Chapter One

Analysis of Algorithms

Introduction to Data Structures and Algorithms Analysis

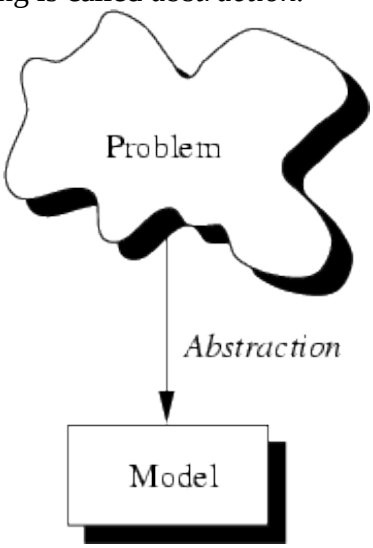
A program is written in order to solve a problem. A solution to a problem actually consists of two things:

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

The way data are organized in a computers memory is said to be Data Structure and the sequence of computational steps to solve a problem is said to be an algorithm. Therefore, a program is nothing but data structures plus algorithms.

1.1. Introduction to Data Structures

Given a problem, the first step to solve the problem is obtaining ones 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

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

Where are algorithms, then?

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.

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 for 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.

Chapter Objectives: At the end of this chapter you should be able to:

- Describe how to analyse a program.
- Explain how to choose the operations that are counted and why others are not.
- Explain how to do a best-case, worst-case and average-case analysis.
- Work with logarithms, probabilities and summations.
- Describe $\Theta(f)$, $\Omega(f)$, $O(f)$, growth rate, and algorithm order.
- Use decision tree to determine a lower bound on complexity
- Convert a simple recurrence relation into its closed form.

1.1 Performance Analysis

By performance of a program, we mean the amount of computer memory and time needed to run a program. There are two approaches to determine the performance of a program:

- Analytical approach- used for performance analysis,
- Experimental approach – conduct experiments for performance measurement.

What is Analysis: - The analysis of algorithm provides background information that a general idea of how long an algorithm will take for a given set. For each algorithm considered, we will come up with an estimate of how long it will take to solve a problem that has a set of N input values. For example:-

i. We might determine how many comparisons a sorting algorithm does to put a list of N values in to ascending order, or,

ii. We might determine how many arithmetic operations it takes to multiply two matrices of size $N \times N$

There a number of algorithms that will solve a problem. Studying the analysis of algorithms gives us the tools to choose between algorithms.

Input Classes

Two important measures of an algorithms performance are its time and space complexity measured as functions of the size of the input. Input plays an important role in analysing algorithms because it is the input that determines what the path of execution though an algorithm will be. For example, if we are interested to find out the largest value in a list of N numbers, we can use the following algorithm:

```
largest = list[1]
for i=2 to N do
        if(list[i] > largest) then
                largest = list[i]
        end if
end for
```

We can see that, if the list is in decreasing order there will only one assignment done before the loop starts. If the list is in increasing order, however, there will be N assignments (one before the loop starts and N-1 inside the loop).

So, our analysis must consider more than one possible set of input, because if we look at one set of input, it may be the set that is solved fastest (slowest). This will give us false impression of the algorithm. Instead we consider all types of input set.

When we look at the input we will try to break up all the different input set into classes based on how the algorithm behaves on each set. This helps to reduce the number of possibilities that we will need to consider.

1.2. Space and Time Complexity

The complexity of an algorithm can be expressed in terms of its space and time complexity.

Space Complexity of a program: - is the amount of memory space it needs to run to completion. We are interested in the space complexity of a program for the following reasons:

- If a program is to run on a multiuser computer system, then we may need to specify the amount of memory to be allocated for the program.
- For any computer system, we would like to know in advance whether or not sufficient memory is available to run the program.
- A problem might have several possible solutions with different space requirements.
- We can use the space complexity to estimate the size of the largest problem that a program can solve. For example, we may have a circuit simulation program that requires $10^6 + 100(c + w)$ bytes of memory to simulate circuits with c components and w wires. If the total amount of memory available is $4 * 10^6$, then we can simulate circuits with $c + w <= 30,000$.

Components of space complexity: - the space needed by a program has the following components:

- **Instruction space**: - is the amount of space needed to store the compiled version of the program instructions. The amount of instruction space that is needed depends on factors such as:
    - The compiler used to compile the program into machine code.
    - The compiler options in effect at time of compilation.
    - The target computer.
- **Data space**: - is the space needed to store all constant and variable values. Data space has two components:
    1. Space needed by constants (e.g. the numbers 0, 1, 2...) and simple variables.

2. Space needed by dynamically allocated objects such as arrays and class instances.

- **Environmental stack space**:- is the space used to save information needed to resume execution of partially completed methods (functions). For example, if method ***func1*** invokes or calls method ***func2***, then we must at least save a pointer to the instruction of ***func1*** to be executed when ***func2*** terminates. Each time a function or method is called or invoked, the following data are stored on the environment stack.
    1. The return address
    2. The values of all local variables and formal parameters in the method or function being invoked or called.

Time Complexity of a program: - is the amount of computer time it needs to run to completion. We are interested in time complexity of program for the following reasons:

- Some computer systems require the user to provide an upper limit on the amount of time the program will run. Once this upper limit is reached, the program is aborted. An easy way out is to simply specify a time limit of a few thousand years. However, this solution could result in serious fiscal problems if the program runs into an infinite loop caused by some discrepancy in the data and you actually get billed for the computer time used. We would like to provide a time limit that is just slightly above the expected run time.
- The program we are developing might need to provide a satisfactory real-time response. For example, all interactive programs must provide such a response. A text editor that takes a minute to move the cursor one page down or one page up will not be acceptable to many users. A spreadsheet program that takes several minutes to re-evaluate the cells in a sheet will be satisfactory only to patient users. A database management system that allows its users adequate time to drink two cups of coffee while it is sorting a relation will not find too much acceptance. Programs designed for interactive use must provide satisfactory real-time response. From the time complexity of the program or program module, we can decide whether or not the response time will be acceptable. If not, we need to either redesign the algorithm or give the user a faster computer.

If we have alternative ways to solve a problem, then the decision on which to use will be based primarily on the expected performance difference among those solutions. We will use some weighted measures of the space and time complexities of the alternative solutions.

Components of time complexity: - the time complexity of a program depends on all the factors that the space complexity depends on.

Since analytical approach to determine the run time of a program is fraud with complications, we attempt only to estimate the run time. To more manageable approach to estimating the run time are:

1. Identify one or more key operations and determine the number of times these are performed, and
2. Determine the total number of steps executed by the program.

1.3 What to count and consider

Deciding what to count involves two steps. The first is choosing the significant operation or operations, and the second is deciding which of these operations are integral to the algorithm and which are overhead or bookkeeping. There are two classes of operations that are typically chosen for the significant operations. These significant classes of operations are ***comparison operations*** and ***arithmetic operations.***

- The comparison operators are all considered equivalent and we count them in algorithms such as searching and sorting. In these algorithms, the important task being done is the comparison of two values to determine, when searching, if the value is the one we are looking for or, when sorting, if the values

are out of order. Comparison operators include ***equal***, ***not equal, less than, greater than, less than or equal, and greater than or equal.***

- We will count arithmetic operators in two groups: ***Additive*** and ***Multiplicative***. Additive operators (usually called additions for short) include ***addition, subtraction, increment,*** and ***decrement***. Multiplicative operators (usually called multiplications for short) include ***multiplication, division, and modulus***.

These two groups are counted separately because multiplications are considered to take longer than additions. In case of integer multiplication or division by a power of 2, this operation can be reduced to a shift operation, which is considered as fast as addition.

Cases to consider

Choosing what input to consider when analysing and algorithm can have a significant impact on how an algorithm will perform. If the input is already sorted, the some sorting algorithms will perform very well, but other sorting algorithms may perform very poorly. The opposite may be true if the list is randomly arranged instead of sorted. Because of this, we will not consider just one input set when we analyse an algorithm. In fact, we will actually look for those input sets that allow an algorithm to perform the most quickly and the most slowly. We will also consider an overall average performance of the algorithm as well.

Best Case

As its name indicates, the best case for an algorithm is the input that requires the algorithm to take the shortest time. This input is the combination of values that cause the algorithm to do the least amount of work.

For example, if we are looking at a searching algorithm, the best case would be if the value we are searching for (commonly called the target or key) was the value stored in the first location that the searching algorithm will check. This would then require only one comparison no matter how complex the algorithm is. Notice that for searching through a list of values, no matter how large, the best case will result in a constant time of 1. Because the best case for an algorithm will usually be a very small and frequently constant value, we will not do a best-case analysis very frequently.

Worst Case

Worst case is an important analysis because it gives us an idea of the most an algorithm will ever take. Worst-case analysis requires that we identify the input values that cause an algorithm to do the most work.

For example, for searching algorithms, the worst case is one where the value is in the last place we check or is not in the list. This could involve comparing the key to each list value for a total of N comparisons. The worst case gives us an upper bound on how slowly parts of our programs may work based on our algorithm choices.

Average Case

The average-case analysis is the toughest to do because there are a lot of details involved. The basic process begins by determining the number of different groups into which all possible input sets can be divided. The second step is to determine the probability that the input will come from each of these groups. The third step is to determine how long the algorithm will run for each of these groups. All of the input in each group should take the same amount of time, and if they do not, the group must be split into two separate groups. When all of this has been done, the average case time is given by the following formula:

$$A(n) = \sum_{i=1}^{m} \boldsymbol{P}i * \boldsymbol{t}i$$

Where n is the size of the input, m is the number of groups, p_i is the probability that the input will be from group ***i***, and t_i is the time that the algorithm will takes for input from group ***i***.

In some cases, we will consider that each of the input groups has equal probabilities. In other words, if there are five input groups, the chance the input will be in group 1 is the same as the chance for group 2, and so on. This would mean that for these five groups all possibilities would be 0.2. We could calculate the average case by the above formula, or we could note that the following simplified formula is equivalent in the case of where all groups are equally probable:

$$A(n) = \frac{1}{m}\sum_{i=1}^{m} Pi * ti$$

1.4 Mathematical Background

There are some mathematical concepts that will be useful in analysis of algorithms.

Floor and Ceiling of a given number

Floor of a number: we say a floor of a X (written as $\lfloor X \rfloor$), is the largest integer that is less than or equal to X. So, $\lfloor 2.5 \rfloor$ would be 2 and $\lfloor -7.3 \rfloor$ would be -8.

Ceiling of a given number: we say that the ceiling of X (written as $\lceil X \rceil$), is the smallest integer greater than or equal to X. So, $\lceil 2.5 \rceil$ would be 3 and $\lceil -7.3 \rceil$ would be -7.

Because we would be using just positive numbers, we can think of the floor as truncation and the ceiling as rounding up. For negative numbers, the effect is reversed.

The floor and ceiling will be used when we need to determine how many times something is done; the value depends on some fraction of the items it is done to.

Logarithms

Because logarithms play an important role in our analysis, there are some properties that must be discussed.

The logarithm base Y of a number X, is the power of Y that will produce the number X. Logarithms are strictly increasing functions. This means that given two numbers X and Y, if X<Y, $\log_B X < \log_B Y$ for all bases B. Logarithms are one-to-one functions. This means if $\log_B X = \log_B Y$, then X=Y. Other properties that are important for you are:

- $\log_B 1 = 0$
- $\log_B B = 1$
- $\log_B (X*Y) = \log_B X + \log_B Y$
- $\log_B X^Y = Y * \log_B X$
- $\log_A X = \frac{\log_B X}{\log_B A}$

Probabilities

Because we will analyse algorithms relative to their input size, we may at times need to consider the likelihood of a certain set of input. This means that we will need to work with the probability that the

input will meet some condition. The probability that something will occur is the given as a number in the range of 0 to 1, where 0 means it will never occur, and 1 means it will always occur.

If we know that there are exactly 10 different possible inputs, we can say that the probability of each of these is between 0 and 1, and the total of all of the individual probabilities is 1, because one of these must happen. If there is an equal chance that any of these can occur, each will have a probability of 0.1 (one out of 10 or $1/10$).

For most of our analysis, we will first determine how many possible situations there are and then assume that all are equally likely. If we determine that there are N possible situations, these results in a probability of 1/N for each of these situations.

Summations

We will be adding up sets of values as we analyse our algorithms. Let's say we have an algorithm with a loop. We notice that when the loop variable is 5, we do 5 steps and when it is 20, we do 20 steps. We determine in general that when the loop variable is m, we do m steps. Overall, the loop variable will take on all values from 1 to N. So the total steps are the sum of the values from 1 through N. To easily express this, we use $\sum_{i=0}^{n}$. The expression below $\sum$ represents the initial value for the summation variable, and the value above the $\sum$ represents the ending value.

Once we have expressed some solution in terms of this summation notation, we will want to simplify this so that we can make comparisons with other formulas. The following are set of standard summation formulas determine the actual values these summations represent.

- $\sum_{i=1}^{N} \mathrm{c} * \mathrm{i} = c * \sum_{i=1}^{N} \mathrm{i}$, with c a constant expression not dependent on i.
- $\sum_{i=c}^{N} \mathrm{i} = \sum_{i=0}^{N} (\mathrm{c} + \mathrm{i})$
- $\sum_{i=c}^{N} \mathrm{i} = \sum_{i=0}^{N} \mathrm{i} - \sum_{i=0}^{c-1} \mathrm{i}$
- $\sum_{i=1}^{N} (\mathrm{A} + \mathrm{B}) = \sum_{i=1}^{N} \mathrm{A} + \sum_{i=1}^{N} \mathrm{B}$
- $\sum_{i=0}^{N} (\mathrm{N} - \mathrm{i}) = \sum_{i=0}^{N} \mathrm{i}$
- $\sum_{i=1}^{N} 1 = \mathrm{N}$
- $\sum_{i=1}^{N} \mathrm{C} = \mathrm{C} * \mathrm{N}$
- $\sum_{i=1}^{N} \mathrm{i} = \frac{\mathrm{N}(\mathrm{N}+1)}{2}$
- $\sum_{i=1}^{N} i^2 = \frac{\mathrm{N}(\mathrm{N}+1)(2\mathrm{N}+1)}{6} = \frac{2N^3+3N^2+N}{6}$
- $\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$
- $\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$, for some number A
- $\sum_{i=1}^{N} i2^i = (N-1)2^{N+1} + 2$
- $\sum_{i=1}^{n} \frac{1}{i} = \ln N$
- $\sum_{i=1}^{N} \log_2 i \simeq \log_2 N - 1.5$

1.5. Complexity analysis

1.5.1 Rates of Growth

Complexity analysis involves two distinct phases:

Algorithm Analysis: Analysis of the algorithm or data structure to produce a function T (n) that describes the algorithm in terms of the operations performed in order to measure the complexity of the algorithm.

- **Order of Magnitude Analysis**: Analysis of the function T (n) to determine the general complexity category to which it belongs.

There is no generally accepted set of rules for algorithm analysis. However, an exact count of operations is commonly used.

Analysis Rules:

1. We assume an arbitrary time unit.
2. Execution of one of the following operations takes time 1:
   - Assignment Operation
   - Single Input/output Operation
   - Single Boolean Operations
   - Single Arithmetic Operations
   - Function Return
3. Running time of a selection statement (if, switch) is the time for the condition evaluation plus the maximum of the running times for the individual clauses in the selection.
4. **Loops**: Running time for a loop is equal to the running time for the statements inside the loop multiplied by the number of iterations.
   The total running time of a statement inside a group of nested loops is the running time of the statements multiplied by the product of the sizes of all the loops.
   For nested loops, analyze inside out.
   - Always assume that the loop executes the maximum number of iterations possible.
5. Running time of a function call is 1 for setup plus the time for any parameter calculations plus the time required for the execution of the function body.

Examples

Example1:

```
int count()
{        int k=0;
  cout<< "Enter an integer";
cin>>n;
for (i=0;i<n;i++)
  k=k+1;
return 0;
  }
```

Time Units to Compute

---

1 for the assignment statement: int k=0
1 for the output statement.
1 for the input statement.
In the for loop:
1 assignment, *n+1* tests, and *n* increments.
n loops of 2 units for an assignment, and an addition.
1 for the return statement.

Example2:

```
int total(int n)
{
  int sum=0;
  for (int i=1;i<=n;i++)
     sum=sum+1;
  return sum;
}
```

Time Units to Compute

---

1 for the assignment statement: int sum=0
In the for loop:
1 assignment, *n+1* tests, and *n* increments.
n loops of 2 units for an assignment, and an addition.
1 for the return statement.

T (n)= *1+(1+n+1+n)+2n+1 = 4n+4 = O(n)*

Example3:

```
void func()
{
 int x=0;int i=0;int j=1;
 cout<< "Enter an Integer value";
 cin>>n;
 while (i<n){
   x++;
   i++;
 }
 while (j<n){
   j++;
 }
}
```

Time Units to Compute

---

1 for the first assignment statement: x=0;
1 for the second assignment statement: i=0;
1 for the third assignment statement: j=1;
1 for the output statement.
1 for the input statement.
In the first while loop:
n+1 tests
n loops of 2 units for the two increment (addition) operations
In the second while loop:
n tests
n-1 increments

T (n)= *1+1+1+1+1+n+1+2n+n+n-1 = 5n+5 = O(n)*

T (n)= *1+1+1+(1+n+1+n)+2n+1 = 4n+6 = O(n)*

Example4:

```
 int sum (int n)
{
 int p_sum = 0;
 for (int i = 1; i <= n; i++)
        p_sum = p_sum +(i * i * i);
 return partial_sum;
}
```

Time Units to Compute

---

1 for the assignment.

1 assignment, *n+1* tests, and *n* increments.

n loops of 4 units for an assignment, an addition, and two multiplications.

1 for the return statement.

T (n)= *1+(1+n+1+n)+4n+1 = 6n+4 = O(n)*

Formal Approach to Analysis

In the above examples we have seen that analysis is a bit complex. However, it can be simplified by using some formal approach in which case we can ignore initializations, loop control, and book keeping.

for Loops: Formally

- In general, a for loop translates to a summation. The index and bounds of the summation are the same as the index and bounds of the for loop.

```
for (int i = 1; i <= N; i++) {
    sum = sum + i;
}
```

$$\sum_{i=1}^{N} 1 = N$$

- Suppose we count the number of additions that are done. There is 1 addition per iteration of the loop, hence *N* additions in total.

Nested Loops: Formally

- Nested for loops translate into multiple summations, one for each for loop.

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= M; j++) {
        sum = sum + i + j;
    }
}
```

$$\sum_{i=1}^{N}\sum_{j=1}^{M} 2 = \sum_{i=1}^{N} 2M = 2MN$$

- Again, count the number of additions. The outer summation is for the outer for loop.

Consecutive Statements: Formally

- Add the running times of the separate blocks of your code

```
for (int i = 1; i <= N; i++) {
    sum = sum + i;
}
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        sum = sum + i + j;
    }
}
```

$$\left[\sum_{i=1}^{N} 1\right] + \left[\sum_{i=1}^{N}\sum_{j=1}^{N} 2\right] = N + 2N^2$$

Conditionals: Formally

- If (test) s1 else s2: Compute the maximum of the running time for s1 and s2.

```
if (test == 1) {
    for (int i = 1; i <= N; i++) {
        sum = sum + i;
}}
else for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            sum = sum + i + j;
}}
```

$$\max\left(\sum_{i=1}^{N} 1, \sum_{i=1}^{N}\sum_{j=1}^{N} 2\right) = \max\left(N, 2N^2\right) = 2N^2$$

Measures of Times

In order to determine the running time of an algorithm it is possible to define three functions $T_{best}(n)$, $T_{avg}(n)$ and $T_{worst}(n)$ as the best, the average and the worst case running time of the algorithm respectively.

Average Case (T_{avg}): The amount of time the algorithm takes on an "average" set of inputs.
Worst Case (T_{worst}): The amount of time the algorithm takes on the worst possible set of inputs.
Best Case (T_{best}): The amount of time the algorithm takes on the smallest possible set of inputs.
We are interested in the worst-case time, since it provides a bound for all input – this is called the "Big-Oh" estimate.

Asymptotic Analysis

In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the ***rate of growth of the algorithm***. What happens with small sets of input data is not as interesting as what happens when the data set gets large.

The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the ***order*** of the function or related algorithm. We then group algorithms based on their order. We group in three categories – those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster.

The following are some of the classes of algorithms: - $\log_2 n$, n, $n * \log_2 n$, n^2, n^3, 2^n.

Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

There are five notations used to describe a running time function. These are:

- Big-Oh Notation (O)
- Big-Omega Notation (Ω)
- Theta Notation (Θ)
- Little-o Notation (o)
- Little-Omega Notation (ω)

The Big-Oh Notation

Big-Oh notation is a way of comparing algorithms and is used for computing the complexity of algorithms; i.e., the amount of time that it takes for computer program to run. It's only concerned with what happens for very a large value of n. Therefore only the largest term in the expression (function) is needed. For example, if the number of operations in an algorithm is $n^2 – n$, n is insignificant compared to n^2 for large values of n. Hence the n term is ignored. Of course, for small values of n, it may be important. However, Big-Oh is mainly concerned with large values of n.

Formal Definition:

We use O(f), called *big – oh* which represents the class of functions that grow to faster than f. This means that for all values of n greater than some threshold n_0, all of the functions in O(f) have values that are no greater than f. The class O(f) has f as an upper bound, so none of the functions in this class grow faster than f. Formally this means that if $g(x) \in O(f), g(n) \leq c * f(n)$ for all $n \geq n_0$ (where c is a positive constant).

i.e. $\boldsymbol{f(n) = O(g(n))}$ if there exists a constant $c, n_0 \in R^+$ such that for all $n \geq n_0, f(n) \leq c * g(n)$.

Examples: The following points are facts that you can use for Big-Oh problems:

- $1 \leq n$ for all $n \geq 1$
- $n \leq n^2$ for all $n \geq 1$
- $2^n \leq n!$ for all $n \geq 4$
- $\log_2 n \leq n$ for all n ≥ 2

- $n \leq n * \log_2 n$ for all $n \geq 2$

Example1: ***f(n)=10n+5*** and ***g(n)=n***. Show that ***f(n)*** is ***O(g(n)).***

To show that ***f(n)*** is ***O(g(n))*** we must show that constants c and n_0 such that

$f(n) \leq c * g(n)$ for all $n \geq n_0$

Or $10n + 5 \leq c * n$ for all $n \geq n_0$

Try $c = 15$. Then we need to show that $10n + 5 \leq 15n$

Solving for n we get: $5 < 5n$ or $1 \leq n$.

So $f(n) = 10n + 5 \leq 15.g(n)$ for all $n \geq 1$.

$(c = 15, n_0 = 1)$.

∴ ***f(n)=10n+5*** is ***O(g(n))***

Example2: $\boldsymbol{f(n) = 3n^2 + 4n + 1}$**.** Show that $\boldsymbol{f(n) = O(n^2)}$.

$4n <= 4n^2$ for all $n \geq 1$ and $1 \leq n^2$ for all $n \geq 1$

$3n^2 + 4n + 1 \leq 3n^2 + 4n^2 + n^2$ for all $n \geq 1$

$\leq 8n^2$ for all $n \geq 1$

So we have shown that $f(n) \leq 8n^2$ for all $n \geq 1$

Therefore, $\boldsymbol{f(n)\ is\ O(n2)}$ $\boldsymbol{(c = 8, n_0 = 1)}$

Typical Orders

Here is a table of some typical cases. This uses logarithms to base 2, but these are simply proportional to logarithms in other base.

N	$O(1)$	$O(\log_2 n)$	$O(n)$	$O(n * \log_2 n)$	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096
1024	1	10	1,024	10,240	1,048,576	1,073,741,824

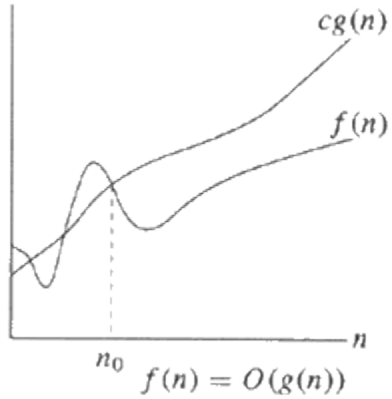
Demonstrating that a function ***f(n)*** is ***big-O*** of a function ***g(n)*** requires that we find specific constants ***c*** and $\boldsymbol{n_0}$ for which the inequality holds (and show that the inequality does in fact hold).

Big-Oh expresses an ***upper bound*** on the growth rate of a function, for sufficiently large values of n.

An ***upper bound*** is the best algorithmic solution that has been found for a problem.

In simple words, $f\ (n)\ = O(g(n))$ means that the growth rate of *f(n)* is less than or equal to *g(n)*.

> The ***big-oh*** 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, the value of $f(n)$ always lies on or below $cg(n)$.



Exercise: $f\ (n)\ =\ (3/2)\ n^2 + (5/2)n - 3$, Show that f(n)= O($n^2$)

Big-O Theorems

> For all the following theorems, assume that f(n) is a function of n and that k is an arbitrary constant.

Theorem 1: k is O(1),
Theorem 2: A polynomial is O(the term containing the highest power of n).

> Polynomial's growth rate is determined by the leading term
> - If *f(n)* is a polynomial of degree *d*, then *f(n)* is $O(n^d)$
>
> In general, f(n) is big-O of the dominant term of f(n).

Theorem 3: k*f(n) is O(f(n))

Constant factors may be ignored
E.g. f(n) =$7n^4+3n^2+5n+1000$ is $O(n^4)$

Theorem 4(Transitivity): If f(n) is O(g(n))and g(n) is O(h(n)), then f(n) is O(h(n)).

Theorem 5: For any base b, $\log_b(n)$ is $O(\log_b n)$.

All logarithms grow at the same rate

$\log_b n$ is $O(\log_d n)$ $\forall b, d > 1$

Theorem 6: Each of the following functions is big-O of its successors:

$$K, n, \log_b n, \quad n\log_b n, n^2, n \text{ to higher powers}, 2^n, \quad 3^n, \text{larger constants to the } n^{th} \text{ power}, n!, n^n.$$

f(n)= $3n\log_b n + 4\log_b n+2$ is $O(n\log_b n)$ and $O(n^2)$ and $O(2^n)$

Properties of the O Notation

Higher powers grow faster

- n^r is $O(n^s)$ if $0 \leq r \leq s$

Fastest growing term dominates a sum

- If $f(n)$ is $O(g(n))$, then $f(n) + g(n)$ is $O(g)$

  E.g $5n^4 + 6n^3$ is $O(n^4)$

Exponential functions grow faster than powers, i.e. $\forall$b > 1 and n $\geq$ 0

E.g. n^{20} is $O(1.05^n)$

Logarithms grow more slowly than powers

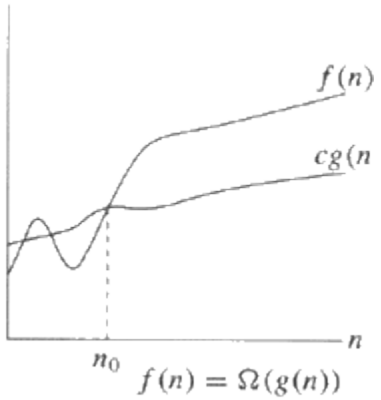
- $\log_b n$ is $O(nk)$ $\forall$ b > 1 and k >= 0

  E.g. $\log_2 n$ is $O(n^{0.5})$

Big-Omega Notation

Just as O-notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Formal Definition: *A function f(n) is Ω(g (n)) if there exist constants c and k* ∈ R^+ *such that f(n) >=c. g(n) for all n>=k.* f(n)= *Ω(g(n))* means that f(n) is greater than or equal to some constant multiple of g(n) for all values of n greater than or equal to some k.



Example: If f(n) =$n^{2,}$ then f(n)= *Ω(n)*
In simple terms, f(n)= *Ω(g (n))* means that the growth rate of f(n) is greater than or equal to g(n).

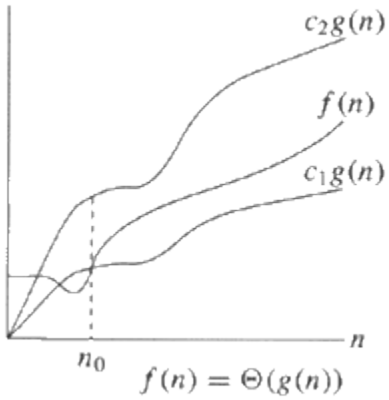
Theta Notation

A function f (n) belongs to the set of Θ (g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1.g(n) and c2.g(n), for sufficiently large values of n.

Formal Definition: A function f (n) is Θ (g(n)) if it is both *O(g(n))* and *Ω (g(n))*. In other words, there exist constants c1, c2, and k >0 such that

c1.g (n)<=f(n)<=c2. g(n) for all n >= k

If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n).
In simple terms, f(n)= Θ (g(n)) means that f(n) and g(n) have the same rate of growth.



Example:
1. If f(n)=2n+1, then f(n) = Θ (n)
2. f(n) =$2n^2$ then f(n)=O(n^4), f(n)=O(n^3), f(n)=O(n^2)

All these are technically correct, but the last expression is the best and tight one. Since $2n^2$ and n^2 have the same growth rate, it can be written as f(n)= Θ(n^2).

Little-o Notation

Big-Oh notation may or may not be asymptotically tight, for example:

$2n^2 = O(n^2)$
$= O(n^3)$

f(n)=o(g(n)) means for all c>0 there exists some k>0 such that f(n)<c.g(n) for all n>=k. Informally, f(n)=o(g(n)) means f(n) becomes insignificant relative to g(n) as n approaches infinity.

Example: f(n)=3n+4 is $o(n^2)$

In simple terms, f(n) has less growth rate compared to g(n).

g(n)= $2n^2$ g(n) =$o(n^3)$, $O(n^2)$, g(n) is not $o(n^2)$.

Little-Omega (ω notation)

Little-omega (ω) notation is to big-omega (Ω) notation as little-o notation is to Big-Oh notation. We use ω notation to denote a lower bound that is not asymptotically tight.

Formal Definition: f(n)= ω (g(n)) if there exists a constant no>0 such that 0<= c. g(n)<f(n) for all n>=k.

Example: $2n^2=\omega(n)$ but it's not ω (n^2).

Relational Properties of the Asymptotic Notations

Transitivity

- if f(n)=Θ(g(n)) and g(n)= Θ(h(n)) then f(n)=Θ(h(n)),
- if f(n)=O(g(n)) and g(n)= O(h(n)) then f(n)=O(h(n)),
- if f(n)=Ω(g(n)) and g(n)= Ω(h(n)) then f(n)=Ω (h(n)),
- if f(n)=o(g(n)) and g(n)= o(h(n)) then f(n)=o(h(n)), and
- if f(n)=ω (g(n)) and g(n)= ω(h(n)) then f(n)=ω (h(n)).

Symmetry

- f(n)=Θ(g(n)) if and only if g(n)=Θ(f(n)).

Transpose symmetry

- f(n)=O(g(n)) if and only if g(n)=Ω(f(n)),
- f(n)=o(g(n)) if and only if g(n)=ω(f(n)).

Reflexivity

- f(n)=Θ(f(n)),
- f(n)=O(f(n)),
- f(n)=Ω(f(n)).

Recurrence Relations

Recurrence relations can be directly derived from a recursive algorithm, but they are in the form that doesn't allow us to quickly determine how efficient the algorithm is. To do that we need to convert the set of recursive equations into what is called closed form by removing the recursive nature of the equations. This is done by a series of repeated substitutions until we can see the pattern that develops. The easiest way to see this process is by a series of examples.

A recurrence relation can be expressed in two ways. The first is used if there are just a few simple cases for the formula.

$$T(n) = 2T(n-2) - 15$$
$$T(2) = 40$$
$$T(1) = 40$$

The second way is the direct solution is applied for a larger number of cases:

$$T(n) = \begin{cases} 4 & \text{for n} \leq 4 \\ 4T(n/2) - 1 & \text{otherwise} \end{cases}$$

These forms are equivalent. We can convert from the second to the first by just using those values for which we have the direct answer. This means that the second recurrence relation above could also be given as

$$T(n) = 4T(n/2) - 1$$
$$T(4) = 4$$
$$T(3) = 4$$
$$T(2) = 4$$
$$T(1) = 4$$

Consider the following relation:

$$T(n) = 2T(n-2) - 15$$
$$T(2) = 40$$
$$T(1) = 40$$

We will want to substitute an equivalent value for $T(n-2)$ back into the first equation. To do so, e replace every n in the first equation with $n-2$, giving:

$$T(n-2) = 2T(n-2-2) - 15$$

$$= 2T(n-4) - 15$$

But now we can see when this substitution is done, we will still have $T(n-4)$ to eliminate. If you think ahead, you will realize that there will be a series of these values that we will need. As a first step, we create a set of these equations for successively smaller values:

$$T(n-2) = 2T(n-4) - 15$$

$$T(n-4) = 2T(n-6) - 15$$

$$T(n-6) = 2T(n-8) - 15$$

$$T(n-8) = 2T(n-10) - 15$$

$$T(n-10) = 2T(n-12) - 15$$

Now we begin to substitute back into the original equation. We will be careful not simplify the resulting equation too much because that will make the pattern more difficult to see. Doing the substitution gives us:

$T(n) = 2T(n-2) - 15 = 2(2T(n-4) - 15) - 15$

$T(n) = 4T(n-4) - 2 * 15 - 15$

$T(n) = 4(2T(n-6) - 15) - 2 * 15 - 15$

$T(n) = 8T(n-6) - 4 * 15 - 2 * 15 - 15$

$T(n) = 8(2T(n-8) - 15) - 4 * 15 - 2 * 15 - 15$

$T(n) = 16T(n-8) - 8 * 15 - 4 * 15 - 2 * 15 - 15$

$T(n) = 16(2T(n-10) - 15) - 8 * 15 - 4 * 15 - 2 * 15 - 15$

$T(n) = 32T(n-10) - 16 * 15 - 8 * 15 - 4 * 15 - 2 * 15 - 15$

$T(n) = 32(2T(n-12) - 15) - 16 * 15 - 8 * 15 - 4 * 15 - 2 * 15 - 15$

$T(n) = 64T(n-12) - 32 * 15 - 16 * 15 - 8 * 15 - 4 * 15 - 2 * 15 - 15$

You are probably beginning to see a pattern develop here. First, we notice that each new term at the end of the equation is *-15* multiplied by the next higher power of 2. Second, we notice that the coefficient of the recursive call to *T* is going through a series of powers of 2. Third, we notice that the value that we are calling *T* with keeps going down by 2 each time.

Now, you might wonder, when does this process end? If we look back at the original equation, you will see that we have a fixed for *T (2)* and *T (1).* How many times would we have to substitute back into this equation to get to either of these values? We can see that $2 = n - (n-2)$ if *n* is even. This seems to indicate that we would substitute back into this equation $\left[\frac{n-2}{2}\right] - 1$ times giving $n/2 - 1$ terms based on -15 in the equation, and the power of the coefficient of T will be $n/2 - 1$. To see this, consider what we would have if the value of *n* was 14. In this case the previous sentence indicates that we would have substituted five times, would have six terms based on *-15,* and would have 2^6 for the coefficient of *T(2).* If you look closely at the last equation and substitute 14 in for *n,* you will see that this is exactly what we have.

What if *n* is an odd number? Will these formulas still work? Let's consider an *n* value of 13. In the above equation, the only thing that would change is that *T* would have a value of 1 instead of 2,

but by out equations, $n/2 - 1$ is 5 (not 6) when n is 13. For odd *n*, we will use *n/2* instead of *n/2 – 1*. We will have two cases in our answer:

$T(n) = 2^{\left(\frac{n}{2}-1\right)}T(2) - 15 * \sum_{i=0}^{\left(\frac{n}{2}\right)-1} 2^i$ if *n* is even.

i.e. $T(n) = 2^{(n/2)-1} * 40 - 15 * \sum_{i=0}^{\left(\frac{n}{2}\right)-1} 2^i$ if n is even.

And $T(n) = 2^{\left(\frac{n}{2}\right)}T(1) - 15 * \sum_{i=0}^{\left(\frac{n}{2}\right)} 2^i$ if n is odd.

i.e. $T(n) = 2^{\left(\frac{n}{2}\right)} * 40 - 15 * \sum_{i=0}^{\left(\frac{n}{2}\right)} 2^i$ if n is odd.

Now, by simplifying the summation, for an even value of n we get

$$T(n) = 2^{\left(\frac{n}{2}\right)-1} * 40 - 15 * (2^{\left(\frac{n}{2}\right)} - 1)$$

$$= 2^{(n/2)} * 20 - 2^{(n/2)} * 15 + 15$$

$$= 2^{\left(\frac{n}{2}\right)}(20 - 15) + 15$$

$$= 2^{(n/2)} * 5 + 15$$

And, if n is odd, we get

$$T(n) = 2^{\left(\frac{n}{2}\right)} * 40 - 15 * (2^{\left(\frac{n}{2}\right)+1} - 1)$$

$$= 2^{(n/2)} * 40 - 2^{(n/2)} * 30 + 15$$

$$= 2^{\left(\frac{n}{2}\right)}(40 - 30) + 15 = 2^{\left(\frac{n}{2}\right)} * 10 + 15$$

1.6. Master Method

The Master Method is used for solving the following types of recurrence

T (n) = a T+ f($\frac{n}{b}$) f(n) with a≥1 and b≥1 be constant & f(n) be a function and $\frac{n}{b}$ can be interpreted as Let T (n) is defined on non-negative integers by the recurrence.

T (n) = a T ($\frac{n}{b}$)+ f (n)

In the function to the analysis of a recursive algorithm, the constants and function take on the following significance:

- n is the size of the problem.
- a is the number of subproblems in the recursion.

- n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- f (n) is the sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the subproblems.
- It is not possible always bound the function according to the requirement, so we make three cases which will tell us what kind of bound we can apply on the function.

Master Theorem:

It is possible to complete an asymptotic tight bound in these three cases:

$$T(n)=\begin{cases}\Theta\left(n^{\log_b a}\right) & f(n)=O\left(n^{\log_b a-\varepsilon}\right)\\ \Theta\left(n^{\log_b a}\log n\right) & f(n)=\Theta\left(n^{\log_b a}\right)\\ \Theta(f(n)) & f(n)=\Omega\left(n^{\log_b a+\varepsilon}\right)\text{AND}\\ & af(n/b)<cf(n)\ \text{forlargen}\end{cases}\quad \begin{matrix}\varepsilon>0\\ c<1\end{matrix}$$

Case1: If f (n) = $O(n^{\log_b a-\varepsilon})$ for some constant ε >0, then it follows that:

T (n) = Θ ($n^{\log_b a}$)

Example:

T (n) = 8 T $\left(\frac{n}{2}\right) + 1000n^2$ apply master theorem on it.

Solution:

Compare T (n) = 8 T $\left(\frac{n}{2}\right) + 1000n^2$ with

T (n) = a T$\left(\frac{n}{b}\right) + f(n) with\ a \geq 1\ and\ b > 1$

a = 8, b=2, f (n) = 1000 n^2, $\log_b{}^a = \log_2 8 = 3$

Put all the values in: f (n) == $O(n^{\log_b a-\varepsilon})$

1000 n^2 = O ($n^{3-\varepsilon}$)

If we choose ε=1, we get: 1000 n^2 = O (n^{3-1}) = O (n^2)

Since this equation holds, the first case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

T (n) = Θ ($n^{\log_b a}$) Therefore: T (n) = Θ (n^3)

Case 2: If it is true, for some constant k ≥ 0 that:

F(n) = Θ ($n^{\log_b a}\ log^k n$)then it follows that: T(n) = Θ ($n^{\log_b a}\ log^{k+1} n$)

Example:

T (n) = $2T\left(\frac{n}{2}\right) + 10n$, solve the recurrence by using the master method.

As compare the given problem with T (n) = a T ($\frac{n}{b}$) f(n) with a≥1 and b≥1 a = 2, b=2, k=0, f (n) = 10n, $\log_b a = \log_2 2 = 1$

Put all the values in f (n) =Θ ($n^{\log_b a} log^k n$), we will get

$10n = \Theta(n^1) = \Theta(n)$ which is true.

Therefore: T (n) =Θ ($n^{\log_b a} log^{k+1} n$)

= Θ (n log n)

Case 3: If it is true f(n) =Ω ($n^{\log_b a+\varepsilon}$)for some constant ε >0 and it also true that: $af\left(\frac{n}{b}\right) \leq cf(n)$for some constant c<1 for large value of n ,then :

1. T (n) =Θ((f (n))

Example: Solve the recurrence relation: T(n) = $2T\left(\frac{n}{2}\right) + n^2$

Solution: Compare the given problem with T(n)= aT $\left(\frac{n}{b}\right)$ f(n) with a≥1 and b≥1

a= 2, b =2, f (n) = n^2, $\log_b a = \log_2 2 = 1$

Put all the values in f(n)= Ω ($n^{\log_b a+\varepsilon}$)..... (Eq. 1)

If we insert all the value in (Eq.1), we will get

$n^2 = \Omega(n^{1+\varepsilon})$ put ε =1, then the equality will hold.

$n^2 = \Omega(n^{1+1}) = \Omega(n^2)$

Now we will also check the second condition: $2(\frac{n}{2})^2 \leq cn^2 \Rightarrow \frac{1}{2}n^2 \leq cn^2$

If we will choose c =1/2, it is true: $\frac{1}{2}n^2 \leq \frac{1}{2}n^2$ ∀ n ≥1 So it follows: T(n)=Θ ((f(n)),T(n) =$\Theta(n^2)$

- Exercise: Apply the Master Theorem on the following problems to determine the order of $T(n)$:

A. $T(n) = 6T\left(\frac{n}{3}\right) + n$

B. $T(n) = 6T\left(\frac{n}{3}\right) + n^2$

C. $T(n) = 6T\left(\frac{n}{3}\right) + n\sqrt{n}$

D. $T(n) = 9T\left(\frac{n}{3}\right) + n^2$

Reading Assignment

- **Review of elementary data structures**
- **Heap and Heap Sort**
- **Hashing**
- **sets representation**
- **UNION, FIND operation**