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 prove the correctness of this algorithm and analyze its running time.

We then introduce the divide-and-conquer approach to the algorithm design, and use it to present the Mergesort algorithm, maybe the best sorting algorithm, as well as its analysis through a recurrence equation.

These and other sorting algorithms will be implemented in Project 1.
Insertion sort

To review, the sorting problem is that, given the input: \((a_1, a_2, \ldots, a_n)\), we wish to find, as the output, a permutation \((a'_1, a'_2, \ldots, a'_n)\) 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, where it finds important applications everywhere in this data driven era. 😊

We choose to use 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 most of the projects in this course.

Check out the course page to find out how such sorting mechanisms work, including bubble sort, insertion sort, Mergesort, quicksort, and selection sort.
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.

**Question:** How does this insertion work?

**Answer:** To insert a card, we compare it with each card in the left hand pile, which is already sorted, *from right to left*, until the correct position is found or we have no more cards to compare with, thus put *it* at the very beginning.

Check out the nine plus minute video on Insertion sort on the course page. 😊
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.

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, when everything is sorted.

Question: Where should we start?

Answer: Position 2.
The algorithm

Below is the pseudocode algorithm, the “for”-version, for the insertion sort.

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

A demo might help.

Question: Why i>0 goes first in Line 5?

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

In Pass 1, $j=2$ and $i=1$, since $A[1]=4 > \text{key}=3$, 4 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., 3, is placed in $A[1]$.

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

Assignment: Play with it by following the process as guided by the algorithm...
What should we do?

“In addition to knowing that something works, we want to know why it works.” 😄

Thus, we now want 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 a sugar coated 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:

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

```plaintext
i=1;
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 for the first time, i.e., $i=n+1$ in this case.
Question: What does the following loop do?

```c
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?

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

A short answer: At the end of the above loop, $k=n+1$.

Question: Why? This is much more important than what... .
A longer answer

We convert the for structure into the equivalent while format:

```
k=1; i=1;
while(i<=n){
    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, \( k=i \).

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

At the beginning of the very first loop, \( k=1=i \).

Assume the statement holds at the beginning of a loop when \( k=i_0 \), 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*, which plays a special role in telling us what a loop does *at its end*. 
Why a loop invariant?

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*.

The combination of the first two (mathy) steps will show a general property by the inductive principle, and the last (csey) step will tell us a concrete result when the loop ends. 😊
More specifically...

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.

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

We thus need a combination of Math and CS to make this work. 😊
Insertion Sort again...

Below is the “while” version of the algorithm, which we will use in the rest of the chapter. We add Steps 0 and 9 to the original.

```
INSERTION-SORT(A)
0  j<-2
1  while j<=length[A]
2    do key <- A[j]
3      //Insert A[j] into the sorted A[1..j-1].
4      i <- j-1
5      while i>0 and A[i]>key
6        do A[i+1]<-A[i]
7        i<-i-1
8      A[i+1]<-key
9    j<-j+1
```

We notice that the index \( j \) points to the current item being inserted into the right place.
Correctness of the insertion sort...

We observe that, at the beginning of each while loop, line 1, 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 while loop, again, in line 1, the subarray A[1..j-1] consists of the elements in the original A[1..j-1], but in sorted order.

**Question:** How do we prove the above statement?

**Answer:** We do it...
... 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, at Line 1, where $j$ holds the value of $j_0$, i.e., the subarray $A[1..j_0-1]$ consists of the elements in the original $A[1..j_0-1]$ in a sorted order.

Now, the inner loop, lines 5-7, 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, in Line 9, the value of the loop variable $j$ is incremented to $j_0 + 1$, which is its value at the beginning of the next loop, at line 1.
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 top while 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*, on Page 19 of the textbook, then complete Exercise 2.1-3.
I want to see...

Loop Invariant: At the start of each iteration of the while loop, $A[1..j-1]$ is sorted.

Thus, at the end, $A[1..n]$ is sorted. 😊
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*, since time is irreplaceable, and cannot be bought. 😞

We usually make such an analysis for several candidates to solve the same problem and pick a best algorithm among them. Check out the chart at the end of Chapter 0 notes, or the one through the Topics button on the course page.

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, \textit{integer} and \textit{floating point}. We also don’t attempt to model the memory hierarchy, such as cache and virtual memory.

The time that the \textit{Insertion sort} algorithm 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 \textit{selection} sort and \textit{mergesort}, it makes quite a difference for insertion sort, where \textit{sorting time depends on how sorted the list is}.

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, the number of edges, and perhaps even the length of such an edge.

We will see examples of this nature later on.
What do you mean by cost?

By “cost”, we could mean the total cost of the algorithm, or we could also focus on the most expensive operations: It takes more time to multiply two things up, as compared to adding them together.

Thus, we could find out the total number of steps, or pseudocode lines, for the algorithm to complete, as we will do shortly for the Insertion sort algorithm.

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 essential operations that an 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 $i$, (See the code on page 15), and let $t_j$ be the number of times the inner while loop test in line 5 is executed for $j \in [2, n]$, we have the following data for each line $i$, $i \in [1, 9]$, as Line 0 does just once.

<table>
<thead>
<tr>
<th></th>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$c_4$</th>
<th>$c_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n$</td>
<td>$n-1$</td>
<td>$n-1$</td>
<td>$\sum_{j=2}^{n} t_j$</td>
</tr>
<tr>
<td>$c_6$</td>
<td>$c_7$</td>
<td></td>
<td>$n-1$</td>
<td>$n-1$</td>
</tr>
<tr>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

$$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1) + c_9 (n - 1).$$
The best case

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

The best case is then simply the following:

\[
T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) \\
+ c_5 (n - 1) + c_8 (n - 1) + c_9 (n - 1) \\
= (c_1 + c_2 + c_4 + c_5 + c_8 + c_9)n \\
- (c_2 + c_4 + c_5 + c_8 + c_9).
\]

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 exact value of the constants does not matter. Still remember the example that we gave at the end of the last chapter, Pages 34 and 35, regarding the constants?
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 $j = 2, 3, \cdots, n$ : $j - 1$ times for the $j - 1$ elements, and one more for the failure of $i > 0$, where no comparison between $t[0]$ and key will be done.

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \left[ \frac{n(n+1)}{2} - 1 \right]$$

$$+ \quad c_6 \left[ \frac{n(n-1)}{2} \right] + c_7 \left[ \frac{n(n-1)}{2} \right] + c_8(n-1) + c_9(n-1)$$

$$= \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 - (c_2 + c_4 + c_5 + c_8 + c_9)$$

$$+ \left( c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 + c_9 \right) n.$$

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

**Homework:** Exercises 2.2-1 and 2.2-2.
Major operations

Sorting algorithms are often involved with two major operations: comparison and movement.

If you look through the algorithm, it does only one comparison of the elements kept in A, in line 5 in between A[i] and key (=A[j]).

In the best case, when the input list is pre-sorted, for each element A[j], j ∈ [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 ∈ [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 always does two movements of elements kept in A, in Line 2 and 8, respectively, and another one in Line 6, within a loop.

In the best case, for all $j \in [2, n]$, no additional movement is made in the loop. We have that

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

In the worst case, for $j \in [2, n]$, all the $j - 1$ elements have to be moved one position to the right in Lin 7. Thus,

$$M_{\text{worst}}(n) = 2(n - 1) + \sum_{j=2}^{n} (j - 1)$$

$$= 2(n - 1) + \frac{(n - 1)n}{2} = \frac{(n - 1)(n + 4)}{2}.$$
Average running time

The *average-case* running time calculates the running time of an algorithm on the average, i.e., for a *typical* input.

For example, out of 5! (=120) permutations of five distinct elements, one each, less than 1% of all the cases, leads to the best and worst case, respectively.

Thus, when considering the *average* case, we need to consider all the 120 cases. 😞

It is much more realistic, but tough to get, because we would need to set up a probabilistic model for the input distribution.

We will go through a detailed analysis of the average running time of the *Insertion sort* algorithm in U5 Probabilistic analysis.

Stay tuned... 😊.
In the case of the insertion sort, given a randomly chosen list of numbers, assuming all the \( n! \) permutations are equally likely to occur, on average, \( t_j \), the number of times that the algorithm has to go through the loop is about \( \frac{j}{2} \), half of the length of the list it is working with.

Hence,

\[
C_{\text{avg}}(n) = \sum_{j=2}^{n} \frac{j}{2} = \frac{1}{2} \sum_{j=2}^{n} j = \frac{1}{4}[n(n + 1) - 2].
\]

We will work out the exact details of the average-case of run time of \textit{Insertion sort} algorithm a little later in the \textit{Probabilistic analysis} chapter, where we will find out that indeed the average-case running time of the Insertion sort is a quadratic function of \( n \), with a constant of \( \frac{1}{4} \).

Because of such a challenge, we are often satisfied with a worst-case analysis of an algorithm, also a quadratic function of \( n \), but with a slightly larger constant of \( \frac{1}{2} \).
Order of growth

When analyzing the *Insertion sort* algorithm, we applied certain simplification technique: We ignore the actual cost of the statements, using a constant $c_i$ 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 “$\Theta(n^2)$”, i.e., “in the order of $n^2$”.

We will study this concept of growth rate in details in the next chapter on *function growth*.
Could we get a faster one?

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 already sorted sublist to get a larger sorted list, until there is nothing left.

We now introduce another method, *divide and conquer*, and present an alternative sorting algorithm, the *mergesort* algorithm, which is significantly faster than the *Insertion sort*: Instead of $\Theta(n^2)$, it always takes $\Theta(n \log n)$ to sort out a list of $n$ elements.

In fact, it is a fastest *comparison based* algorithm, as suggested by John von Neumann himself back in 1945.

**Question:** Have you seen, or heard from, Dr. von Neumann on the course page?
Divide and conquer

When solving a problem *recursively*, an algorithm calls itself, thus *recursively*, to deal with *smaller and/or simper* problems of the same nature.

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

We cannot finish the whole 12 inch pizza in a single bite, so we cut *it* in pieces....
A picture worths how many words?

A divide and conquer process can be described as follows.

Here $m \in [1, n - 1]$, in other words, we divide a list, $A$, with $n$ elements, into two sublists, $A_1$ with $m$ elements, and $A_2$ with $n - m$ elements. We find out solutions $A'_1$ and $A'_2$, which are finally combined into a solution $A'$. 

35
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 *selection sort*, *insertion sort*, *quicksort*, before looking at *merge-sort* in details.

There are also a few related videos on the course page. You want to watch them to have a better understanding… .
Mergesort

Following the general principle of divide and conquer, the **mergesort** 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, cut (*divide*) the $n$–element sequence to be sorted into two subsequences of $n/2$ elements each, with their difference being at most 1. Notice that

   $\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = n.$

2. Sort the two subsequences recursively using **mergesort**.

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

We can now present the *mergesort* 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.

When p=r, n=1, A contains just one element. There is nothing to do.

For the general case, let's have a look at an example.
I want to see...

... an example of *mergesort*...

So, the only thing we need to work out is this merging piece, i.e., how to merge two sorted sublists, (2, 4, 7, 8) and (1, 5, 6, 9) into (1, 2, 4, 5, 6, 7, 8, 9).
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*, which is the precondition to apply this procedure.

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 merging process makes at most \(n\) comparisons.

For an alternative implementation of the Merge procedure, check out Exercise 2.3-2.

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

MERGE(A, p, q, r)
1. \( n_1 <- q - p + 1 \) //size of A[p..q]
2. \( n_2 <- r - q \) //size od A[q+1..r]
3. // create arrays L[1..n1+1] and R[1..n2+1]
4. for \( i <- 1 \) to \( n_1 \)
5. do \( L[i] <- A[p+i-1] \)
6. for \( j <- 1 \) to \( n_2 \)
7. do \( R[j] <- A[q+j] \)
8. \( L[n1+1] <- \max \text{Int} \) //Set the bedrock
9. \( R[n2+1] <- \max \text{Int} \)
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 \( r - p + 1 \) comparisons, since in each of the for loop, at Line 12, one of this many elements is sent out to A.
I want to see...

... an example of the Merge process...

Notice that once \( j \) hits 5, it simply moves the last two elements of \( L \) to position 8 and 9 of the array \( A \) because \( R[5] = \infty \), larger than anything.

**Question:** It is time to work on Project 2.
What does Merge do?

Again, 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 (Cf. Page 40)

Steps 1 through 9 essentially copy out the two sorted parts into two separate lists L and R, then, Steps 10 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.

**Question:** Still remember this stuff?

Have another look at the stuff from Page 11 through 19.
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 \( A[p..k-1] \) (=\( A[p..p-1] \)) 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 (Precondition 😊). 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.

As a result, 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.

Notice that, at the end, $i = n_1 + 1$, and $j = n_2 + 1$.

Thus, this merge procedure indeed does what we claimed.

We are now ready for the mergesort algorithm itself.
I want to see...

Below demonstrates the proof:

1. L \[ \text{ Initial case: } \]
   i = j = 1
   k = p

2. L \[ \text{ Assume it holds } \]
   When \( i = i_0, j = j_0 \)
   and \( k = k_0 \)

3. A \[ \text{ At the end, A[p, r] is sorted. } \]
   k = r + 1
déjà vu

Here goes the *mergesort* algorithm again, which uses Merge as a subroutine, as follows:

```
MERGE-SORT(A, p, r)
0. //if A[p..r] contains more than 1 element
1. if (p<r)
2. //Element A[q] is the "middle point"
3. then q<-(p+r)/2
4. MERGE-SORT(A, p, q)
5. MERGE-SORT(A, q+1, r)
6. //Now both A[p..q] and A[q+1..r] are sorted
7. 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.
Correctness of *mergesort*

**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 3 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 4 and 5 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, obtained following a Divide’n Conquer approach.

Let $T(n)$ be the time running such a recursive algorithm 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 of size $n$ into $a$ pieces, each of which is $1/b$ the size of the original (When $n$ is not a multiple of $a$, $a \neq b$); and let $D(n)$ be the cutting time, and $C(n)$ be the solution combining time, we get the following:

\[
T(c) = 1 \\
T(n) = aT(n/b) + D(n) + C(n).
\]
Mergesort analysis

Since *mergesort* cuts the original problem into two “equal” pieces, we immediately have the following:

\[
T(1) = \Theta(1)
\]

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

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 (See next page for details).

For each such cut, it takes at most \(n\) comparison in the merging phase, since each element will be sent out exactly once, after at most one comparison (See the figures on Pages 39 and 42).

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 $\Theta(n) = cn$, we have the following relation for $T(n)$:

\[
T(1) = c_1 \\
T(n) = 2T\left(\frac{n}{2}\right) + c_2n.
\]

It is easy to carry out the following calculation:

\[
T(n) = 2 \left[ 2T\left(\frac{n}{2^2}\right) + \frac{c_2n}{2} \right] + c_2n \\
= 2^2 T\left(\frac{n}{2^2}\right) + 2c_2n \\
= \ldots \\
= 2^k T\left(\frac{n}{2^k}\right) + kc_2n.
\]

Now, since $n$ is a power of 2, for some $k_0$, $\frac{n}{2^{k_0}} \geq 1$, hence, $n \geq 2^{k_0}$, i.e., $k_0 \leq \log n$.

Hence,

\[
T(n) \leq c_1n + c_2n \log(n) = \Theta(n \log(n)).
\]
The general case

Assume that $n$ is *not* a power of 2, then, similar to the case shown on Page 17 in Chapter 0, for some $m \geq 1, 2^m < n < 2^{m+1}$, i.e., $m < \log n < m + 1$. thus, $m = \lfloor \log n \rfloor$.

Since $T(n)$ is *monotonically non-decreasing*,

$$T(2^m) \leq T(n) \leq T(2^{m+1}).$$

By the previous result, we have

$$c_1 m 2^m + c_2 m \leq T(n) \leq c_3 (m+1) 2^{m+1} + c_4 (m+1).$$

The above leads to the following:

$$c_1 m 2^m \leq T(n) \leq c_5 m 2^m.$$ 

With $m = \lfloor \log n \rfloor$, we have

$$c_1 \lfloor \log n \rfloor 2^{\lfloor \log n \rfloor} \leq T(n) \leq c_5 \lfloor \log n \rfloor 2^{\lfloor \log n \rfloor}.$$

As we will see in the next mathy (messy?) chapter, the above means that

$$T(n) \equiv \Theta(n \log n),$$

which turns out to grow much slower than $\Theta(n^2)$ (?), the time taken by *Insertion sort*. 

55
I want to see...

Assignments:

1. Complete the rest of the analysis section

2. Exercises 2.3-3 and 2.3-6