2. ALGORITHMIC ANALYSIS

What is an 'Algorithm'? The best formal definition of 'Algorithm' is the one from Wikipedia, *In mathematics and computer science, an algorithm is a finite sequence of well-defined, computer-implementable instructions, used typically to solve a class of problems or to perform some computation.*

The word 'algorithm' has its roots in latinizing the name of the mathematician *Muhammad ibn Musa al-Khwarizmi*. Al-Khwārizmī (c. 780–850) was a mathematician, astronomer, geographer and scholar in the House of Wisdom in Baghdad, whose name means 'the native of Khwarazm', a region that was a part of The Greater Iran and is now in Uzbekistan. About in 825, al-Khwarizmi wrote an Arabic language treatise on the Hindu–Arabic numeral system, which was translated into Latin during the 12th century under the title *Algoritmi de numero Indorum.* This title means *"Algoritmi on the numbers of the Indians"*, where *"Algoritmi"* was the translator's Latinization of Al-Khwarizmi's name.

Not going into the details of the historic events, we now focus on the analysis of algorithms. Before we analyse any algorithm, we must have an algorithm at hand that is supposed to be analysed. So, at this stage, we assume that there is some algorithm which solves a particular class of problems and we have to analyse it. Algorithmic analysis consists of two main parts:

a) *Analysis based on Correctness.* There is no sense of discussing the efficiency of an algorithm if it fails to produce correct output. So, the first and foremost requirement from any algorithm is that it should be correct. Many times, claiming '*it's obvious*' won't suffice to prove that the algorithm is correct. Formally, every algorithm comes with a *proof of correctness.* Some proofs may be intuitive and obvious, while some require a rigorous mathematical treatment. Hence this particular section of analysis is handled by mathematicians, as they are the ones who can provide marvellous proofs and justifications. In Computer Science, one isn't much involved in proving algorithms. Designing an algorithm, proving its correctness mathematically, improving its efficiency, etc. under 30 minutes or so, isn't quite feasible. Hence, programmers work with intuition while designing algorithms. The more they design algorithms, find flaws in it, correct them, practice more and more, the better they get at their intuitive reasoning skills.
   Nonetheless, I would like to provide you a little glimpse of proving algorithms correct mathematically, in this and upcoming chapters.

b) *Analysis based on Efficiency.* Once the algorithm is proven to be correct, by mathematical proof or intuitive reasoning, it's time to analyse its efficiency. This is the analysis where computer scientists are a lot more interested than mathematicians. And you will realize this while reading this book, how 90% of the times we are just concerned about the efficiency. So, without further ado, let's get started.

2.1 Specification of a problem

Before we can design or analyse any software component – including an algorithm or data structure – we must first know what it is supposed to accomplish. A formal statement of what a software component is meant to accomplish is called a *specification.* Here, we will discuss specifically the specification of an algorithm. The specification of a data structure is similar, but a bit more involved.

Suppose, for example, that we wish to find the *kth smallest element* of an array of n numbers. Thus, if *k = 1*, we are looking for the smallest element in the array, or if *k = n*, we are looking for the largest. We will refer to this problem as *the selection problem.* Even for such a seemingly simple problem, there is a potential for ambiguity if we are not careful to state the problem precisely. For example, what do we expect from the algorithm if k is larger than n? Do we allow the algorithm to change the array? Do all of the elements in the array need to be different? If not, what exactly do we mean by *the kth smallest*?

Specifically, we need to state the following:

- *a precondition:* this is the statement of the assumptions we make about the input of the algorithm and the environment in which it will be executed.
- *a postcondition:* this is the statement of the required result of executing the algorithm, assuming the precondition is satisfied.

The precondition and postcondition, together with a function header giving a name and parameter list, constitute the specification of the algorithm. We say that the algorithm meets its specification if the postcondition is satisfied whenever the precondition is satisfied. Note that we guarantee nothing about the algorithm if its precondition is not satisfied.

The selection problem will have two parameters, an array A and a positive integer k. The precondition should specify what we are assuming about these parameters. For this problem, we will assume that the first element of A is indexed by 1 and the last element is indexed by some natural number (i.e., nonnegative integer) n; hence, we describe the array more precisely using the notation A[1…n]. We will assume that each element of A[1…n] is a number; hence, our precondition must include this requirement. Furthermore, we will not require the algorithm to verify that k is within a proper range; hence, the precondition must require that $1 \leq k \leq n$, where $k \in N$ (N is the mathematical notation for the set of natural numbers).

Now let us consider how we might specify a postcondition. We need to come up with a precise definition of the kth smallest element of A[1…n]. Consider first the simpler case in which all elements of A[1…n] are distinct. In this case, we would need to return the element A[i] such that exactly k elements of A[1…n] are less than or equal to A[i]. However, this definition of the kth smallest element might be meaningless when A[1…n] contains duplicate entries. Suppose, for example, that A[1…2] contains two 0s and that k = 1. There is then no element A[i] such that exactly 1 element of A[1…2] is less than or equal to A[i].

Specification 2.1: The Selection Problem:

Precondition: A[1…n] is an array of Numbers, $1 \le k \le n$, k and n are NATs.

Postcondition: Returns the value x in A[1…n] such that fewer than k elements of A[1…n] are strictly less than x, and at least k elements of A[1…n] are less than or equal to x. The elements of A[1…n] may be permuted.

Select (A[1…n], k)

To express the data types of n and the elements of A, we use NAT to denote the natural number type and NUMBER to denote the number type. Note that NAT is a subtype of NUMBER — every NAT is also a NUMBER. In order to place fewer constraints on the algorithm, we have included in the postcondition a statement that the elements of A[1…n] may be permuted (i.e. rearranged). In order for a specification to be precise, the postcondition must state when side-effects such as this may occur. In order to keep specifications from becoming overly wordy, we will adopt the convention that no values may be changed unless the postcondition explicitly allows the change.

2.2 Designing a simple algorithm

Once we have a specification, we need to produce an algorithm to implement that specification. This algorithm is a precise statement of the computational steps taken to produce the results required by the specification. An algorithm differs from a program in that it is usually not specified in a programming language. In this book, we describe algorithms using a notation that is precise enough to be implemented as a programming language, but which is designed to be read by humans.

A straightforward approach to solving the selection problem is as follows:

1. Sort the array in nondecreasing order.
2. Return the kth element.

If we already know how to sort, then we have solved our problem; otherwise, we must come up with a sorting algorithm. By using sorting to solve the selection problem, we say that we have reduced the selection problem to the sorting problem. Solving a problem by reducing it to one or more simpler problems is the essence of the top-down approach to designing algorithms. One advantage to this approach is that it allows us to abstract away certain details so that we can focus on the main steps of the algorithm.

When we reduce the selection problem to the sorting problem, we need a specification for sorting as well.

For this problem, the precondition will be that A[1…n] is an array of NUMBERS, where n ∈ N. Our postcondition will be that A[1…n] is a permutation of its initial values such that for 1 ≤ i < j ≤ n, A[i] ≤ A[j] i.e., that A[1…n] contains its initial values in nondecreasing order. Our simple selection algorithm is given below. Note that Sort() is only specified — its algorithm is not provided.

Algorithm 2.1: The Selection Problem

```
SimpleSelect (A[1…n], k):
  Sort (A[1…n])
  return A[k]
```

Precondition: A[1…n] is an array of Numbers, n is a NAT.

Postcondition: A[1…n] is a permutation of its initial values such that for 1 ≤ i < j ≤ n, A[i] ≤ A[j]. Such a permutation is achieved by Sort (A[1…n]).

Let us now refine the SimpleSelect algorithm by designing a sorting algorithm. We will reduce the sorting problem to the problem of inserting an element into a sorted array. In order to complete the reduction, we need to have a sorted array in which to insert. We have thus returned to our original problem. We can break this circularity, however, by using the top-down approach in a different way.

Specifically, we reduce larger instances of sorting to smaller instances. In this application of the top-down approach, the simpler problem is actually a smaller instance of the same problem. Though this application of the top-down approach may at first seem harder to understand, we can think about it in the same way as we did for SimpleSelect.

If n ≤ 1, the postcondition for Sort(A[1…n]) (in the algorithm given below) is clearly met. For n > 1, we use the specification of Sort to understand that InsertSort([1…n − 1]) sorts A[1…n − 1]. Thus, the precondition for Insert(A[1…n]) is satisfied. We then use the specification of Insert(A[1…n]) to understand that when the algorithm completes, A[1…n] is sorted. The thought process outlined above might seem mysterious because it doesn't follow the sequence of steps in an execution of the algorithm. However, it is a much more powerful way to think about algorithms.

Algorithm 2.2: Sorting an array using Insertion Sort recursively

```
InsertSort (A[1…n]):
   if n > 1:
      InsertSort (A[1…n − 1])
      Insert (A[1…n])
```

Precondition: A[1…n] is an array of NUMBERS such that n is a NAT, and for $1 \le i < j \le n - 1$, $A[i] \le A[j]$.

Postcondition: A[1…n] is a permutation of its initial values such that for $1 \le i < j \le n$, $A[i] \le A[j]$.

To complete the implementations of SimpleSelect and InsertSort, we need an algorithm for Insert. We can again use the top-down approach to reduce an instance of Insert to a smaller instance of the same problem. According to the precondition mentioned above, A[1…n − 1] must be sorted in nondecreasing order; hence, if either n = 1 or A[n] ≥ A[n − 1], the postcondition is already satisfied. Otherwise, A[n − 1] must be the largest element in A[1…n]. Therefore, if we swap A[n] with A[n − 1], A[1…n − 2] is sorted in nondecreasing order and A[n] is the largest element in A[1…n]. If we then solve the smaller instance A[1…n−1], we will satisfy the postcondition.

Algorithm 2.3: Inserting in a sorted array recursively

```
RecursiveInsert (A[1…n]):
    if n > 1 and A[n] < A[n − 1]:
        A[n] ↔ A[n − 1]
        RecursiveInsert (A[1…n − 1])
```

A logical 'and' or 'or' is evaluated by first evaluating its first operand, then if necessary, evaluating its second operand. Therefore, in evaluating the 'if' statement in RecursiveInsert, if n ≤ 1, the second operand will not be evaluated, as the value of the expression must be false. We use the notation x ↔ y to swap the values of variables x and y.

One can also write any recursive algorithm, in an iterative way as well. But before you look at the algorithm, you should know one term – *An Invariant.* An invariant is a property that is held true throughout the working of an algorithm. Suppose you say, for your algorithm to work, at some step, the condition x < y should always be true. The values x and y can change, but the condition should not get violated. Such a condition is called an invariant. You can now have a look at following algorithms and try to understand, how they work.

Algorithm 2.4: Inserting in a sorted array iteratively

```
IterativeInsert (A[1…n]):
    j ← n
    // Invariant: A[1…n] is a permutation of its original values such that
    // for 1 ≤ k < k′ ≤ n,  if k′ ≠ j,  then A[k] ≤ A[k′].
    while j > 1 and A[j] < A[j − 1]:
        A[j] ↔ A[j − 1];    j ← j – 1
```

Algorithm 2.5: Sorting an array using Insertion Sort iteratively

```
InsertionSort (A[1…n]):
    // Invariant: A[1…n] is a permutation of its original values
    // such that A[1…i − 1] is in nondecreasing order.
    for i ← 1 to n:
        j ← i
        // Invariant: A[1…n] is a permutation of its original values
        // such that for 1 ≤ k < k′ ≤ i, if k′ ≠ j, then A[k] ≤ A[k′].
        while j > 1 and A[j] < A[j − 1]:
            A[j] ↔ A[j − 1];
            j ← j – 1
```

After doing all this work, we can say – we have successfully designed a simple algorithm for our *selection problem.* Sigh! You might be thinking that this was a lot of work in designing a simple algorithm for a simple problem. We haven't even discussed whether the proposed algorithm is efficient or not, or whether some sophisticated efficient algorithm exists or not and so on. But trust me, you will never ever design algorithms like this. You will be thinking and solving a lot of complex problems, at least more complex in comparison to our selection problem, within few minutes. This was just a glimpse of how formally an algorithm is designed. Mathematicians who work with algorithms deal with such formal definitions and procedures all the time.

Nonetheless, after all the complex algorithmic design process which we followed, here is one good news: We have our first algorithm ready! We are now ready to have a glimpse of the next topic in the series, *proving the correctness of the algorithm!*

2.3 Proving the algorithm's correctness

The ability to prove the correctness of the program/algorithm is quite possibly the most underrated skill in the entire discipline of computing. People design algorithms or programs and then instead of proving the algorithm for all test cases in general, they perform some tests with all the possible input data they can think of. If they fail to find any counter-input (an input on which algorithm or program produces wrong output), they claim that algorithm to be correct. And doing so is kind of alright, because the proofs are often based on solid mathematical foundations and writing proofs for huge algorithms isn't quite possible. Despite being such an underrated concept, it is not dead yet (and probably it will never ever die). This is due to many reasons.

First, knowing how to prove algorithm's correctness also helps us in the design of algorithms. Specifically, once we understand the mechanics of correctness proofs, we can design the algorithm with a proof of correctness in mind. This approach makes designing correct algorithms much easier. Second, the exercise of working through a correctness proof — or even sketching such a proof — often uncovers subtle errors that would be difficult to find with testing alone. Third, this ability brings with it a capacity to understand specific algorithms at a much deeper level. Thus, the ability to prove algorithm's correctness is a powerful tool for designing and understanding algorithms. So, let's get started, proving some algorithms.

Once you have an algorithm at hand, your first attempt should always be to prove it incorrect. Find an input such that the algorithm yields an incorrect answer. Such input cases are called *counter-examples*. No rational person will ever leap to the defence of an algorithm after a counter-example has been identified. Hunting for counter-examples is a skill worth developing. It bears some similarity to the task of developing test sets for computer programs, but relies more on inspiration than exhaustion. Here are some techniques to aid your quest:

- *Think for smaller inputs:* It is a fact that when algorithms fail, there is usually a very simple example on which they fail. Amateur algorists tend to draw a big messy instance and then stare at it helplessly. The pros look carefully at several small examples, because they are easier to verify and reason about.
- *Hunt for the weaknesses:* If a proposed algorithm is of the form "always take the biggest" (better known as the greedy algorithm), think about why that might prove to be the wrong thing to do. Create situations when everything is of same size and there is no biggest element. Suddenly the algorithm has nothing to base its decision on and perhaps does something inefficient then.
- *Seek extremes:* Many counter-examples are mixtures of huge and tiny, left and right, few and many, near and far. It is usually easier to verify or reason about extreme examples than more muddled ones.

Once you are sure that there are no counter-cases, according to you, you can assume it is correct and try using that code in real-life scenario. Failure to find a counterexample to a given algorithm does not mean "it is obvious" that the algorithm is correct. So, the better option is – try to prove your algorithm to be correct for all instances of input in general.

Once you have developed a good intuition in algorithmic design, you won't need to do this step. But as of now, we assume ourselves as novice programmers and we would like to prove the algorithms.

First consider the algorithm SimpleSelect. This algorithm is simple enough that ordinarily we would not bother to give a formal proof of its correctness; however, such a proof serves to illustrate the basic approach. Recall that in order for an algorithm to meet its specification, it must be the case that whenever the precondition is satisfied initially, the postcondition is also satisfied when the algorithm finishes.

Theorem 2.1: SimpleSelect meets the specification mentioned in Specification 2.1 of section 2.1.

Proof:

First, we assume that the precondition for SimpleSelect(A[1…n]) is satisfied initially; i.e., we assume that initially A[1…n] is an array of numbers, $1 \le k \le n$, and both k and n are natural numbers.

This assumption immediately implies the precondition for Sort(A[1…n]), namely, that A[1…n] is an array of numbers, and that n is a natural number. In order to talk about both the initial and final values of A without confusion, let us represent the final value of A by A′ and use A only to denote its initial value. Because Sort permutes A (we know this from its postcondition), A′ contains the same collection of values as does A.

Suppose $A'[i] < A'[k]$ for some i, $1 \le i \le n$. Then $i < k$, for if $k < i$, $A'[k] > A'[i]$ violates the postcondition of Sort. Hence, there are fewer than k elements of A, and hence of A, with value less than A′[k].

Now suppose $1 \le i \le k$. From the postcondition of Sort, $A'[i] \le A'[k]$. Hence, there are at least k elements of A with value less than or equal to A′[k].

Therefore, the returned value A′[k] satisfies the postcondition of Select. Hence, proved.

Here are few things which you should observe in above proof. Every proof of the algorithm will begin by assuming the precondition to be true and we need to show that by the end of the algorithm, the postcondition is true. If we are able to do so, we have proved the correctness of the algorithm. During the process of going through the algorithm, we reason about each and every statement in the algorithm. If a statement is a call to another algorithm, we use that algorithm's specification. When we shift to other algorithm used as sub-routine, we must ensure that, by the moment we have reached to that step, the precondition of helper algorithm is satisfied. We can finally use the postcondition of the helper algorithm to continue with our reasoning.

Above theorem 2.1 illustrates a common difficulty with the correctness proofs. In algorithms, variables typically change their values as the algorithm progresses. However, in proofs, a variable must maintain a single value in order to maintain consistent reasoning. In order to avoid confusion, we introduce different names for the value of the same variable at different points in the algorithm. It is customary to use the variable name from the algorithm to denote the initial value, and to add a "prime" (′) to denote its final value. If intermediate value must be considered, "double-primes" (″), superscripts, or subscripts can be used.

The proof of theorem 2.1 is straightforward because the flow of the control in the algorithm proceeds sequentially from the first statement to the last. The addition of *if statement* complicates matters only a little bit. However, once we find recursions or loops – reasoning gets bit difficult.

2.4 Handling Recursion and Iterations in Proofs

We would split this section into two parts:

a) Handling Recursion
b) Handling Iterations

Case a) – Handling Recursion

When I first learnt about mathematical induction it seemed like complete magic. You proved a formula like $\sum_{i=1}^{n} i = n(n+1)/2$ for some basis case like 1 or 2, then assumed it was true all the way to n − 1 before proving it was true for general n using the assumption. That was a proof? Ridiculous!

When I first learned the programming technique of recursion it also seemed like complete magic. The program tested whether the input argument was some basis case like 1 or 2. If not, you solved the bigger case by breaking it into pieces and calling the subprogram itself to solve these pieces. That was a program? Ridiculous!

The reason both seemed like magic is because recursion is mathematical induction. In both, we have general and boundary conditions, with the general condition breaking the problem into smaller and smaller pieces. The initial or boundary condition terminates the recursion. Once you understand either recursion or induction, you should be able to see why the other one also works.

I've heard it said – that a computer scientist is a mathematician who only knows how to prove things by induction. This is partially true because computer scientists are lousy at proving things, but primarily because so many of the algorithms we study are either recursive or incremental, and induction provides an easy way to handle these things.

Let us now consider InsertSort of Algorithm 2 of section 2.2. To handle the if statement, we can consider two cases, depending on whether n > 1. In case n > 1, there is a recursive call to InsertSort. As suggested, we might simply use the specification of InsertSort just like we would for a call to any other algorithm. However, this type of reasoning is logically suspect — we are assuming the correctness of InsertSort in order to prove the correctness of InsertSort. In order to break this circularity, we use the *principle of mathematical induction*. The way we would use induction here is by proving the property that InsertSort(A[1…n]) satisfies its specification for every natural number n.

Suppose we denote the property, that we wish to be true for n in general, by P; then the readers who are familiar with induction can guess that our proof will involve three steps:

a) *The base case.* P(0) or P(1) is shown to be true.
b) *The induction hypothesis.* P(n) is assumed to be true for some arbitrary natural number.
c) *The induction step.* Using the hypothesis, P(n+1) is proved.

We will now illustrate these three steps on InsertSort.

Theorem 2.2: InsertSort, given as Algorithm 2.2 in section 2.2, satisfies the specification of Sort given in Algorithm 2.1.

Proof: (By induction on n)

Base Case: *n ≤ 1.* In this case the algorithm does nothing, but its postcondition is vacuously satisfied (i.e., there are no i, j such that $1 \le i < j \le n$).

Induction Hypothesis: Assume that for some n > 1, for every k < n, InsertSort(A[1…k]) satisfies its specification.

Induction Step: We first assume that initially, the precondition for InsertSort(A[1…n]) is satisfied. Then the precondition for InsertSort(A[1…n − 1]) is also initially satisfied. By the Induction Hypothesis, we conclude that InsertSort(A[1…n − 1]) satisfies its specification; hence, its postcondition holds when it finishes. Let A″ denote the value of A after InsertSort(A[1…n − 1]) finishes. Then A″[1…n − 1] is a permutation of A[1…n − 1] in nondecreasing order, and A″[n] = A[n]. Thus, A″ satisfies the precondition of Insert. Let A′ denote the value of A after Insert(A[1…n]) is called. By the postcondition of Insert, A′[1…n] is a permutation of A[1…n] in nondecreasing order.

InsertSort therefore satisfies its specification. Hence, proved.

Case b) – Handling Iterations

In order to reason formally about the loops, what we need are *the invariants*. We know that the invariants are the properties that must hold true, throughout the process of the algorithm. We know that we can prove the correctness of recursion by induction in 3 steps. Likewise, we have 3 well-defined steps to prove the correctness of the loops. Suppose we wish to show that property P holds upon the completion of the loop, we have to show each of the following steps:

a) *Initialization:* The invariant holds prior to the first iteration of the loop.
b) *Maintenance:* If the invariant holds at the beginning of an arbitrary loop iteration, then it must also hold at the end of that iteration.
c) *Termination:* The loop always terminates.

The idea of the correctness comes from the fact that – the loop invariant is true at anytime of the loop i.e. the property is maintained & the loop terminates. Hence, at the end of the loop, the property should also hold.

Before we handle the cases of nested loops, look at the Algorithm 2.4 of IterativeInsert, given in section 2.2. The while loop in this algorithm is pretty trivial. There are two conditions at which the while loop will terminate. Either we have found the correct position of the element based on inequality or in extreme case, the value of j will eventually become 1 as it gets decreased with every iteration of loop. So, at max the loop will run j times.

In short, proving the while loop's termination is very trivial in this case. However, there are many cases in which proving the termination condition for while is a lot harder. For example:

The Collatz Conjecture

```
while n > 1:
    if n mod 2 = 0:
        n ← n/2
    else:
        n ← 3n + 1
```

In mathematics, a conjecture is a conclusion or a proposition which is suspected to be true due to preliminary supporting evidence, but for which no proof or disproof has yet been found.

One such famous conjecture is *The Collatz Conjecture.* It says that: *"Start with any positive integer n. Then each subsequent term is obtained from the previous term as follows: If the previous term is even, the next term is half of the previous term. If the previous term is odd, the next term is thrice the previous term plus one. The conjecture is that - No matter what the value of n is, the sequence will always reach 1."*

This conjecture was proposed by Lothar Collatz in 1937, and is also famously known as *3n+1 conjecture.* Jeffrey Lagarias, mathematician and professor at University of Michigan, in 2010 claimed that based only on known information about this problem, *"this is an extraordinarily difficult problem, completely out of reach of the present-day mathematics."* The point I am trying to make here is – you saw one trivial while loop whose terminating condition was obvious and you saw one nontrivial while loop whose terminating condition has not been discovered by mathematicians yet. So, you can fairly expect all different type of while loops in this difficulty range.

Let us now have a look at Algorithm 2.5 of InsertionSort, given in section 2.2. When loops are nested, we apply the same technique to each loop as we encounter it. Specifically, in order to prove maintenance for the outer loop, we need to prove that the inner loop satisfies some correctness property, which should in turn be sufficient to complete the proof of maintenance for the outer loop. Thus, nested within the maintenance step of the outer loop is a complete proof (i.e., initialization, maintenance and termination) for the inner loop.

When we prove initialization for the inner loop, we are not simply reasoning about the code leading to the first execution of that loop. Rather, we are reasoning about the code that initializes the loop on any iteration of the outer loop. For this reason, we cannot consider the initialization code for the outer loop when proving the initialization step for the inner loop. Instead, because the proof for the inner loop is actually a part of the maintenance proof for the outer loop, we can use any facts available for use in the proof of maintenance for the outer loop.

Specifically, we can use the assumption that the invariant holds at the beginning of the outer loop iteration, and we can reason about any code executed prior to the inner loop during this iteration. We must then show that the invariant of the inner loop is satisfied upon executing this code.

We will now illustrate this technique by giving a complete proof that InsertionSort meets its specification.

Theorem 2.3: InsertionSort meets its specification.

Proof: We have to show that when the for loop finishes, A[1…n] is a permutation of its original values in non-decreasing order.

Initialization (Outer loop):

When the loop begins, i = 1 and the contents of A[1…n] have not been changed. Because A[1…i − 1] is an empty array, it is in nondecreasing order.

Maintenance (Outer loop):

Suppose the invariant holds at the beginning of some iteration. Let A′[1…n] denote the contents of A at the end of the iteration, and let i′ denote the value of i at the end of the iteration. Then i′ = i + 1. We must show that the while loop satisfies the correctness property that A′[1…n] is a permutation of the original values of A[1…n], and that A′[1…i′ − 1] = A′[1…i] is in non-decreasing order.

Initialization (Inner loop):

Because A[1…n] has not been changed since the beginning of the current iteration of the outer loop, from the outer loop invariant, A[1…n] is a permutation of its original values. From the outer loop invariant, A[1…i − 1] is in non-decreasing order; hence, because j = i, we have for $1 \le k < k' \le i$, where $k' \ne j$, $A[k] \le A[k']$.

Maintenance (Inner loop):

Suppose the invariant holds at the beginning of some iteration. Let A′[1…n] denote the contents of A[1…n] following the iteration, and let j′ denote the value of j following the iteration. Hence,

i. A′[j] = A[j − 1];

ii. A′[j − 1] = A[j];

iii. A′[k] = A[k] for $1 \le k \le n$, $k \ne j$, and $k \ne j - 1$; and

iv. j′ = j − 1.

Thus, A′[1…n] is a permutation of A[1…n]. From the invariant, A′[1…n] is therefore a permutation of the original values of A[1…n]. Suppose $1 \le k < k' \le i$, where $k' \ne j' = j - 1$.

We must show that $A'[k] \le A'[k']$. We consider three cases:

Case 1: k′ < j − 1. Then A′[k] = A[k] and A′[k′] = A[k′]. From the invariant, A[k] ≤ A[k′]; hence, A′[k] ≤ A′[k′].

Case 2: k′ = j. Then A′[k′] = A[j − 1]. If k = j − 1, then A′[k] = A[j], and from the while loop condition, A[j] < A[j − 1]. Otherwise, k < j − 1 and A′[k] = A[k]; hence, from the invariant, A[k] ≤ A[j − 1]. In either case, we conclude that A′ [k] ≤ A′[k′].

Case 3: k′ > j. Then A′[k′] = A[k′], and A′[k] = A[l], where l is either k, j, or j − 1. In each of these cases, l < k′; hence, from the invariant, A[l] ≤ A[k′]. Thus, A′[k] ≤ A′[k′].

Termination (Inner loop):

Each iteration decreases the value of j by 1; hence, if the loop keeps iterating, j must eventually be no greater than 1. At this point, the loop will terminate.

Correctness of Inner loop:

Let A′[1…n] denote the contents of A[1…n] when the while loop terminates, and let i and j denote their values at this point. From the invariant, A[1…n] is a permutation of its original values. We must show that A′[1…i] is in nondecreasing order. Let 1 ≤ k < k′ ≤ i. We consider two cases.

Case 1: k′ = j. Then j > 1. From the loop exit condition, it follows that A′[j − 1] ≤ A′[j] = A′[k′]. From the invariant, if k ≠ j − 1, then A′[k] ≤ A′[j − 1]; hence, regardless of whether k = j − 1, A′[k] ≤ A′[k′].

Case 2: k′ ≠ j. Then from the invariant, A′[k] ≤ A′[k′].

This completes the proof for the inner loop, and hence the proof of maintenance for the outer loop.

Termination (Outer loop):

Because the loop is a range-based for loop, it must terminate.

Correctness of Outer loop:

Let A′[1…n] denote its final contents. From the invariant, A′[1…n] is a permutation of its original values. From the loop exit condition ($i = n + 1$) and the invariant, A′[1…n] is in non-decreasing order.

Therefore, the postcondition is satisfied. Hence, proved.

With the end of this proof, pat your back. Congrats! You just completed the basics of how to prove the algorithms correct. I am sure that even this little knowledge will help you a lot, if in case you ever stumbled upon some algorithm which is written in mathematical way. You will find these concepts very commonly if you grabbed some research paper on any famous algorithm. And if you ever come to read them, hopefully you will be familiar with the words, even if you may or may not be able to understand the actual intent of its usage.

Finally, one thing which you must have realized should be – *how much difficult it is, especially for the students of computer science, to formally prove the correctness of the algorithm!* Hence, there is a lot greater need for developing the right design intuition, with practice and theory of algorithmic design techniques, so that we could design the algorithms which are correct right from first place.

With the end of our first part of analysis, we are now set to start exploring another domain of analysis, which is very very very important to us – *The Complexity Analysis!*

2.5 Motivation behind Complexity Analysis

'Complexity Analysis' is a formal tool that helps us to measure how much better a program/an algorithm is, in comparison to other. *But why do we need such analysis?* Let's start by motivating the topic a little bit.

We already know that there are tools to measure how fast a program runs. These programs are called *profilers* which measure the running time of a program in milliseconds and can help us optimize our code by spotting bottlenecks. While this is a useful tool, it isn't really relevant to algorithm complexity. Algorithm complexity is something designed to compare two algorithms at the idea level; ignoring low-level details such as the implementation programming language, the hardware the algorithm runs on, or the instruction set of the given CPU. We want to compare algorithms in terms of just what they are: Ideas of how something is computed. Counting milliseconds won't help us in that. It's quite possible that a bad algorithm written in a low-level programming language such as Assembly runs much quicker than a good algorithm written in a high-level programming language such as Python or Ruby.

Complexity analysis not only helps us define – what a better algorithm really is, but also allows us to explain how an algorithm behaves when the input grows larger. If we feed it a different input, *how will the algorithm behave*? If our algorithm takes 1 second to run for an input of size 1000, how will it behave if I double the input size? *Will it run just as fast, half as fast, or four times slower?* For example, if we've made an algorithm for a web application that works well with 1000 users and measure its running time, using algorithm complexity analysis we can have a pretty good idea of what will happen once we get 2000 users instead. For algorithmic competitions, complexity analysis gives us insight about how long our code will run for the largest testcases that are used to test our program's correctness.

So, if we've measured our program's behaviour for a small input, we can get a good idea of how it will behave for larger inputs.

Let's start by a simple example: *Finding the maximum element in an array.*

Counting Instructions:

The maximum element in an array can be looked up using a simple piece of code given below.

```
int m = arr[0];
for (int i = 0; i < n; i++) {
    if (arr[i] >= m)
        m = arr[i];
}
```

Now the first thing we'll do is – count how many *fundamental instructions* this piece of code executes. We will do this only once in this entire book and that too for this example only. Once you develop the theory, you will be able to analyse time complexity without counting instructions.

By *fundamental instructions,* we assume that our processor can do following things in one instruction (*let's make this hypothetical assumption for now*):

- Assigning a value to a variable
- Looking up for a particular element in an array
- Comparing two values
- Incrementing a value
- Basic arithmetic operations like addition, subtraction, division and multiplication. *(Even though multiplication and division are costlier as compared to addition, we assume all of these take 1 instruction.)*

The first line of code is:

```
int m = arr[0];
```

This requires 2 instructions: one for looking arr[0] and one for assigning the value to m. These 2 instructions are always required by this algorithm, irrespective of value of *n* as far as there is at least one element in the array.

Two more instructions will always get executed in *for* loop's initialization part, even before *for* lop starts executing.

`i = 0` and `i < n`

So, in total there are 4 instructions that get executed irrespective of the size of the array.

After each *for* loop iteration, we need two more instructions to run to check if we stay in the loop:

`i++` and `i < n`

So, if we ignore the body of the loop, the number of instructions this algorithm needs are 4 + 2n. That is, 4 instructions at the beginning of the *for* loop and 2 instructions at the end of each iteration of loop until n.

We can now define a mathematical function $f(n)$, such that, given the size of input n as parameter, it gives us the number of instructions the algorithm needs. So, for an empty *for loop,*

$$f(n) = 4 + 2n.$$

Worst Case Analysis:

When we start analysing the body of *for loop* as well, defining $f(n)$ is not easy. This is because – our number of instructions doesn't solely depend on n but also on our input. For example, for arr = [1, 2, 3, 4], the algorithm will take a greater number of instructions than for arr = [4, 3, 2, 1]. This is because, in prior case, the statements inside *if block* are also executed. In computer science, we are always interested in finding the worst-case scenarios.

The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken in any instance of size n. Similarly, the *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken in any instance of size n; and the *average-case complexity* of the algorithm, which is the function defined by the average number of steps over all instances of size n.

The worst-case complexity proves to be most useful of these three measures in practice. Many people find this counter-intuitive. To illustrate why, try to project what will happen if you bring n dollars into a casino to gamble. The best case, that you walk out owning the place, is possible but so unlikely that you should not even think about it. The worst case, that you lose all n dollars, is easy to calculate and distressingly likely to happen. The average case, that the typical bettor loses 87.32% of the money that he brings to the casino, is difficult to establish and its meaning subject to debate. What exactly does average mean? Stupid people lose more than smart people, so are you smarter or stupider than the average person, and by how much? Card counters at blackjack do better on average than customers who accept three or more free drinks. We avoid all these complexities and obtain a very useful result by just considering the worst case.

So, in worst-case, we have 4 instructions to run within *for loop.* So, we have –

$$f(n) = 4 + 2n + 4n = 6n + 4$$

Asymptotic behaviour:

Given such a function, we can have a pretty good idea of how fast an algorithm is. However, as I promised, we won't be needing to go through this tedious task of counting instructions in our program. The exact function depends upon compiler (i.e. programming language) and on CPU's instruction set. We said that we will

ignoring this minor details. Hence, we will pass our function $f(n)$ through a filter, which will help us get rid of those minor details.

In our function, we have two terms – 6n and 4. In complexity analysis, we only care about what happens to the instruction-counting function as the program input's size grows larger. Hence, we will first drop all those terms which grow very slowly and only keep the ones that grow very fast, as n becomes larger. Therefore, the first thing which we will do is – drop that 4 and just keep 6n. Hence,

$$f(n) = 6n$$

And doing so makes sense if you think about it; as 4 is just an initialization constant. Different programming languages may require a different time to set up. For example, Java needs some time to initialize its virtual machine. Since we're ignoring programming language differences, it only makes sense to ignore this value.

The second thing we'll ignore is the constant multiplier in front of n, and so our function will become $f(n) = n$. As you can see this simplifies things quite a lot. Again, it makes some sense to drop this multiplicative constant if we think about how different programming languages compile. The "array lookup" statement in one language may compile to different instructions in different programming languages. For example, in C, doing A[i] does not include a check that i is within the declared array size, while in Pascal it does. So, the following Pascal code:

```
M := A[i]
```

Is equivalent to following C/C++ code:

```
if (i >= 0 && i < n) {
    M = A[i];
}
```

So, it's reasonable to expect that different programming languages will yield different factors when we count their instructions. In our example in which we are using a dumb compiler for Pascal that is oblivious of possible optimizations, Pascal requires 3 instructions for each array access instead of the 1 instruction C requires. Dropping this factor goes along the lines of ignoring the differences between particular programming languages and compilers and only analysing the idea of the algorithm itself.

This filter of "dropping all factors" and of "keeping the largest growing term" as described above is what we call asymptotic behaviour. So, the asymptotic behaviour of $f(n) = 2n + 8$ is described by the function $f(n) = n$.

Let us find the asymptotic behaviour of the following example functions by dropping the constant factors and by keeping the terms that grow the fastest.

1. $f(n) = 5n + 12$ gives $f(n) = n$.
2. $f(n) = 109$ gives $f(n) = 1$. We're dropping the multiplier 109 * 1, but we still have to put a 1 here to indicate that this function has a non-zero value.

3. $f(n) = n^2 + 3n + 112$ gives $f(n) = n^2$. Here, n^2 grows larger than $3n$ for sufficiently large n, so we're keeping that.
4. $f(n) = n^3 + 1999n + 1337$ gives $f(n) = n^3$. Even though the factor in front of n is quite large, we can still find a large enough n so that n^3 is bigger than $1999n$. As we're interested in the behaviour for very large values of n, we only keep n^3.
5. $f(n) = n + sqrt(n)$ gives $f(n) = n$. This is so because n grows faster than $sqrt(n)$ as we increase n.

2.6 Orders of Growth

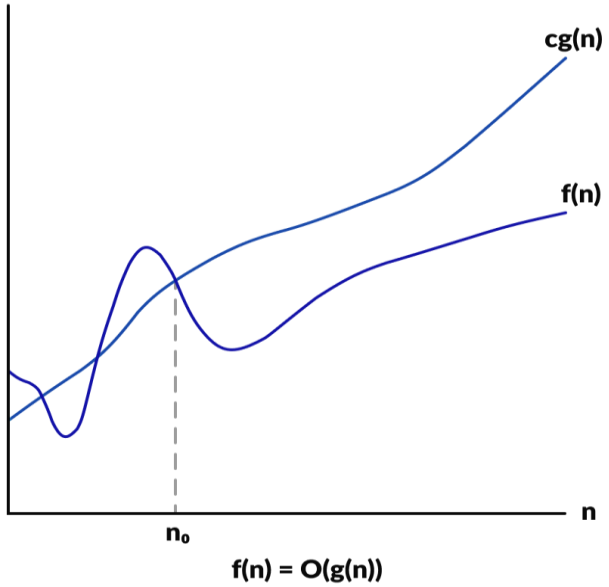
Here I would like to introduce to you – 3 notations, that are very common in complexity analysis as they help us to define the asymptotic growth more formally.

Notation 1: O – Notation (Big O Notation)

This notation gives an upper bound for a function to within a constant factor. We write,

$$f(n) = O(g(n))$$

if there are positive constants n_0 and c, such that to the right of n_0 (i.e. all values greater than n_0), the value of $f(n)$ always lies on or below $c.g(n)$.



If $f(n) = 4n^2 + 5n + 92$, then we know that, there exists some constants n_0 & c, such that, each of the following $g(n)$ satisfies the condition $c.g(n) \geq f(n)$, for all $n \geq n_0$.

- $g(n) = n^3$
- $g(n) = n^2$
- $g(n) = n^4$

It is accurate to say that n^4 is the upper bound of $4n^2 + 5n + 92$. But such an upper bound is called *a loose upper bound.*

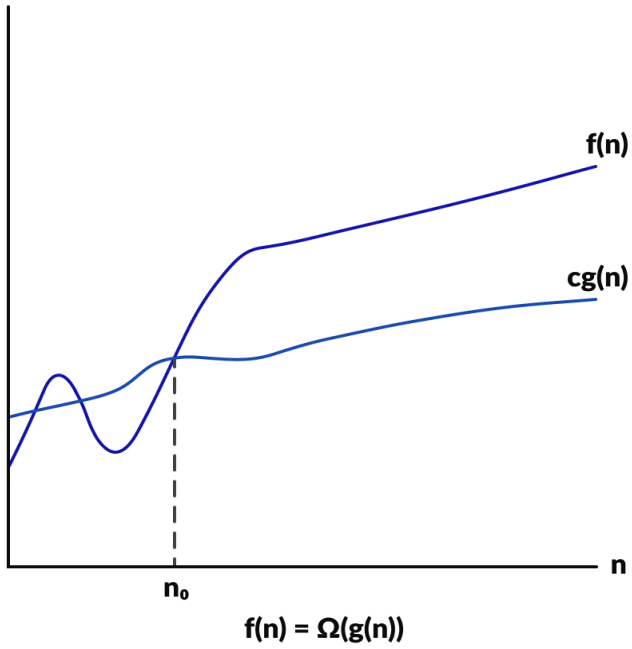
On the other hand, beyond certain n_0, even $5n^2$ will also surpass $4n^2 + 5n + 92$. So, n^2 is also an upper bound but it is *a tight upper bound.*

Notation 2: Ω – Notation (Omega Notation)

This notation gives a lower bound for a function to within a constant factor. We write,

$$f(n) = \Omega(g(n))$$

if there are positive constants n_0 and c, such that to the right of n_0 (i.e. all values greater than n_0), the value of $f(n)$ always lies on or above $c.g(n)$.



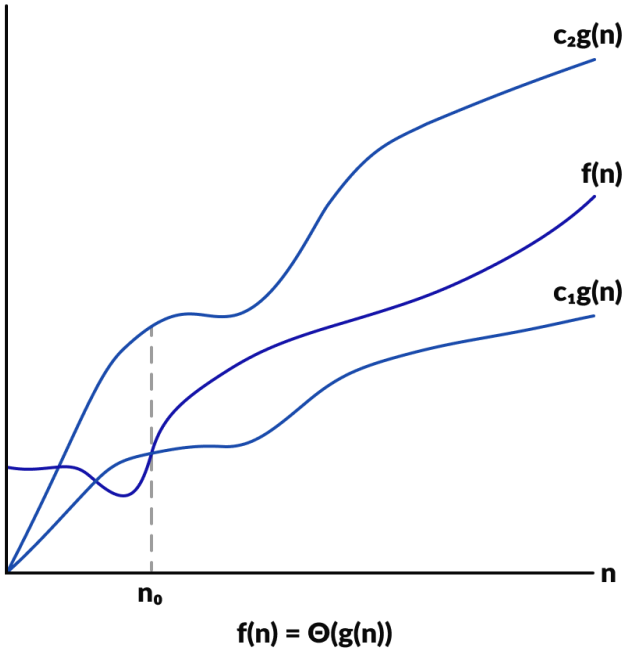
Notation 3: Θ – Notation (Theta Notation)

This notation, encloses the function from above and below. This means it gives a function that grows with same order as that of $f(n)$. We write,

$$f(n) = \theta(g(n))$$

if there exist two constants c_1 and c_2, such that the function $f(n)$ always lies between $c_1.g(n)$ and $c_2.g(n)$.

Here, $\theta(g(n))$ is also called – asymptotically tight bound notation.



2.7 Calculating Time Complexities

Calculating time complexities is very important whenever you write some code. But more importantly, this is a tool which is used by many problem-solvers and competitive programmers to guess if their code will run within a stipulated amount of time or not. So, let's understand some basic principles of calculating time complexity.

Loops:

A common reason why an algorithm is slow is that it contains many loops that go through the input. The more nested loops the algorithm contains, the slower it is. If there are k nested loops, the time complexity is $O(n^k)$.

For example, the time complexity of the following code is $O(n)$:

```
for (int i = 1; i <= n; i++) {
    // constant number of fundamental instructions
}
```

And the time complexity of the following code is $O(n^2)$:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // constant number of fundamental instructions
    }
}
```

Order of Magnitude:

A time complexity does not tell us the exact number of times the code inside a loop is executed, but it only shows the order of magnitude. In the following examples, the code inside the loop is executed $3n, n+5$ and $n/2$ times, but the time complexity of each code is $O(n)$.

```
for (int i = 1; i <= 3*n; i++) {
    // constant number of fundamental instructions
}
```

```
for (int i = 1; i <= n+5; i++) {
    // constant number of fundamental instructions
}
```

```
for (int i = 1; i <= n; i += 2) {
    // constant number of fundamental instructions
}
```

As another example, the time complexity of following code is $O(n^2)$:

```
for (int i = 1; i <= n; i++) {
    for (int j = i+1; j <= n; j++) {
        // constant number of fundamental instructions
    }
}
```

Phases:

If the algorithm consists of consecutive phases, the total time complexity is the largest time complexity of a single phase. The reason for this is that the slowest phase is usually the bottleneck of the code. For example, the following code consists of three phases with time complexities $O(n)$, $O(n^2)$ and $O(n)$. Thus, the total time complexity is $O(n^2)$.

```
for (int i = 1; i <= n; i++) {
    // constant number of fundamental instructions
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // constant number of fundamental instructions
    }
}
```

```
for (int i = 1; i <= n; i++) {
    // constant number of fundamental instructions
}
```

Multiple Parameters:

Sometimes the time complexity depends on several factors. In this case, the time complexity formula contains several variables.

For example, the time complexity of the following code is $O(nm)$:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        // constant number of fundamental instructions
    }
}
```

Recursion:

The time complexity of a recursive function depends on the number of times the function is called and the time complexity of a single call. The total time complexity is the product of these values.

For example, consider the following function:

```
void f(int n) {
    if (n == 1)
        return;
    f(n-1);
}
```

The call $f(n)$ causes n function calls, and the time complexity of each call is $O(1)$. Thus, the total time complexity is $O(n)$.

As another example, consider the following function:

```
void g(int n) {
    if (n == 1)
        return;
    g(n-1);
    g(n-1);
}
```

In this case each function call generates two other calls, except for n = 1. Let us see what happens when g is called with parameter n. The following table shows the function calls produced by this single call:

Function call	*Number of calls*
$g(n)$	1
$g(n-1)$	2
$g(n-2)$	4
…	…
$g(1)$	2^{n-1}

Based on this, the time complexity is,

$$1 + 2 + 4 + \cdots + 2^{n-1} = 2^n - 1 = O(2^n)$$

Notice how adding one more recursive call turns a linear program to exponential. Hence, we need to be extra careful while dealing with recursive functions.

Thumb Rules:

Here I wish to present 3 points which I felt were pretty awesome and useful while calculating time complexities. You can find such tips more online, but these are few which I am presenting here.

Point 1 – Guessing the approximate running time of program based on the constraints.

By calculating the time complexity of an algorithm, it is possible to check, before implementing the algorithm, that it is efficient enough for the problem or not. The starting point for estimations is the fact that a modern computer can perform some hundreds of millions of operations in a second. But for safer side, we will assume that our computer can execute only 10^8 fundamental instructions per second.

For example, assume that the time limit for a problem is one second and the input size is $n = 10^5$. If the time complexity is $O(n^2)$, the algorithm will perform about $(10^5)^2 = 10^{10}$ operations. This should take at least some tens of seconds (near about 100 seconds with our assumption). So, the algorithm seems to be too slow for solving the problem. On the other hand, given the input size, we can try to guess the required time complexity of the algorithm that solves the problem.

The following table contains some useful estimates assuming a time limit of one second.

Input size	*Required Complexity*
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n)$ *or* $O(n \log n)$
$n \geq 10^8$	$O(1)$ *or* $O(\log n)$

Point 2 – Sum of $1/x_i$

Try to find the complexity of following code snippet:

```
void func() {
    for (int i = 0; i <= n; i++) {
        for (int j = i; j <= n; j += i)
            cout << "Hello!" << endl;
    }
}
```

If you will analyse properly, you will find that the number of times the loop is running (i.e. the number of times we expect 'Hello' to get printed) is as follows:

$$f(n) = n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} + \cdots + \frac{n}{n} = n\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right) = n\sum_{i=1}^{n}\frac{1}{i}$$

In above expression, we are a lot more interested in the summation term. One trick which many people use in such cases (i.e. whenever they see summations), is that they replace the summation with an integral. From your basic calculus theory, you must be knowing,

$$\int_1^n \frac{1}{i} = \log n$$

And that's indeed what the actual answer is! The time complexity of above code is $O(n \log n)$.

Point 3 – Sum of $\frac{1}{primes}$

This you won't encounter many times, but since the result is so beautiful, that I can't resist myself from sharing this with you.

Here is the code snippet which finds prime numbers. The algorithm is known as *Sieve of Eratosthenes*. If you don't understand the algorithm, no worries. You should just focus on the time complexity calculation. Here is the code:

```
vector<int> primes;
void sieve() {
    bool is_composite[n];
    memset(is_composite, 0, sizeof(is_composite));
    for (int i = 2; i <= n; i++) {
        if (!is_composite[i]) {
            primes.push_back(i);
            for (int j = i; j <= n; j += i)
                is_composite[i] = 1;
        }
    }
}
```

(One additional point for those who understood this program a little: memset() is used to initialize the entire array. However, this initialization works with only 0 or -1. This is because memset() either sets all bits to 0 or 1, without worrying about the type of the array/container. In 2's complement form, all 0s means 0, and all 1s means -1.)

The above code is faster than $O(n.\log n)$ because we are not running the inner *for loop* for all the values of i. We are doing that only when i is prime. So, the time complexity is,

$$f(n) = \frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \cdots + \frac{n}{last\ prime\ less\ than\ n}$$

$$\therefore f(n) = n\left(\sum \frac{1}{primes\ upto\ n}\right)$$

Here is one result which I wish you to accept it (i.e. without proof), that the sum of the reciprocals of the primes is of order $\log(\log n)$. Explaining the proof is beyond the scope of this book. But people, who are interested enough, can explore this result on their own from this article:

https://en.wikipedia.org/wiki/Divergence_of_the_sum_of_the_reciprocals_of_the_primes

$$\therefore f(n) = O(n.\log\log n)$$

The term $\log\log n$ is a very slow growing term. Even if n is as large as 10^7, the value of this term is approximately 5. So, this can be considered to have almost linear time complexity.

2.8 Case-Study: Maximum Sum Subarray Problem

There are often several possible algorithms for solving a problem such that their time complexities are different. This section discusses a classic problem that has a straightforward $O(n^3)$ solution. However, by designing a better algorithm, it is possible to solve the problem in $O(n^2)$ time and even in $O(n)$ time.

Given an array of n numbers, our task is to calculate the maximum subarray sum, i.e., the largest possible sum of a sequence of consecutive values in the array. The problem is interesting when there may be negative values in the array. For example, in the array

−1	2	4	−3	5	2	−5	2

the following subarray produces the maximum sum 10:

−1	2	4	−3	5	2	−5	2

We assume that an empty subarray is allowed, so the maximum subarray sum is always at least 0.

Algorithm 1:

A straightforward way to solve the problem is to go through all possible subarrays, calculate the sum of values in each subarray and maintain the maximum sum. The following code implements this algorithm:

```
void maximumSubarray() {
    int best = 0;
    for (int a = 0; a < n; a++) {
        for (int b = a; b < n; b++) {
            int sum = 0;
            for (int k = a; k <= b; k++) {
                sum += array[k];
            }
            best = max(best, sum);
        }
    }
    cout << best << "\n";
}
```

The variables a and b fix the first and last index of the subarray, and the sum of values is calculated to the variable sum. The variable best contains the maximum sum found during the search. The time complexity of the algorithm is $O(n^3)$, because it consists of three nested loops that go through the input.

Algorithm 2:

It is easy to make Algorithm 1 more efficient by removing one loop from it. This is possible by calculating the sum at the same time when the right end of the subarray moves.

The result is the following code:

```
void maximumSubarray() {
    int best = 0;
    for (int a = 0; a < n; a++) {
        int sum = 0;
        for (int b = a; b < n; b++) {
            sum += array[b];
            best = max(best, sum);
        }
    }
    cout << best << "\n";
}
```

After this change, the time complexity is $O(n^2)$.

Algorithm 3:

Surprisingly, it is possible to solve the problem in $O(n)$ time, which means that just one loop is enough. The idea is to calculate, for each array position, the maximum sum of a subarray that ends at that position. After this, the answer for the problem is the maximum of those sums.

Consider the subproblem of finding the maximum-sum subarray that ends at position k. There are two possibilities:

- The subarray only contains the element at position k.
- The subarray consists of a subarray that ends at position $k - 1$, followed by the element at position k.

In the latter case, since we want to find a subarray with maximum sum, the subarray that ends at position $k - 1$ should also have the maximum sum. Thus, we can solve the problem efficiently by calculating the maximum subarray sum for each ending position from left to right.

The following code implements the algorithm:

```
void maximumSubarray() {
    int best = 0, sum = 0;
    for (int k = 0; k < n; k++) {
        sum = max(array[k], sum + array[k]);
        best = max(best, sum);
    }
    cout << best << "\n";
}
```

The algorithm only contains one loop that goes through the input, so the time complexity is $O(n)$. This is also the best possible time complexity, because any algorithm for the problem has to examine all array elements at least once.

The above algorithm is famously known as *Kadane's Algorithm.*