assume our RAM machine contains instructions commonly found in real computers, such as arithmetic, data movement, and control, subroutine call, and each such instruction takes a constant amount of time.
Forget about the details

We only allow two data types, integers and floating point. We also don’t attempt to model the memory hierarchy, such as cache and virtual memory.

The time that the Insertion sort takes certainly depends on the size of the input: the larger $n$ is, the longer it takes the algorithm to run.

Another factor is how nearly sorted the list is: while it is irrelevant to some sorting algorithms such as selection sort and merge sort, it makes quite a difference for insertion sort.

But, in general, we describe the time of an algorithm as a function of the input size only.
What size?

It really depends on the problem. For many problems, such as search and sorting, it is natural to measure the size of the input in terms of the number of the elements in the input.

For some other problems, such as multiplying two integers, it might be the total number of bits needed to represent the input in a binary notation.

Sometimes, it might be appropriate to use two numbers rather than one. For example, when working with a graph, we need to know both the number of points and the number of edges.
How much is the cost?

It also depends on the problem at hand. We will focus on the time part.

We could find out the total number, or the total number of steps, or pseudocode lines, for the algorithm to complete, as we will do shortly for the insertions sort. To alleviate our effort, we adopt the assumption that a constant amount of time is required to execute each line of our pseudocode algorithms.

We could also focus on the total number of key operations that the algorithm has to go through, such as comparisons and assignments for the insertion sort.
Insertion sort analysis

Assume that it takes $c_i$ units of time to execute line $i$, and let $t_j$ be the number of times the while loop test in line 5 is executed for $j \in [2, n]$, we have the following data for each line $i$, $i \in [1, 8]$.

<table>
<thead>
<tr>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$c_3$</th>
<th>$c_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$n - 1$</td>
<td>0</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>$\sum_{j=2}^{n} t_j$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
<td>$n - 1$</td>
</tr>
</tbody>
</table>

The total time, $T(n)$, is just the sum of the time spent on all the lines:

$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j$

$\quad + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1)$.  

23
The best case

If the array is already sorted, for each \( j = 2, 3, \ldots, n \), we have that \( A[i] \leq \text{key} \). Thus, the test is done only once, i.e., for all \( j \in [2, n] \), \( t_j = 1 \).

The best case is then simply the following:

\[
T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) \\
+ c_5 (n - 1) + c_8 (n - 1) \\
= (c_1 + c_2 + c_4 + c_5 + c_8) n \\
- (c_2 + c_4 + c_5 + c_8).
\]

This running time can be represented as a linear function of \( n \), \( an + b \) for constants \( a \) and \( b \), depending only on the statement costs \( c_i \).
The worst case

If the array is reversely sorted, we end up with the worst case: We must compare $A[j]$ with each and every element in the entire subarray $A[1..j-1]$, which leads to $t_j = j$ for each $j = 2, 3, \ldots, n : j – 1$ times for the $j – 1$ elements, and one more for the failure of $i > 0$, where no comparison will be done.

$$T(n) = c_1n + c_2(n - 1) + c_4(n - 1) + c_5 \left[ \frac{n(n + 1)}{2} - 1 \right]$$
$$+ c_6 \left[ \frac{n(n - 1)}{2} \right] + c_7 \left[ \frac{n(n - 1)}{2} \right] + c_8(n - 1)$$
$$= \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left( c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n$$
$$- (c_2 + c_4 + c_5 + c_8).$$

This worst case can be represented as a quadratic function of $n$, $an^2 + bn + c$, for constants $a, b$ and $c$, depending only on the statement costs $c_i$.

**Homework:** Exercises 2.2-1 and 2.2-2.
How many comparisons?

If you look through the algorithm, it does only one comparison, in Line 5.

In the best case, when the input list is pre-sorted, for each element \( A[j], j \in [2, n] \), we make exactly one comparison. Thus,

\[
C_{\text{best}}(n) = n - 1.
\]

In the worst case, when the list is completely out of order, for all \( j \in [2, n] \), it compares with all the previous \( j - 1 \) elements, we have that

\[
C_{\text{worst}}(n) = \sum_{j=2}^{n} (j - 1) = \sum_{j=1}^{n-1} j = \frac{(n - 1)n}{2}.
\]
How many movements?

If you look through the algorithm, it does two movements, in Line 2 and 8, respectively, and another one in Line 6.

In the best case, when for all \( j \in [2,n] \), no comparison is made. We have that

\[
M_{\text{best}}(n) = 2(n - 1).
\]

In the worst case, for all the \( j - 1 \) elements, \( A[j], j \in [2,n] \), we have to move \( A[j] \) to the right. Thus,

\[
C_{\text{worst}}(n) = 2(n - 1) + \sum_{j=2}^{n} (j - 1)
\]

\[
= 2(n - 1) + \frac{(n - 1)n}{2} = \frac{(n - 1)(n + 4)}{2}.
\]
Yet another reason

3. The \textit{average-case} running time is often roughly the same as the worst case, but much more difficult to get. In the case of the insertion sort, given a randomly chosen list of numbers and apply the algorithm, on average, $t_j = \frac{j}{2}$, which also leads to a quadratic function of $n$, although with a smaller constant of $\frac{1}{4}$.

We will look at the details of the average-case of run time of the Insertion sort algorithm a little later.

\textbf{Homework:} Exercise 2.2-3.
Order of growth

When analyzing the Insertion sort algorithm, we applied certain simplification technique: We ignore the actual cost of the statements, using $c_i$ to represent it. We then combine all the $c_i$’s to come up with either a linear function or a quadratic function of $n$.

We can even take another simplifying abstraction, the rate of growth, or the order of growth. We will then only consider the leading term of a formula, e.g., the $n^2$ term in the quadratic function since the lower-order terms are less significant for large $n$. Thus, we will express the worst-case running time of the Insertion sort algorithm as $\Theta(n^2)$.

We will study this concept of growth rate in details in the next Chapter.
Algorithm design

There are quite a few ways to design algorithms. Insertion sort follows an incremental approach: having sorted the subarray $A[1..j-1]$, we repeatedly insert the next element, $A[j]$, into the list to get a larger sorted list, until there is nothing to increment.

We now introduce another method, *divide and conquer*, and present another sorting algorithm, the *merge sort* algorithm, that is significantly more efficient than the insertion sort.
Divide and conquer

When solving a problem recursively, an algorithm calls itself recursively one or more times to deal with closely related smaller and/or simpler problems.

Such an algorithm typically follows the *divide and conquer* approach: breaking the original problem into several subproblems that are similar to the original ones, but smaller in size, or simpler in its nature; solve those subproblems recursively, then combine their solutions to create a solution for the original problem.

We cannot finish the whole 10 inch pizza in a single bite, so we bite it in pieces....
More specifically,...

a divide-and-conquer algorithm contains the following steps:

1) *divide* the problem into a number of sub-problems;

2) *conquer* the sub-problems by solving them recursively. If the size is small enough, solve it directly;

3) *combine* the solutions of the sub-problems into a solution for the original one.

Let’s look at a few sorting examples, following this approach, including insertion sort, selection sort, quicksort and merge sort.
Merge sort

Following the general principle of divide and conquer, the merge sort algorithm consists of the following three steps:

0. If the size is 1, there is nothing to do.

1. If the size is larger than 1, divide the $n$-element sequence to be sorted into two subsequences of $n/2$ elements each, with their difference being at most 1.

2. Sort the two subsequences recursively using merge sort.

3. *Merge* the two sorted subsequences into the sorted answer.

So, the only thing we need to work out is this merge piece.
The Merge procedure

The Merge(A, p, q, r) procedure, where A is an array, p, q, r are indices such that \( p \leq q < r \), assuming that \( A[p..q] \) and \( A[q+1, r] \) are in sorted order.

This procedure merges the two sorted subarrays to form a single sorted subarray, by repeatedly comparing two items, one from each subarray, finally replaces the current \( A[p..r] \) with the sorted one.

Since each comparison sends out one item, and there are \( n \) items in the list, the whole merge process makes at most \( n \) comparisons.

**Homework:** Exercises 2.3-1, 2.3-2, and 2.3-4.
The code for Merge

MERGE(A, p, q, r)
1. n1<-q-p+1 //size of A[p..q]
2. n2<-r-q //size od A[q+1..r]
3. create arrays L[1..n1+1] and R[1..n2+1]
4. for i<-1 to n1
5. do L[i]<-A[p+i-1]
6. for j<-1 to n2
7. do R[j]<-A[q+j]
8. L[n1+1]<-maxInt //Set the bedrock
9. R[n2+1]<-maxInt
10. i<-1
11. j<-1
12. for k<-p to r
13. do if L[i]<=R[j] //Comparison
14. then A[k]<-L[i]
15. i<-i+1
16. else A[k]<-R[j]
17. j<-j+1

This process makes exactly \( p - r + 1 \) comparisons.
What does Merge do?

Before applying the above MERGE(A, p, q, r), we have to make sure that both A[p..q] and A[q+1..r] are already sorted, which is the precondition of Merge.

Steps 1 through 9 essentially copy out the two sorted parts into two separate lists L and R, then, Steps 12 through 17 merge these two sorted parts into one sorted list containing all the elements.

This latter fact can be proved using a loop invariant related argument.

Still remember this stuff?
The loop invariant of Merge

After creating and copying the elements into the two respective arrays, L and R, lines 10-17 of the Merge algorithm works correctly by maintaining the following loop invariant:

At the start of each iteration of the for loop of lines 12-17, the following are true:

1. The subarray $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n1+1]$ and $R[1..n2+1]$, in sorted order.

2. $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into $A$. 
The very first case

Prior to the first iteration of the loop, $k = p$, thus the subarray $A[p..k-1]$ is empty, which contains 0 smallest elements of $L$ and $R$. Thus, part 1 of the invariant holds.

Moreover, since $i = j = 1$, both $L[i]$ and $R[j]$ are indeed the smallest elements of their arrays, since both $L$ and $R$ are sorted. Thus, part 2 also holds.
The maintenance part

Assume that the invariant holds for \( k = k_0 \), when \( i = i_0 \) and \( j = j_0 \); i.e., 1) The subarray \( A[p..k_0-1] \) contains the \( k_0 - p \) smallest elements of \( L \) and \( R \), in sorted order; and 2) \( L[i_0] \) and \( R[j_0] \) are the smallest elements of their arrays that have not been copied back into \( A \).

We also assume that \( L[i_0] \leq R[j_0] \). (The other case is the same.) Then, \( L[i_0] \) is the smallest element, among the remaining elements in \( L \) and \( R \), yet copied back into \( A \). Step 14 copies \( L[i_0] \) into \( A[k_0] \), then the other steps increment both \( i \) and \( k \) to \( i_0 + 1 \) and \( k_0 + 1 \), respectively.

Thus, at the beginning of the next loop where \( k = k_0 + 1 \), the segment \( A[p..k-1] \), i.e., \( A[p..k_0] \), contains the \( (k_0 - p) + 1(= k - p) \) smallest elements of \( L \) and \( R \). They are indeed in sorted order by assumption. Thus, part 1 of the invariant holds.
Moreover, $L[i] = L[i_0 + 1]$ is the smallest element of $L$ among those yet copied back, since $L$ is sorted; and $R[j] = R[j_0]$ still holds the smallest element of $R$ that has not been copied back. Thus, the second part of the invariant also holds.

Therefore, this invariant holds at the beginning of every for loop.

In particular, for the termination part, at the end, $k = r + 1$. By the first part of the invariant, the (sub)array $A[p..r]$ contains $k - p = r - p + 1$ (smallest) elements of $L$ and $R$, in sorted order.

Thus, this merge procedure indeed does what we claimed.

We are now ready for the Merge sort algorithm, usually credited to John von Neumann.
Merge sort

We can now present the *merge sort* algorithm, which uses Merge as a subroutine, as follows:

MERGE-SORT(A, p, r)
1. if (p<r)
2. then q<-(p+r)/2
3. MERGE-SORT(A, p, q)
4. MERGE-SORT(A, q+1, r)
5. MERGE(A, p, q, r)

To sort the entire sequence as contained in A, we simply make the following call

MERGE-SORT(A, 1, n),

where n is the length of A.

Let's prove its correctness with strong induction.
Correctness of Merge sort

Claim: Merge-sort(A, p, r) sorts A[p..r].

Proof by strong induction on |A[p..r]| (= n):
Assume that the procedure correctly sorts A[p..r] if its length is less than n, we consider an array A[p..r] such that |A[p..r]| = r - p + 1 = n > 1.

Since n > 1, we have that p < r. Step 2 sets q to (p + r)/2. Assume that p + r is even, i.e., for some m, p + r = 2m, then q = m.

We now calculate the length of the first part as follows:

\[ |A[p..q]| = q - p + 1 = m - p + 1 \]
\[ = \frac{p + r}{2} - p + 1 = \frac{r - p + 2}{2} \]
\[ = \frac{(r - p + 1) + 1}{2} = \frac{|A[p..r]| + 1}{2}. \]
Since by assumption, $|A[p..r]| > 1$, we have that


Regarding $|A[q+1..r]|$, the length of the second part, we have that


This leads to the following:

$$1 \leq |A[q + 1..r]| < |A[p..r]|.$$  

Thus, the length of both parts are strictly less than that of $A[p..r]$. The other case of $p + r$ is odd can be similarly argued.

By the inductive assumption, Steps 3 and 4 sort both $A[p..q]$ and $A[q+1..r]$ into order before calling the Merge procedure.

The proof now completes, following the correctness proof of the Merge procedure.
Divide’n Conquer analysis

We often use a recurrence relation to analyze a recursive algorithm, which can often be derived from the process.

Let $T(n)$ be the running time on a problem of size $n$. If $n \leq c$ for some small $c$, a straightforward solution is applied, which often takes $\Theta(1)$ time.

In general, assume that we cut the problem into $a$ pieces, each of which is $1/b$ the size of the original; and let $D(n)$ be the cutting time, and $C(n)$ be the solution combining time, we get the following:

$$T(n) = aT(n/b) + D(n) + C(n).$$
Merge sort analysis

Since merge sort cuts the original problem into two equal pieces, we immediately have the following:

\[ T(1) = \Theta(1) \]
\[ T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n), n > 1. \]

A “master theorem” (Cf. Section 4.5) shows that

\[ T(n) = \Theta(n \log n). \]

Intuitively speaking, since each cut will take away half of the problem, we can make at most \( \log n \) such cuts.

For each such cut, it takes \( \Theta(n) \) to do the merge, since each element will be sent out exactly once. Thus, the total time has to be \( \Theta(n \log n) \).
A not-so-simple case

Assume $n$ is a power of 2, and take the assumption that $\Theta(n) = n$, we have the following relation for $T(n)$:

$$
T(1) = 1
$$

$$
T(n) = 2T\left(\frac{n}{2}\right) + cn.
$$

It is easy to carry out the following calculation:

$$
T(n) = 2 \left[ 2T\left(\frac{n}{2^2}\right) + \frac{cn}{2} \right] + cn
$$

$$
= 2^2 T\left(\frac{n}{2^2}\right) + 2cn
$$

$$
= \cdots
$$

$$
= 2^k T\left(\frac{n}{2^k}\right) + kcn.
$$

Now, $\frac{n}{2^k} \geq 1$, hence, $n \geq 2^k$, i.e., $k \leq \log n$.

Hence, $T(n) \leq n + cn \log(n) = \Theta(n \log(n))$. 
The general case

Assume that \( n \) is not a power of 2, then, for some \( m \geq 1, 2^m < n < 2^{m+1} \), i.e., \( m = \lfloor \log n \rfloor \).

Since \( T \) is monotonically non-decreasing,
\[
T(2^m) \leq T(n) \leq T(2^{m+1}).
\]
By the previous result, we have
\[
c_1m2^m + m \leq T(n) \leq c_2(m+1)2^{m+1} + m + 1, \text{ i.e.,} \]
\[
c_1m2^m \leq T(n) \leq c_3m2^m.
\]
With \( m = \lfloor \log n \rfloor \), we have
\[
c_1\lfloor \log n \rfloor 2^{\lfloor \log n \rfloor} \leq T(n) \leq c_3\lfloor \log n \rfloor 2^{\lfloor \log n \rfloor}.
\]
As we will see a bit later, the above means that \( T(n) = \Theta(n \log n) \).
Assignments

1. Complete the rest of the analysis section.

2. Exercises 2.3-3, 2.3-5 and 2.3-6.


4. Have you checked the project page recently?