Chapter 2 Getting Started

Insertion Sort: This algorithm will help us solve the sorting problem introduced in the previous chapter. It is an efficient algorithm for sorting a small number of elements.

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
Output: A permutation (reordering) $\langle a_1', a_2', \ldots, a_n' \rangle$ of the input sequence such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

The numbers that we wish to sort are known as the **keys**

Example: Suppose we have $A = \langle 5,2,4,6,1,3 \rangle$. To sort this array of numbers, first 2 need to be compared with 5, since 2 is less than 5, we exchange the two numbers, and hence our array becomes: $A = \langle 2,5,4,6,1,3 \rangle$. Next we must compare 4 with 5 and 2, since 4 is smaller than 5, we exchange 4 and 5, then 4 is compared with 2, but 4 is larger than 2, so nothing to be done, so we get: $A = \langle 2,4,5,6,1,3 \rangle$. Next we compare 6 with 5. Since 6 is the larger than 5, no exchange is done. Now, 1 is to be compared with 6, 5, 4, and 2. Since 1 is the smallest, it is to be exchanged until we get the following array: $A = \langle 1,2,4,5,6,3 \rangle$. Finally, 3 is exchanged with 6, then 3 is exchanged with 5, then it is exchanged with 4. Hence, we have the following sorted array: $A = \langle 1,2,3,4,5,6 \rangle$.

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j-1]
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i - 1
8      A[i+1] = key
```

A **loop invariant** is an invariant used to prove properties of loops. Informally, a loop invariant is a statement of the conditions that should be true on entry into a loop and that are guaranteed to remain true on every iteration of the loop.

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

Initialization: It is true prior to the first iteration of the loop
Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Now, let us see how these properties hold for insertion sort

Initialization: When $j = 2$, the subarray $A[1..j-1]$ consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. But, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: We will show this step informally, the body of the **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4-7), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1..j]$ then consist of the elements originally in $A[1..j]$, but in sorted order. Incrementing j for the next iteration of the **for** loop then preserves the loop invariant.

Termination: The condition causing the **for** loop to terminate is that $j \succ A.length = n$. Because each loop iteration increases j by 1, we must have $j = n+1$ at that time. Substituting $n = 1$ for j in the wording of the loop invariant, we have that the subarray $A[1..n]$ consists of the elements originally in $A[1..n]$, but in sorted order. Observing that the subarray $A[1..n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

Analysis of Insertion Sort: The time taken by the Insertion-Sort procedure depends on the input: sorting a 1000 numbers takes longer than sorting 3 numbers. In general, the time taken by an algorithm grows with the size of the input. The running time is a function of the ***input size***. The running time also depends on how well sorted the input sequence is.

Input Size: In most cases the input size is the number of items in the input. But in some cases, like multiplying large numbers, the total number of bits in the input is a better measure.

Running Time: The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed. For now, a constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the ith line takes time c_i, where c_i is a constant.

Running Time of Insertion Sort: Let $n = A.length$. Let t_j be the number of times line 5 is executed for that value of j. The number of times each line is executed is shown below: (Notice that for ***while*** and ***for*** loops the condition is executed once more than the body of the loop)

		cost	times
1	**for** $j = 2$ **to** $A.length$	c_1	n
2	$key = A[j]$	c_2	$n-1$
3	// Insert A[j] into the sorted sequence // A[1 .. j – 1]	0	$n-1$
4	$i = j - 1$	c_4	$n-1$
5	**while** $i \succ 0$ and $A[i] \succ key$	c_5	$t_2 + t_3 + ... + t_n$
6	$A[i+1] = A[i]$	c_6	$(t_2 - 1) + (t_3 - 1) + ... + (t_n - 1)$
7	$i = i - 1$	c_7	$(t_2 - 1) + (t_3 - 1) + ... + (t_n - 1)$
8	$A[i+1] = key$	c_8	$n-1$

The running time of the algorithm is:

$\sum$ (cost of statements). (number of times statement is executed). The sum is taken over all statements.

The total running time on the given machine is:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(t_2 + t_3 + ... + t_n) + c_6[(t_2 - 1) + (t_3 - 1) + ... + (t_n - 1)]$$
$$+ c_7[(t_2 - 1) + (t_3 - 1) + ... + (t_n - 1)] + c_8(n-1)$$

The running time depends on the values of t_j. These vary according to the input.

Best-Case: Line 5 is executed just once for each j. This happens if A is already sorted. For all j, $t_j = 1$ and

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

$T(n)$ can be expressed as $an + b$ for constants a and b that depend on the given machine. It is a *linear function* of n.

Worst-Case: Line 5 is executed j times for each j. This can happen if A starts out in reverse order. For all j, $t_j = j$,

$t_2 + t_3 + ... + t_n = 2 + 3 + ... + n = \frac{(n+2)(n-1)}{2}$ and

$(t_2 - 1) + (t_3 - 1) + ...(t_n - 1) = 1 + 2 + ... + n - 1 = \frac{n(n-1)}{2}$. Hence, T(n) could be expressed as:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\frac{(n+2)(n-1)}{2} + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1)$$

Upon simplification, one gets:

$T(n) = an^2 + bn + c$, where a, b, and c are constants that depend on the statement costs c_i. Notice that $T(n)$ is a ***quadratic function*** of n.

Average-Case: We assume that on the average, half of the elements in $A[1..j-1]$are greater than the key and the other half are less. Line 5 is executed $\frac{j}{2}$ times for each j so that $t_j = \frac{j}{2}$. Then

$t_2 + t_3 + ... + t_n = \frac{2}{2} + \frac{3}{2} + ... + \frac{n}{2} = \frac{(n+2)(n-1)}{4}$ and

$$(t_2 - 1) + (t_3 - 1) + ... + (t_n - 1) = (0) + (0.5) + (1) + ... + (\frac{n}{2} - 1)$$

If we simplify the above expression, we get: $\frac{(n^2 - 3n + 2)}{4}$

Hence, we have:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \frac{(n+2)(n-1)}{4} + c_6 \frac{(n^2 - 3n + 2)}{4}. + c_7 \frac{(n^2 - 3n + 2)}{4}. + c_8(n-1)$$

Upon simplification, one gets that:

$T(n) = an^2 + bn + c$, where a, b, and c are constants that depend on the given machine. Notice that T(n) is a ***quadratic function*** of n.

Now, another important concept one likes to consider is the running time for Insertion-Sort for small values of n or for large values of n.

For small values of n, Insertion-Sort will probably run very quickly that we do not care how long it takes.

For large values of n, Insertion-Sort will probably run very slowly, and it will take a long time and its running time is very important.

Consider the worst-case running time for Insertion-Sort:
$T(n) = an^2 + bn + c$, where a, b, and c are constants that depend on the given machine. As n gets larger and larger the an^2 term grows faster than the bn and c terms. As n gets large

$\frac{T(n)}{n^2} = a + \frac{b}{n} + \frac{c}{n^2}$ approaches the constant a and the other two terms go toward 0.

When we analyze any algorithm, we are interested in the running time for large input sizes (large n) and we just consider the dominant term (the term that grows the fastest). Our analysis of algorithms should not depend on any particular machine, so we ignore the constant in front of the dominant term. We only look at the rate of growth as the input size gets larger and larger.

For Insertion-Sort we say the worst-case running time is $\Theta(n^2)$, and the best-case running time is $\Theta(n)$.

In general, when analyzing an algorithm one has to concentrate on the average-case or worst-case. This is because:

a) The worst-case running time of an algorithm is an upper bound on the running time for any input
b) For some algorithms, the worst case occurs fairly often.
c) The average-case is often roughly as bad as the worst case. (See the average-case analysis for the Insertion-sort, it was as bad as the worst-case as seen above)

Designing Algorithms: There are many ways to design algorithms. For example, insertion sort is an ***incremental algorithm***. It builds the sorted sequence one number at a time. Other algorithms use a divide and conquer approach. It simply splits a large problem into a number of smaller sub-problems. Then solve the smaller sub-problems and combine their solutions to solve the large problem. The divide and conquer approach could be summarized in the following steps:
Divide the problem into a number of sub-problems.
Conquer the sub-problems by solving them recursively.
Combine the sub-problem solution to give a solution to the original problem.

Merge Sort: This is a sorting algorithm which is based on the divide and conquer approach. Let $A[1..n]$ be an array of n numbers. The procedure Merge-Sort(A, p, r) sorts the sub-array $A[p..r]$ containing the $r - p + 1$ numbers, a_p through a_r. Calling Merge-Sort(A,1,n) sorts the whole array. Here's how it works:

Divide by splitting into two sub-arrays $A[p..q]$ and $A[q+1..r]$, where q is the halfway point of $A[p..r]$.
Conquer by recursively sorting the two sub-arrays $A[p..q]$ and $A[q+1..r]$.
Combine by merging the two sorted sub-arrays $A[p..q]$and $A[q+1..r]$ to produce a single sorted sub array $A[p..r]$.

Notice that the recursion bottoms out when the sub-array has just 1 element, so that it is trivially sorted. Here is the Merge-Sort algorithm.

Merge-Sort (A, p, r)

```
if p < r                          // check for base case
q = ⌊(p + r)/2⌋                   // Divide
Merge-Sort(A, p, q)               // Conquer
Merge-Sort(A, q+1, r)             // Conquer
Merge(A, p, q, r)                 // Combine
```

Initial Call: Merge-Sort(A, 1, n)

Now let us look at Merge and see how it could merge two lists.

```
Merge(A, p, q, r)
1     n1 = q − p + 1
2     n2 = r − q
3     Let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
4     for i = 1 to n1
5         L[i] = A[p + i − 1]
6     for j = 1 to n2
7         R[j] = A[q + j]
8     L[n1 + 1] = ∞
9     R[n2 + 1] = ∞
10    i = 1
11    j = 1
12    for k = p to r
13        if L[i] ≤ R[j]
14            A[k] = L[i]
15            i = i + 1
16        else A[k] = R[j]
17            j = j + 1
```

Now, let us see that the loop invariant properties hold for Merge-Sort

Initialization: Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p..k-1]$ is empty. This empty subarray contains the $k - p = 0$ smallest elements of L and R, and since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Maintenance: First suppose that $L[i] \le R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p..k-1]$ contains the $k - p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p..k]$ will contain the $k - p + 1$ smallest elements. Incrementing k (in the **for** loop update) and i (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] \succ R[j]$, then lines 16-17 perform the appropriated action to maintain the loop invariant.

Termination: At termination, $k = r + 1$. By the loop invariant, the subarray $A[p..k-1]$, which is $A[p..r]$, contains the $k - p = r - p + 1$ smallest elements of $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$, in sorted order. The arrays L and R together contain $n_1 + n_2 + 2 = r - p + 3$ elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.

Notice that the first **for** loops take $\Theta(n_1 + n_2) = \Theta(n)$ time. The last **for** loop makes n iterations, each taking constant time, for $\Theta(n)$ time. Hence, the worst-case running time for Merge is $\Theta(n)$ where n is the number of elements in the final sorted list.

To simplify the calculations of the worst-case running time of Merge-Sort, we assume $n = 2^p$ for some integer p. The last call to Merge merges n numbers in $\Theta(n)$ time. It was preceded by two calls to Merge which merged $\frac{n}{2}$ numbers each; the time for both calls is $\Theta(n)$. Similarly, those two calls were preceded by four calls to Merge which merged $\frac{n}{4}$ numbers each; the time for the four calls is $\Theta(n)$, etc. There are p levels of calls to Merge and each level has a running time of $\Theta(n)$. Thus, when $n = 2^p$, the running time of Merge-Sort is $\Theta(np)$ or $\Theta(n \lg n)$. It can be shown that for any n the worst-case running time for Merge-Sort is $\Theta(n \lg n)$. Notice that Merge-Sort is faster than Insertion sort; however, Merge-Sort requires extra memory space.

Analyzing divide-and-conquer algorithms

When an algorithm contains a recursive call to itself, we can often describe its running time by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size ***n*** in terms of the running time on smaller inputs. Then we can use some mathematical tools to solve the recurrence and provide bound on the performance of the algorithm.

A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic paradigm. Let $T(n)$ be the running time on a problem of size ***n.*** If the problem size is small enough, say $n \le c$ for some constant ***c,*** the straightforward solution takes constant time, which we write as $\Theta(1)$. Suppose that our division of the problem yields ***a*** subproblems each of which is $1/b$ the size of the original (For merge sort, both ***a*** and ***b*** are 2). It takes time $T(n/b)$ to solve one subproblem of size n/b, and so it takes time $aT(n/b)$ to solve ***a*** of them. If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & if\ n \le 1 \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & otherwise \end{cases}$$

Analysis of merge sort

In order to simplify the recurrence-based analysis, we assume that original problem size is a power of 2. Each divide step then yields two subsequences of size exactly $\frac{n}{2}$. To set up the recurrence for $T(n)$, the worst-case running time of merge sort on *n* numbers. Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

Conquer: We recursively solve, two subproblems, each of size n/2, which contributes $2T\left(\frac{n}{2}\right)$ to the running time.

Combine: The MERGE procedure on an *n*-element subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$

When we add the functions D(n) and C(n) for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of *n*, that is, $\Theta(n)$. Adding it to the 2T(n/2) term from the "conquer" step gives the recurrence for the worst-case running time T(n) of merge sort:

$$T(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & otherwise \end{cases}$$

It can be proved that T(n) is $\Theta(n \lg n)$. Talk about figure 2.5 page 38.