Chapter 2 Getting Started

2.1 Insertion Sort

Sorting problem: Given a sequence of n numbers $(a_1, a_2, \ldots, a_n)$, produce a premutation $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

We present a first solution to the sorting problem, the *insertion sort* algorithm, as a subroutine which takes as parameter an array $A[1..n]$ containing a sequence of length n to be sorted. When the procedure INSERTION-SORT is finished, it rearranges the elements within A so that they are sorted.

INSERTION-SORT(A)

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

Loop Invariants and Correctness: In order to prove for correctness of some algorithms, we can use a *loop invariant*, which is a property of the state of the algorithm which holds during all iterations of the loop. We must show three things in order to prove that a loop invariant holds:

- **Initialization:** It is true prior to the first iteratin 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.

When the first two properties hold, we show that the loop invariant holds prior to every iteration of the loop. The third property helps us use the loop invariant to prove correctness (typically using the loop invariant along the condition which made the loop to terminate).

We can now give a loop invariant of the given INSERTION-SORT algorihtm: *At the start of each iteration of the* ***for*** *loop of lines 1 - 8 the subarray* $A[1..j-1]$ *consists of the elements originally in* $A[1..j-1]$*, but in sorted order.*

Let us see the loop invariant holds and see how it can prove correctness of the sorting algorithm.

- *Initialization:* Before the first iteration, at $j = 2$, the subarray is just $A[1]$ which trivially contains the elements of $A[1]$ in sorted order.
- *Maintenance:* 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 $A[j]$. Which leaves the subarray $A[1..j]$ consisting in elements originally in $A[1..j]$ but in sorted order. Incrementing j for the next iteration of the loop preserves the loop invariant. (A more formal proof would require to prove another loop invariant for this **while** loop.)
- *Termination:* The loop terminates when $j > A.length = n$. Because each iteration increments j by one, we must have $j = n + 1$ at that time. Since the loop invariant holds at the termination time, we see that the array $A[1..n]$ consists of the elements originally in $A[1..n]$ but in sorted order. Hence the entire array A is sorted at the end, hence the algorithm is correct.

2.2 Analyzing Algorithms

Analyzing an algorithm means to predict the resource that the alorithm requires. Such resources may be memory, communication bandwidth, computer hardware, or computational time. Generally, by analyzing several algorihtms for a problem, we can identify a most efficient one (or discard inferior candidate algorithms). We will mostly study the computational time taken by algorithms in this book.

Model: Before we can analyze an algorithm, we must have a *model* of the implementation technology that we will use, including a model for the resources of that technology and their costs. Unless explicited, assume a generic one processor, random-access machine (RAM). That is, instructions executed one after another with no concurrent operations, where each instruction taking constant time (being the set of instroctions in the RAM model the commonly found in real computers: load, store, copy, arithmetic, conditional branch, subroutine call and return.

2.2.1 Analysis of Insertion Sort

The time taken by the INSERTION-SORT procedure depends on the input size, and also depending on how are arranged the item inside the array for two inputs of the same size. In general, running time will be a function of input size.

The notion of *input size* depends on the problem: it may be the number of items in the input (as in the array size in a sorting problem), or the total number of bits to represent input (like the problem of integer multiplication). Sometimes, it is more appropiate to describe the size of input with two parameters (or more), such as in a graph problem with number of edges and number of vertices.

The *running time* of an algorithm on a particular input is the time it takes to execute each primitive operation times the number of executions of those operations. We may assume that a constant amount of time is required to execute each line of pseudocode. That is, each execution of the ith line takes c_i time, where c_i is a constant.

Line Number	Cost	# of Executions
1	c_1	n
2	c_2	$n-1$
3	c_3	$n-1$
4	c_4	$\sum_{j=2}^{n} t_j$
5	c_5	$\sum_{j=2}^{n} (t_j - 1)$
6	c_6	$\sum_{j=2}^{n} (t_j - 1)$
8	c_8	$n-1$

We analyzed above the INSERTION-SORT procedure time cost of each statement and the number of times each statement is executed. For each $j = 2, 3, \ldots, n$, where $n =$

A.length, denote t_j be the number of times the **while** loop in line 5 is executed for that value of j. Then, the execution time of the algorithm is the sum of all the cost of each line times its number of executions:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1).$$

Even for inputs of the same size, an algorithm's running time may depend on which input of that size is given. In this case, the *best case* occurs when the array is already sorted, since for each j we find that $A[i] \leq key$ in line 5 when $i = j-1$ initially, thus having $t_j = 1$. Which yields a running time:

$$\begin{aligned} T_{\text{best}}(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_8)n - (c_2 + c_3 + c_4 + c_8). \end{aligned}$$

We thus can express this running time as $T_{\text{best}}(n) = an+b$ for some constants a, b; thus a linear function of n.

On the other side, if the array is in reverse order, we get the *worst case* of $t_j = j$ for each value of j. And using that $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$, and that $\sum_{j=2}^{n}(j-1) = \frac{n(n+1)}{2} - 1$, we can rearrange the terms as before to get $T_{\text{worst}}(n) = an^2 + bn + c$, for some constants a, b, c: a quadratic function of n.

We could also compute an *average-case* running time for a random input, and compute the expected value of the running time.

2.2.2 Order of Growth

It is the rate of growth or order of growth of the running time that interests us. We therefore consider only the most significative term as the size of input increases. Thus, the worst case running time of Insertion-Sort has a rate of growth of n^2. We sa that it has a worst case running time of $\Theta(n^2)$. We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth.

2.3 Designing algorithms

For insertion sort, we used an *incremental* approach: having sorted $A[1..j-1]$ we then insert $A[j]$ in the proper place, yielding the sorted subarray $A[1..j]$. We examine a different approach now:

2.3.1 Divide and Conquer

Many algorithms are *recursive* in sturcture: to solve a given problem, they call themselves recursively one or more times to deal with a closely related subproblem. These typically follow a *divide-and-conquer* approach:

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** he subproblems by soving them recursively. If the subproblem sizes are small enough, solve the problem in a straightforward manner.
- **Combine** the solutions to the subproblems into the solution for the original problem.

When an algorithm contains a recursive call to itelf, we can describe its running time by a *recurrence*, which describes the overall running time on a problem of size n in terms of the running time of smaller inputs.

In divide and conquer algorithms we can describe the running time as a recurrence. As said, if the problem size is small enough, say $n \leq c$ for some constant c. the straightforward solution takes constant time, write as $\Theta(1)$. Suppose that the division of the problem yields a subproblems, with each one being $1/b$ size of the original. And define $D(n)$ and $C(n)$ be the times to divide the subproblems and combine them, respectively, then we have the running time as:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

The Merge-Sort algorithm follows this divide-and-conquer approach:

- Divide the n-element array to be sorted into two subarrays of $n/2$ elements each.
- Sort the two subarrays recursively using merge sort.
- Merge the two sorted arrays to produce the sorted array.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since it is an already sorted 1-element array.

The Merge-Sort array relies on the auxiliary procedure Merge(A, p, q, r), where A is an array, p, q and r are indices into the array such that $p \leq q < r$. The procedure assumes the subarrays $A[p..q]$ and $A[q+1..r]$ are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$. This procedure can be easily implemented to have a runtime of $\Theta(n)$, where $n = r - p + 1$.

The merge sort algorithm is then:

```
Merge-Sort(A, p, r)
1  if p < r then
2      q = ⌊(p + r)/2⌋
3      Merge-Sort(A, p, q)
4      Merge-Sort(A, q + 1, r)
5      Merge(A, p, q, r)
6  end if
```

And to sort the required sequence A we call the algorithm procedure as Merge-Sort$(A, 1, A.length)$. Assuming that n is a power of 2, we can calculate the running time of the algorithm can be calculated as.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ 2T(n/2) + \Theta(n) & \text{otherwise.} \end{cases}$$

In later chapters we will solve the recurrence using the Master Theorem.

Chapter 3 Growth Of Functions

3.1 Asymptotic notation

When studying the running-time of algorithms, we are interested in the asymptotic behaviour of a time-cost function taking values in the set of natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$.

Definition Let g be a function defined in $\mathbb{N}$. We denote $\Theta(g(n))$ the set of functions defined by:

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2 \in \mathbb{R}^+, \text{ and } n_0 \in \mathbb{N} \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n),\ \forall n \geq n_0\}.$$

If f is another function taking values in $\mathbb{N}$, we denote that f is in this set as "$f(n) = \Theta(g(n))$". We say that g is an *asymptotically tight bound* for f.

Note how the definition of $\Theta(g(n))$ requires that every member f of the set to be *asymptotically nonnegative* ($f(n) \geq 0$ for sufficiently large n). Consequently g itself must be asymptitically nonnegative, or $\Theta(g(n))$ is the empty set.

Definition Let g be a function defined in $\mathbb{N}$. We denote $O(g(n))$ the set of functions defined by:

$$O(g(n)) = \{f(n) : \exists c \in \mathbb{R}^+, \text{ and } n_0 \in \mathbb{N} \text{ such that } 0 \leq f(n) \leq cg(n),\ \forall n \geq n_0\}.$$

If f is another function taking values in $\mathbb{N}$, we denote that f is in this set as "$f(n) = O(g(n))$". We say that g is an *asymptotic upper bound* for f.

Definition Let g be a function defined in $\mathbb{N}$. We denote $\Omega(g(n))$ the set of functions defined by:

$$\Omega(g(n)) = \{f(n) : \exists c \in \mathbb{R}^+, \text{ and } n_0 \in \mathbb{N} \text{ such that } 0 \leq cg(n) \leq f(n),\ \forall n \geq n_0\}.$$

If f is another function taking values in $\mathbb{N}$, we denote that f is in this set as "$f(n) = \Omega(g(n))$". We say that g is an *asymptotic lower bound* for f.

Note from the above definitions that for any function g, we have:

$$\Theta(g(n)) \subseteq \Omega(g(n)), \text{ and } \Theta(g(n)) \subseteq O(g(n)).$$

Theorem 3.1 For any two functions f and g, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Asymptotic notation in equations and inequalities. When using asymptotic notation in formulas, we interpret it as standing for some anonymous function that we do not to care. For example, $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that $2n^2 + 3n + 1 = 2n^2 + f(n)$, for some function f in the set $\Theta(n)$ (In this case $f(n) = 3n + 1$ which is indeed $\Theta(n)$).

Using asymptotic notation in this manner we can help eliminate inessential detail and clutter in an equation. For example in merge sort recurrence: $T(n) = 2T(n/2) + \Theta(n)$. If we are interested only in the asymptotic behavoir of $T(n)$, there is no point in specifying all the lower-order terms exactly; they are all understood to be included in the anonymous function denoted by the term $\Theta(n)$.

The number of anonymous functions in an expression is understood to be equal to the number of times the asymptotic notation appears. In some cases, asymptotic notation appears on the left-hand side of an equation, like: $2n^2+\Theta(n) = \Theta(n^2)$. We interpret it using the rule: *No matter how the anonymous functions are chosen on the left hand of the equals sign, there is a way to choose the anonymous function on the right of the equal sign to make the equation valid.* Thus on our example, $\forall f \in \Theta(n)$, $\exists g \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n), \forall n$.

Asymptotically tight bounds. The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. For example $2n^2 + 3 = O(n^2)$ is asymptotically tight, but $n = O(n^2)$ isn't. We use o-notation to denote an upper bound that is not asymptotically tight.

Definition We denote $o(g(n))$ the set of functions defined by:

$$o(g(n)) = \{f(n) : \forall c > 0,\ \exists n_0 \in \mathbb{N} \text{ such that } 0 \leq f(n) < cg(n),\ \forall n \geq n_0\}.$$

That is equivalent to say that if $f(n) = o(g(n))$ if, and only if, $\lim_{n\to\infty} f(n)/g(n) = 0$.

By analogy we have lower bound that are not asymptotically tight.

Definition We denote $\omega(g(n))$ the set of functions defined by:

$$\omega(g(n)) = \{f(n) : \forall c > 0,\ \exists n_0 \in \mathbb{N} \text{ such that } 0 \leq cg(f < f(n),\ \forall n \geq n_0\}.$$

That is equivalent to say that if $f(n) = \omega(g(n))$ if, and only if, $\lim_{n\to\infty} f(n)/g(n) = \infty$.

3.1.1 Comparing Functions

Many of the relational properties of real numbers apply to asymptotic comparisons as well. For the following assume f, g be asymptotically positive functions.

Symmetry:

i) $f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$.

Reflexivity:

i) $f(n) = \Theta(f(n))$.

ii) $f(n) = O(f(n))$.

iii) $f(n) = \Omega(f(n))$.

Transitivity:

i) $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \implies f(n) = \Theta(h(n))$.

ii) $f(n) = O(g(n))$ and $g(n) = O(h(n)) \implies f(n) = O(h(n))$.

iii) $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n)) \implies f(n) = \Omega(h(n))$.

iv) $f(n) = o(g(n))$ and $g(n) = o(h(n)) \implies f(n) = o(h(n))$.

v) $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n)) \implies f(n) = \omega(h(n))$.

Transpose Symmetry:

i) $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$.

ii) $f(n) = o(g(n)) \iff g(n) = \omega(f(n))$.

We have an analogy between asymptotic comparisons of two functions and the comparisons of two real numbers. For example $f(n) = O(g(n))$ is like $a \leq b$; also $\Omega(\cdot)$ like $\geq$, $\Theta(\cdot)$ like $=$, $o(\cdot)$ like $<$, $\omega(\cdot)$ like $>$.

Remark If we define the relation $f(n) \sim g(n)$ if, and only if, $f(n) = \Theta(g(n))$, then this relation is an equivalence relation.

Chapter 4 Divide And Conquer

In divide-and-conquer, we solve a problem recursively, applying three steps at each level of the recursion:

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solveing them recursively. If the subproblem sizes are small enough, just solve them i a straightforward manner.
- **Combine** the solutions to the subproblems into the solution for the original problem.

When the subproblems are large enough to solve recursively, we call that the *recursive case*. Once the problems become small enough that we no longer recurse, we say that the recursion "bottoms out" and that we have gotten down to the *base case*. Sometimes, in addition to ubproblems that are smaller instances of the same problem, we have to solve subproblems that are NOT quite the same as the original problem, and cconsider solvim that problem part of the combine step.

4.3 The substitution method for solving recurrences

The *substitution method* for solving recurrences comprises two steps:

1. Guess the form of the solution.
2. Use the Principle of Mathematical Induction to find the constants and show that the solution works.

We can use the substitution method to establish either upper or lower bounds on a recurrence. As an example consider the recurrence given by: $T(n) = 2T(\lfloor n/2 \rfloor)+n$. We guess that the solution is $T(n) = O(n \log n)$. We need to see that $T(n) \leq cn \log n$ for some $c > 0$, for all $n > n_0 \in \mathbb{N}$. We start by assuming that the bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$. And substituting into the recurrence yields:

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \leq cn \log(n/2) + n \\ &= cn \log n + (1 - c \log 2)n \leq cn \log n, \end{aligned}$$

where the last inequality holds iff $c \geq 1/\log 2$. To complete the proof we need to check that our solution holds for a base case. That is, we need to show in our case, that we can choose c great enough so that the bound holds for the base case too.

This requeriment may lead to problems sometimes. Assume for the sake of the argument that $T(1) = 1$, then the bound $T(n) \leq cn \log n$ yields to $T(1) \leq c \cdot 1 \cdot \log 1 = 0$!! We can overcome this by taking advantage of asymptotic definition, requiring us that the bound holds for $n \geq n_0$, for some $n_0 \in \mathbb{N}$