

Chapter One

Chapter one: Introduction to Data Structures and Algorithm

1. Overview

This chapter gives an overview of data structure and the theory of the analysis of the complexity of algorithms. First, definition of data structures and algorithms, the terms computational complexity and asymptotic complexity are introduced. Next, the common notations for specifying asymptotic complexity are described. Some common classes of algorithm complexity are listed, and examples of how to classify algorithms into these complexity classes are given. The best case, worst case and average case efficiencies are introduced with examples. Finally the topic of amortized complexity is described.

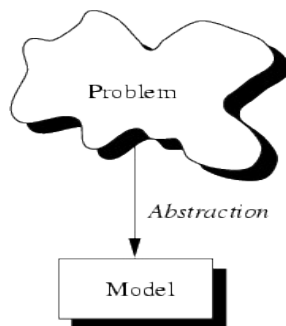
A program is written in order to solve a problem. A solution to a problem actually consists of two things:

- A way to organize the data
- Sequence of steps to solve the problem

The way data are organized in a computer's memory is said to be Data Structure and the sequence of computational steps to solve a problem is said to be an algorithm. Therefore, program is a combination of data structures plus algorithms.

1.1. Introduction to Data Structures

Given a problem, the first step to solve the problem is obtaining one's own abstract view, or model, of the problem. This process of modeling is called abstraction.



The model defines an abstract view to the problem. This implies that the model focuses only on problem related stuff and that a programmer tries to define the **properties** of the problem.

These properties include:

- The **data** which are affected and
- The **operations** that are involved in the problem.

With abstraction you create a well-defined entity that can be properly handled. These entities define the **data structure** of the program.

An entity with the properties just described is called an **abstract data type** (ADT).

1.1.1. Abstract Data Types (ADT)

An ADT consists of an **abstract data structure** and **operations**. Put in other terms, an ADT is an abstraction of a data structure.

The ADT specifies:

1. What can be stored in the Abstract Data Type?
2. What operations can be done on/by the Abstract Data Type?

For example, if we are going to model employees of an organization:

- This ADT stores employees with their relevant attributes and discarding irrelevant attributes.
- This ADT supports hiring, firing, retiring, operations.

A data structure is a language construct that the programmer has defined in order to implement an abstract data type.

There are lots of formalized and standard Abstract data types such as Stacks, Queues, Trees, etc.

Do all characteristics need to be modeled? Not at all

- It depends on the scope of the model
- It depends on the reason for developing the model

1.1.2. Abstraction

Abstraction is a process of classifying characteristics as relevant and irrelevant for the particular purpose at hand and ignoring the irrelevant ones.

Applying abstraction correctly is the essence of successful programming

How do data structures model the world or some part of the world?

- The value held by a data structure represents some specific characteristic of the world
- The characteristic being modeled restricts the possible values held by a data structure
- The characteristic being modeled restricts the possible operations to be performed on the data structure.

Note: Notice the relation between characteristic, value, and data structures

1.2. Algorithms

An algorithm is a well-defined computational procedure that takes some value or a set of values as input and produces some value or a set of values as output. Data structures model the static part of the world. They are unchanging while the world is changing. In order to model the dynamic part of the world we need to work with algorithms. Algorithms are the dynamic part of a program's world model.

An algorithm transforms data structures from one state to another state in two ways:

- An algorithm may change the value held by a data structure
- An algorithm may change the data structure itself

The quality of a data structure is related to its ability to successfully model the characteristics of the world. Similarly, the quality of an algorithm is related to its ability to successfully simulate the changes in the world.

However, independent of any particular world model, the quality of data structure and algorithms is determined by their ability to work together well. Generally speaking, correct data structures lead to simple and efficient algorithms and correct algorithms lead to accurate and efficient data structures.

1.2.1. Properties of an algorithm

- **Finiteness:** Algorithm must complete after a finite number of steps.
- **Definiteness:** Each step must be clearly defined, having one and only one interpretation. At each point in computation, one should be able to tell exactly what happens next.

- **Sequence:** Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted.
- **Feasibility:** It must be possible to perform each instruction.
- **Correctness:** It must compute correct answer all possible legal inputs.
- **Language Independence:** It must not depend on any one programming language.
- **Completeness:** It must solve the problem completely.
- **Effectiveness:** It must be possible to perform each step exactly and in a finite amount of time.
- **Efficiency:** It must solve with the least amount of computational resources such as time and space.
- **Generality:** Algorithm should be valid on all possible inputs.
- **Input/output:** There must be a specified number of input values, and one or more result values.

2. Computational and Asymptotic Complexity

The field of complexity analysis is concerned with the study of the efficiency of algorithms, therefore the first question we must ask ourselves is: what is an algorithm? An algorithm can be thought of as a set of instructions that specifies how to solve a particular problem. For any given problem, there are usually a large number of different algorithms that can be used to solve the problem. All may produce the same result, but their efficiency may vary. In other words, if we write programs (e.g. in C++) that implement each of these algorithms and run them on the same set of input data, then these implementations will have different characteristics. Some will execute faster than others; some will use more memory than others. These differences may not be noticeable for small amounts of data, but as the size of the input data becomes large, so the differences will become significant.

To compare the efficiency of algorithms, a measure of the degree of difficulty of an algorithm called *computational complexity* was developed in the 1960's by Juris Hartmanis and Richard E. Stearns. Computational complexity indicates how much effort is needed to execute an algorithm, or

what its cost is. This cost can be expressed in terms of execution time (*time efficiency*, the most common factor) or memory (*space efficiency*).

- **Time Complexity:** Determine the approximate number of operations required to solve a problem of size n .
- **Space Complexity:** Determine the approximate memory required to solve a problem of size n .

Since time efficiency is the most important, we will focus on this for the moment. When we run a program on a computer, what factors influence how fast the program runs?

1. One factor is obviously the efficiency of the algorithm,
2. The speed of the computer the program is run on is also a factor.
3. The amount of input data is another factor: it will normally take longer for a program to process 10 million pieces of data than 1.
4. Another factor is the language in which the program is written. Compiled languages are generally much faster than interpreted languages, so a program written in C/C++ may execute up to 20 times faster than the same program written in BASIC.

It should be clear that we couldn't use real-time units such as microseconds to evaluate an algorithm's efficiency. A better measure is to use the number of *operations* required to perform an algorithm, since this is independent of the computer that the program is run on.

(Here, an operation can mean a single program statement such as an assignment statement.) Or a step in an algorithm.

Even this measure has problems, since high-level programming language statements do more than low-level programming language statements, but it will do for now.

We need to express the relationship between the size n of the input data and the number of operations t required to process the data. For example, if there is a linear relationship between the size n and the number of operations t (that is, $t = c.n$ where c is a constant), then an increase in the size of the data by a factor of 5 results in an increase in number of operations by factor of 5.

Similarly, if $t = \log_2 n$ then a doubling of n causes t to increase by 1.

In other words, in complexity analysis we are not interested in how many microseconds it will take for an algorithm to execute. We are not even that interested in how many operations it will take. The important thing is how fast the number of operations grows as the size of the data grows.

The examples given in the preceding paragraph are simple. In most real-world examples the function expressing the relationship between n and t

would be much more complex. Luckily it is not normally necessary to determine the precise function, as many of the terms will not be significant when the amount of data becomes large. For example, consider the function $t = f(n) = n^2 + 5n$. This function consists of two terms, n^2 and $5n$. However, for any n larger than 5 the n^2 term is the most significant, and for very large n we can effectively ignore the $5n$ term. Therefore we can *approximate* the complexity function as $f(n) = n^2$. This simplified measure of efficiency is called *asymptotic complexity* and is used when it is difficult or unnecessary to determine the precise computational complexity function of an algorithm. In fact it is normally the case that determining the precise complexity function is not feasible, so the asymptotic complexity is the most common complexity measure used.

Definition 1: The *computational complexity* of an algorithm is a measure of the cost (usually in execution time) incurred by applying the algorithm.

Definition 2: The *asymptotic complexity* of an algorithm is an approximation of the computational complexity that holds for large amounts of input data.

2.1. Big-O Notation

The most commonly used notation for specifying asymptotic complexity, that is, for estimating the rate of growth of complexity functions, is known as *big-O notation*. Big-O notation was actually introduced before the invention of computers (in 1894 by Paul Bachman) to describe the rate of function growth in mathematics. It can also be applied in the field of complexity analysis as we are dealing with functions that relate then number of operations t and the size of the data n .

Given two positive-valued functions f and g , consider the following definition:

Definition 3: The function $f(n)$ is $O(g(n))$ if there exist positive numbers c and k such that $f(n) \leq c \cdot g(n)$ for all $n \geq k$.

This definition states that $g(n)$ is an *upper bound* on the value of $f(n)$. In other words, in the long run (for large n) f grows *at most* as fast as g .

To illustrate this definition, consider the previous example where $f(n) = n^2 + 5n$. We showed in the last section that for large values of n we could approximate this function by the n^2 term only; that is, the asymptotic complexity of $f(n)$ is n^2 . Therefore, we can say now that $f(n)$ is $O(n^2)$. In

the definition, we substitute n^2 for $g(n)$, and we see that it is true that $f(n) \leq 2.g(n)$ for all $n \geq 5$ (i.e. in this case $c=2$, $k=5$).

The problem with definition 3 is that it does not tell us how to calculate c and N . In actual fact, there are usually an infinite number of pairs of values for c and N . We can show this by solving the inequality from definition 3 and substituting the appropriate terms, i.e.

$$\begin{aligned} f(n) &\leq c.g(n) \\ n^2 + 5n &\leq c. n^2 \\ 1 + (5/n) &\leq c \end{aligned}$$

Therefore, if we choose $k=5$, then $c=2$; if we choose $k=6$, then $c=1.83$, and so on. So what are the 'correct' values for c and k ? The answer to this question, it should be determined for which value of k a particular term in $f(n)$ becomes the largest and stays the largest. In the above example, the n^2 term becomes larger than the $5n$ term at $n>5$, so $k=5$, $c=2$ is a good choice.

Another problem with definition 3 is that there are actually infinitely many functions $g(n)$ that satisfy the definition. For example, we chose n^2 , but we could also have chosen n^3 , n^4 , n^5 , and so on. All of these functions satisfy definition 3. To avoid this problem, the smallest function g is chosen, which in this case is n^2 .

Properties of Big-O Notation

There are a number of useful properties of big-O notation that can be used when estimating the efficiency of algorithms:

Fact 1: If $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$ then $f(n) + g(n)$ is $O(h(n))$.

In terms of algorithm efficiency, this fact states that if your program consists of, for example, one $O(n^2)$ operation followed by another independent $O(n^2)$, then the final program will also be $O(n^2)$.

Fact 2: The function $a.n^k$ is $O(n^k)$ for any a and k .

In other words, multiplying a complexity function by a constant value (a) does not change the asymptotic complexity.

Fact 3: The function $\log_a n$ is $O(\log_b n)$ for any positive numbers a and $b \neq 1$

This states that in the context of big-O notation it does not matter what the base of the logarithmic function is - all logarithmic functions have the

same rate of growth. So if a program is $O(\log_2 n)$ it is also $O(\log_{10} n)$. Therefore from now on we will leave out the base and just write $O(\log n)$.

2.2. Ω , Θ and Little-o Notations

There exist three other, less common, ways of specifying the asymptotic complexity of algorithms. We have seen that big-O notation refers to an *upper bound* on the rate of growth of a function, where this function can refer to the number of operations required to execute an algorithm given the size of the input data. There is a similar definition for the *lower bound*, called *big-omega* (Ω) notation.

Definition 4: The function $f(n)$ is $\Omega(g(n))$ if there exist positive numbers c and N such that $f(n) \geq c \cdot g(n)$ for all $n \geq N$.

This definition is the same as definition 3 apart from the direction of the inequality (i.e. it uses $\geq$ instead of $\leq$). We can say that $g(n)$ is a lower bound on the value of $f(n)$, or, in the long run (for large n) f grows at *least* as fast as g .

Ω notation has the same problems as big-O notation: there are many potential pairs of values for c and N , and there are infinitely many functions that satisfy the definition. When choosing one of these functions, for Ω notation we should choose the largest function. In other words, we choose the smallest upper bound (big-O) function and the largest lower bound (Ω) function. Using the example we gave earlier, to test if $f(n) = n^2 + 5n$ is $\Omega(n^2)$ we need to find a value for c such that $n^2 + 5n \geq c \cdot n^2$. For $c=2$ this expression holds for all $n \geq 5$.

For some algorithms (but not all), the lower and upper bounds on the rate of growth will be the same. In this case, a third notation exists for specifying asymptotic complexity, called *theta* (Θ) notation.

Definition 5: The function $f(n)$ is $\Theta(g(n))$ if there exist positive numbers c_1 , c_2 and N such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq N$.

This definition states that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$. In other words, the lower and upper bounds on the rate of growth are the same.

For the same example, $f(n) = n^2 + 5n$, we can see that $g(n) = n^2$ satisfies definition 5, so the function $n^2 + 5n$ **is** $\Theta(n^2)$. Actually we have shown this already by showing that $g(n) = n^2$ satisfies both definitions 3 and 4.

The final notation is *little-o notation*. You can think of little-o notation as the opposite of Θ notation.

Definition 6: The function $f(n)$ is $o(g(n))$ if $f(n)$ is $O(g(n))$ but $f(n)$ is not $\Theta(g(n))$.

In other words, if a function $f(n)$ is $O(g(n))$ but not $\Theta(g(n))$, we denote this fact by writing that it is $o(g(n))$. This means that $f(n)$ has an upper bound of $g(n)$ but a different lower bound, i.e. it is not $\Omega(g(n))$.

2.3. OO Notation

The four notations described above serve the purpose of comparing the efficiency of various algorithms designed for solving the same problem. However, if we stick to the strict definition of big-O as given in definition 3, there is a possible problem. Suppose that there are two potential algorithms to solve a certain problem, and that the number of operations required by these algorithms is 10^8n and $10n^2$, where n is the size of the input data. The first algorithm is $O(n)$ and the second is $O(n^2)$. Therefore, if we were just using big-O notation we would reject the second algorithm as being too inefficient. However, upon closer inspection we see that for all $n < 10^7$ the second algorithm requires fewer operations than the first. So really when deciding between these two algorithms we need to take into consideration the expected size of the input data n .

For this reason, in 1989 UdiManber proposed one further notation: OO notation:

Definition 7: The function $f(n)$ is $OO(g(n))$ if it is $O(g(n))$ but the constant c is too large to be of practical significance.

Obviously in this definition we need to define exactly what we mean by the term “practical significance”. In reality, the meaning of this will depend on the application.

2.4. Complexity Classes

We have seen now that algorithms can be classified using the big-O, Ω and Θ notations according to their time or space complexities. A number

of complexity classes of algorithms exist, and some of the more common ones are illustrated in Figure 1.

Table 1 gives some sample values for these different complexity classes. We can see from this table how great is the variation in the number of operations when the data becomes large. As an illustration, if these algorithms were to be run on a computer that can perform 1 billion operations per second (i.e. 1 GHz), the quadratic algorithm would take 16 minutes and 40 seconds to process 1 million data items, whereas the cubic algorithm would take over 31 years to perform the same processing. The time taken by the exponential algorithm would probably exceed the lifetime of the universe!

It is obvious that choosing the right algorithm is of crucial importance, especially when dealing with large amounts of data.

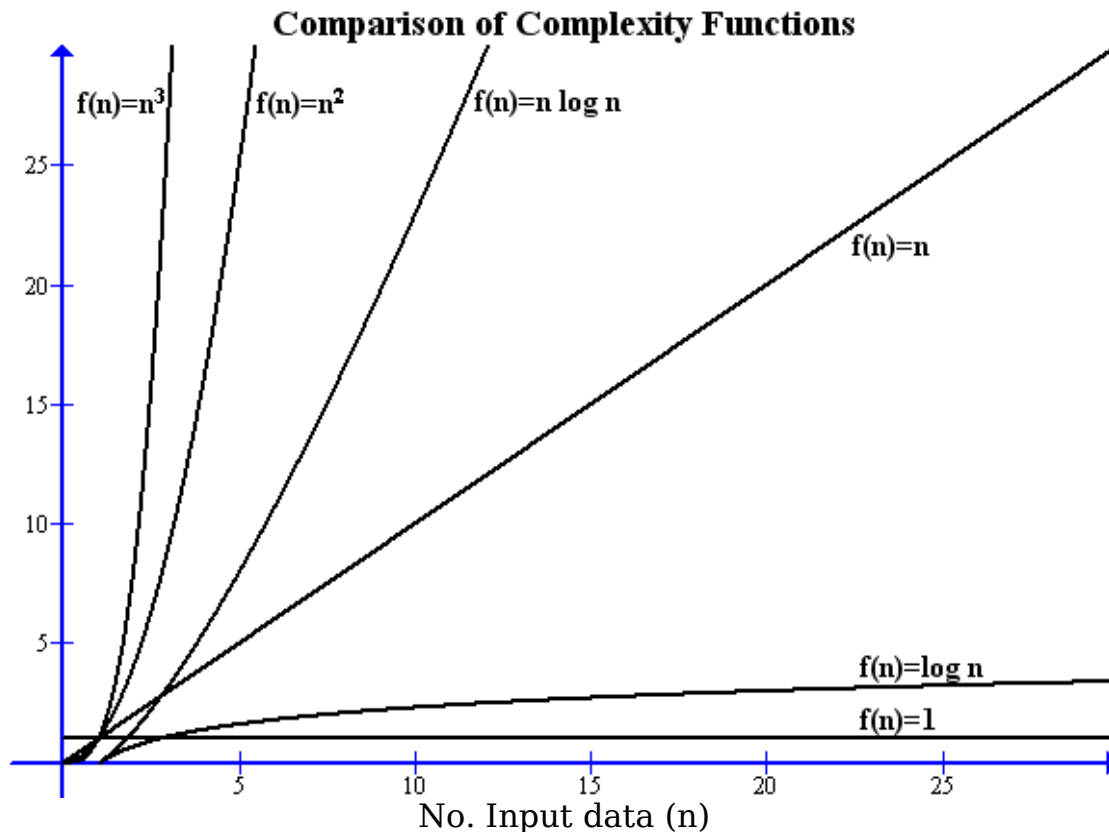


Figure 1 - A comparison of various complexity classes

Complexity Class		Number of operations performed based on size of input data n					
Name	Big-O	n=1 0	n=10 0	n=10 00	n=10,0 00	n=100,0 00	n=1,000, 000
Constant	O(1)	1	1	1	1	1	1
Logarith	O(log n)	3.32	6.64	9.97	13.3	16.6	19.93

mic							
Linear	$O(n)$	10	100	1000	10,000	100,000	1,000,000
$n \log n$	$O(n \log n)$	33.2	664	9970	133,000	$1.66 * 10^6$	$1.99 * 10^7$
Quadratic	$O(n^2)$	100	10,000	10^6	10^8	10^{10}	10^{12}
Cubic	$O(n^3)$	1000	10^6	10^9	10^{12}	10^{15}	10^{18}
Exponential	$O(2^n)$	1024	10^{30}	10^{301}	10^{3010}	10^{30103}	10^{301030}

(Note: the values for the logarithmic complexity class were calculated using base 2 logarithms)

Table 1 - The number of operations required by algorithms of various complexity classes

2.5. Finding Asymptotic Complexity: Examples

Recall that asymptotic complexity indicates the expected efficiency, with regard to time or space, of algorithms when there is a large amount of input data. In most cases we are interested in time complexity. The examples in this section show how we can go about determining this complexity.

Given the variation in speed of computers, it makes more sense to talk about the number of operations required to perform a task rather than the execution time. In these examples, to keep things simple, we will measure the number of assignment statements and ignore comparison and other operations.

Consider the following C++ code fragment to calculate the sum of numbers in an array:

```
for (i = sum = 0; i < n; i++)
    sum += a[i];
```

First, two variables (i and sum) are initialised. Next, the loop iterates n times, with each iteration involving two assignment statements: one to add the current array element $a[i]$ to sum , and one to increment the loop control variable i . Therefore the function that determines the total number of assignment operations t is:

$$t = f(n) = 2 + 2n$$

Since the second term is the largest for all $n > 1$, and the first term is insignificant for very large n , the asymptotic complexity of this code is $O(n)$.

As a second example, the following program outputs the sums of all subarrays that begin with position 0:

```

for (i = 0; i < n; i++) {
    sum = a[0];
    for (j = 1; j <= i; j++)
        sum += a[j];
    cout << "sum for subarray 0 to " << i
    << " is " << sum << endl;
}

```

Here we have a nested loop. Before any of the loop starts, i is initialized. The outer loop is executed n times, with each iteration executing an inner for loop, a print statement, and three assignment statements (to assign $a[0]$ to sum , to initialize j to 1, and to increment i). The inner loop is executed i times for each i in $\{0, 1, 2, \dots, n-1\}$ and each iteration of the inner loop contains two assignments (one for sum and one for j). Therefore, since $0 + 1 + 2 + \dots + n-1 = n(n-1)/2$, the total number of assignment operations required by this algorithm is

$$t = f(n) = 1 + 3n + 2 \cdot n(n-1)/2 = 1 + 2n + n^2$$

Since the n^2 term is the largest for all $n > 2$, and the other two terms are insignificant for large n , this algorithm is $O(n^2)$. In this case, the presence of a nested loop changed the complexity from $O(n)$ to $O(n^2)$. This is often, but not always the case. If the number of iterations of the inner loop is constant, and not independent of the state of the outer loop, the complexity will remain at $O(n)$.

Analysis of the above two examples is relatively uncomplicated because the number of operations required did not depend on the data in the arrays at all. Computation of asymptotic complexity is more involved if the number of operations is dependent on the data.

Consider the following C++ function to perform a *binary search* for a particular number val in an ordered array arr :

```

Int binarySearch (const int arr[], int size,
const int &val)
{
    int lo = 0, mid, hi = size - 1;
    while (lo <= hi) {
        mid = (lo + hi) / 2;
        if (val < arr[mid])
            hi = mid - 1; //try left half of arr
        else if (arr[mid] < val)
            lo = mid + 1; //try right half of arr
        else
            return mid; //success: return index
    }
}

```

```
return -1; //failure: val is not in arr
}
```

The algorithm works by first checking the middle number (at index `mid`). If the required number `val` is there, the algorithm returns its position. If not, the algorithm continues. In the second trial, only half of the original array is considered: the left half if `val` is smaller than the middle element, and the right half otherwise. The middle element of the chosen subarray is checked. If the required number is there, the algorithm returns its position. Otherwise the array is divided into two halves again, and if `val` is smaller than the middle element the algorithm proceeds with the left half; otherwise it proceeds with the right half. This process of comparing and halving continues until either the value is found or the array can no longer be divided into two (i.e. the array consist of a single element).

If `val` is located in the middle element of the array, the loop executes only one time. How many times does the loop execute if `val` is not in the array at all? First, the algorithm looks at the entire array of size n , then at one of its halves of size $n/2$, then at one of the halves of this half of size $n/4$, and so on until the array is of size 1. Hence we have the sequence $n, n/2, n/2^2, \dots, n/2^m$, and we want to know the value of m (i.e. how many times does the loop execute?). We know that the last term $n/2^m$ is equal to 1, from which it follows that $m = \log n$. Therefore the maximum number of times the loop will execute is $\log n$, so this algorithm is $O(\log n)$.

2.6. The Best, Average and Worst Cases

This last example indicates the need for distinguishing a number of different cases when determining the efficiency of algorithms. The *worst case* is the maximum number of operations that an algorithm can ever require, the *best case* is the minimum number, and the *average case* comes somewhere in between these two extremes.

Finding the best and worst case complexities is normally relatively straightforward. In simple cases, the average case complexity is established by considering the possible inputs to an algorithm, determining the number of operations performed by the algorithm for each of the inputs, adding the number of operations for all inputs and dividing by the number of inputs. For example, consider the task of sequentially searching an unordered array for a particular value. If the array is of length n , then the best case is when the number is found in the first element (1 loop executed). The worst case is when it is found in the last element or not found at all (n loops executed). In the average

case the number of loops executed is $(1 + 2 + \dots + n) / n$, which is equal to $(n + 1) / 2$. Therefore, according to Fact 2, the average case is $O(n)$.

The above analysis assumes that all inputs are equally probable. That is, that we are just as likely to find the number in any of the elements of the array. This is not always the case. To explicitly consider the probability of different inputs occurring, the average complexity is defined as the average over the number of operations for each input, weighted by the probability for this input,

$$C_{avg} = \sum_i p(input_i).operations(input_i)$$

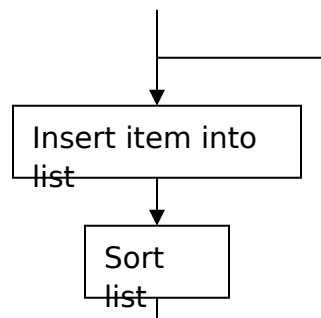
where $p(input_i)$ is the probability of input i occurring, and $operations(input_i)$ is the number of operations required by the algorithm to process input i .

In the binary search example, the best case is that the loop will execute 1 time only. In the worst case it will execute $\log n$ times. But finding the average case for this example, although possible, is not trivial. It is often the case that finding the average case complexity is difficult for real-world examples. For this reason, approximations are used, and this is where the big-O, Ω and Θ notations useful.

2.7. Amortized Complexity - Complexity of Sequence of Algorithms

In many situations, data structures are subject to a sequence of algorithms rather than a single algorithm. In this sequence, one algorithm may make some modifications to the data that have an impact on the run-time of later algorithms in the sequence. How do we determine the complexity of such sequences of algorithms?

One way is to simply sum the worst case efficiencies for each algorithm. But this may result in an excessively large and unrealistic upper bound on run-time. Consider the example of inserting items into a sorted list. In this case, after each item is inserted into the list we need to re-sort the list to maintain its ordering. So we have the following the sequence of algorithms:



In this case, if we have only inserted a single item into the list since the last time it was sorted, then resorting the list should be much faster than sorting a randomly ordered list because it is almost sorted already.

Amortized complexity analysis is concerned with assessing the complexity of such sequences of *related* operations. In this case the operations are related because they operate on the same list data structure and they both change the values in this data structure. If the operations are not related then Fact 1 specifies how to combine the complexities of the two algorithms.

To illustrate the idea of amortized complexity, consider the operation of adding a new element to a list. The list is implemented as a fixed length array, so occasionally the array will become filled up. In this case, a new array will be allocated, and all of the old array elements copied into the new array. To begin with, the array is of length 1. When this becomes full up, an array of length 2 will be allocated; when this becomes full an array of length 4 will be allocated, and so on. In other words, each time the array becomes full, a new array with double the length of the old one will be allocated. The cost in operations of adding an element to the array is 1 if there is space in the array. If there is no space, the cost is equal to the number of elements in the old array (that have to be copied to the new array) plus 1 to add the new element. Table 2 lists the costs in operations for adding each subsequent element. For the first element ($N=1$) the cost is just 1 for inserting the new element. For the second element, the array (currently of length 1) is full up, so it has to be copied to a new array (cost=1) and the new element added (cost=1). For the third element the array (now of length 2) is also full up, so the 2 values in the old array have to be copied to the new array (cost=2) and the new element added (cost=1). For the fourth element there is space in the array as it is now of length 4, so the cost to add the new element is just 1.

We can see from Table 2 that for most iterations the cost of adding a new element is 1, but occasionally there will be a much higher cost, which will raise the average cost for all iterations.

In amortized analysis we don't look at the best or worst case efficiency, but instead we are interested in the expected efficiency of a sequence of operations. If we add up all of the costs in Table 2 we get 51, so the overall average (up to 20 iterations) is 2.55. Therefore if we specify the amortized cost as 3 (to be on the safe side), we can rewrite Table 2 as follows.

N	Cost		N	Cost
1	1		11	1
2	1+1		12	1

3	2+1		13	1
4	1		14	1
5	4+1		15	1
6	1		16	1
7	1		17	16+1
8	1		18	1
9	8+1		19	1
10	1		20	1

Table 2 - The cost of adding elements to a fixed-length array

N	Cost	Amortized cost	Units left		N	Cost	Amortized cost	Units left
1	1	3	2		11	1	3	7
2	1+1	3	3		12	1	3	9
3	2+1	3	3		13	1	3	11
4	1	3	5		14	1	3	13
5	4+1	3	3		15	1	3	15
6	1	3	5		16	1	3	17
7	1	3	7		17	16+1	3	3
8	1	3	9		18	1	3	5
9	8+1	3	3		19	1	3	7
10	1	3	5		20	1	3	9

Table 3 - The amortized cost of adding elements to a fixed-length array

This time we have assigned an *amortized cost* of 3 at each iteration. If at any stage the actual cost is less than the amortized cost we can store this 'saving' in the *units left* column. You can think of this column as a kind of bank account: when we have spare operations we can deposit them there, but later on we may need to make a withdrawal. For example, at the first iteration the actual cost is 1, so we have 2 'spare' operations that we deposit in the units left column. At the second iteration the actual cost is 2, so we have 1 'spare' operation, which we deposit in the units left. At the third iteration the actual cost is 3, so we have no spare operations. At the fourth iteration the actual cost is 1, so we deposit 2 'spare' operations. At the fifth iteration the actual cost is 5, compared with the amortized cost of 3, so we need to withdraw 2 operations from the units left column to make up the shortfall. This process continues, and so long as our 'stored' operations do not become negative then everything is OK, and the amortized cost is sufficient.

This is a simple example chosen to illustrate the concepts involved in amortized complexity analysis. In this case the choice of a constant function for the amortized cost is adequate, but often it is not, and amortized complexity analysis can become more challenging.

Summary of Key Points

The following points summarize the key concepts in this handout:

- Data structure is a way of organizing data in a computer
- Algorithm is a finite set of computational steps to solve a problem
- The *computational complexity* of an algorithm is a measure of the cost incurred by applying the algorithm.
- The cost can be in terms of memory (space efficiency), but is more commonly in terms of execution time (time efficiency).
- A complexity function expresses the number of operations required to apply an algorithm t in terms of the size of the input data n .
- The *asymptotic complexity* is an approximation of the true computational complexity that holds for large amounts of input data.
- *Big-O notation* is the most common way to specify asymptotic complexity.
- Using big-O notation, the function $f(n)$ is $O(g(n))$ if there exist positive numbers c and N such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$.
- Whereas big-O notation specifies an upper bound on the rate of growth of a function, *big-omega (Ω) notation* specifies a lower bound.
- Using Ω notation, the function $f(n)$ is $\Omega(g(n))$ if there exist positive numbers c and N such that $f(n) \geq c \cdot g(n)$ for all $n \geq N$.
- *Theta (Θ) notation* can be used when the lower and upper bounds on the rate of growth of a function are the same.
- Using Θ notation, the function $f(n)$ is $\Theta(g(n))$ if there exist positive numbers c_1 , c_2 and N such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq N$.
- *Little-o notation* is used when the lower and upper bounds on the rate of growth of a function are not the same.
- Using little-o notation, the function $f(n)$ is $o(g(n))$ if $f(n)$ is $O(g(n))$ but it is not $\Theta(g(n))$.
- The function $f(n)$ is $OO(g(n))$ if it is $O(g(n))$ but the constant c is too large to be of practical significance.

- Common complexity classes are *constant*, *logarithmic*, *linear*, $n \log n$, *quadratic*, *cubic* and *exponential*.
- Sometimes the number of operations required to apply an algorithm will depend on the input data.
- The *worst case* is the maximum number of operations that an algorithm can ever require to execute.
- The *best case* is the minimum number of operations that an algorithm can ever require to execute.
- The *average case* is the average number of operations that an algorithm requires to execute.
- *Amortized complexity* analysis is concerned with assessing the efficiency of sequences of related operations.