CHAPTER ONE

1. ALGORITHM ANALYSIS CONCEPTS

1.1. 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:

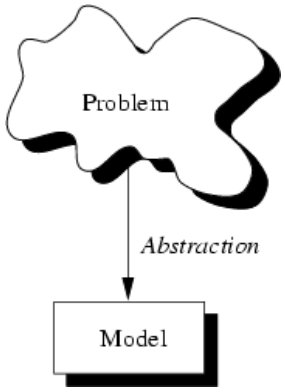
- A way to organize the data
- Sequence of steps to solve the problem

The way data are organized in a 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, a program is nothing but data structures plus algorithms.

A data structure is a systematic way of organizing and accessing data and an algorithm is a step-by-step procedure for performing some task in a finite amount of time.

1.1.1. 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).

i. Abstract Data Types

An abstract data type (ADT) is a set of objects together with a set of operations. Abstract data types are mathematical abstractions; nowhere in an ADT's definition is there any mention of how the set of operations is implemented. Objects such as lists, sets, and graphs, along with their operations, can be viewed as ADTs, just as integers, real's, and booleans are data types. Integers, real's, and booleans have operations associated with them, and so do ADTs. For the set ADT, we might have such operations as add, remove, size, and contains.

An ADT is a mathematical model of a data structure that specifies the type of the data stored, the operations supported on them, and the types of the parameters of the operations. An ADT specifies what each operation does, but not how it does it.

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.

ii. Abstraction

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

The notion of abstraction is to distill a complicated system down to its most fundamental parts and describe these parts in a simple, precise language. Typically, describing the parts of a system involves naming them and explaining their functionality. Applying the abstraction paradigm to the design of data structures gives rise to abstract data types (ADTs).

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?

1.1.2. 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

What makes an algorithm good?

- ***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.

1.2. Algorithm Analysis Concepts

Algorithm analysis refers to the process of determining the amount of computing time and storage space required by different algorithms. In other words, it's a process of predicting the resource requirement of algorithms in a given environment.

It is the study of efficiency of programs. Input size of the program, machine type used, implementation quality, running time of algorithm and others affect the efficiency of a program.

Once an algorithm is given for a problem and determined to be correct, the next step is to determine the amount of resources that the algorithm will require. This step is called algorithm analysis.

In order to solve a problem, there are many possible algorithms. One has to be able to choose the best algorithm for the problem at hand using some scientific method. To classify some data structures and algorithms as good, we need precise ways of analyzing them in terms of resource requirement. The main resources are:

- Running Time
- Memory Usage
- Communication Bandwidth

Algorithm analysis tries to estimate these resources required to solve a problem at hand.
Running time is usually treated as the most important since computational time is the most precious resource in most problem domains.

There are two approaches to measure the efficiency of algorithms:

- **Empirical (Experimental) method**: Programming competing algorithms and trying them on different instances. This method is used for absolute time measurement.
- **Analytical (Theoretical) Method**: Determining the quantity of resources required mathematically (like execution time, memory space, etc.) needed by each algorithm.

However, it is difficult to use actual clock-time as a consistent measure of an algorithm's efficiency, because clock-time can vary based on many things. For example,

- Specific processor speed
- Current processor load
- Specific data for a particular run of the program
  - Input Size
  - Input Properties
- Operating Environment

Accordingly, we can analyze an algorithm according to the number of operations required, rather than according to an absolute amount of time involved. This can show how an algorithm's efficiency changes according to the size of the input.

1.2.1. Complexity Analysis

Complexity Analysis is the systematic study of the cost of computation, measured either in time units or in operations performed, or in the amount of storage space required.
The goal is to have a meaningful measure that permits comparison of algorithms independent of operating platform.
There are two things to consider:

- **Time Complexity**: Determine the approximate number of operations required to solve a problem of size n. The running times of operations on the data structure should be as small as possible.
- **Space Complexity:** Determine the approximate memory required to solve a problem of size n. The data structure should use as little memory as possible.

The critical resource for a program is most often its running time. *Running time is the amount of time that any algorithm takes to run.* However, you cannot pay attention to running time alone. You must also be concerned with other factors such as the space required to run the program (both main memory and disk space). The primary analysis tool we use in this course involves characterizing the running times of algorithms and data structure operations. Running time is a natural measure of "goodness," since time is a precious resource - computer solutions should run as fast as possible.

Many factors affect the running time of a program. Some relate to the environment in which the program is compiled and run. Such factors include the speed of the computer's CPU, bus, and peripheral hardware. The programming language and the quality of code generated by a particular compiler can have a significant effect also. It is also affected by amount of input that it must process, the hardware environments (like the processor, memory, disk, etc.) and

software environments (such as operating system, programming language, compiler, interpreter, etc.) in which the algorithm is implemented, compiled, and executed.

Nevertheless, in spite of the possible variations that come from different environmental factors, we would like to focus on the relationship between the running time of an algorithm and the size of its input. The running time of an algorithm is thus a function of the input size.

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 (constant time, linear time, logarithmic time, quadratic time, exponential time or other).

1.2.2. Complexity of Algorithms

There is no generally accepted set of rules for algorithm analysis. However, an exact count of operations is commonly used. Arbitrary time unit is assumed for analyzing the algorithm, and we have the following set of analysis rules to be considered in so doing the analysis.

Rule 1: Basic operations

The execution of the following operations takes one (1) time unit.

- Assignment Operation
- Single Input/ Output Operation
- Single Boolean Operations
- Single Arithmetic Operations
- Function Return

Rule 2: Selection statements

Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection. This can be an overestimate in some cases, but it is never an underestimate.

Rule 3: Loops

Running time for a loop is equal to the running time for the statements inside the loop body multiplied (*) by number of iterations of the loop.

Rule 4: Nested Loops

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.

Rule 5: Function call

Running time of a function call is 1 for setup + the time for any parameter calculations + the time required for the execution of the function body.

Rule 6: Consecutive statements

For consecutive statements, the running time will be computed as the sum of the running time of the separate blocks of code.

The following examples show us how running time of the code fragments is computed.

Examples 1:

```
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.
- ✓ For the loop (for):
  - 1 assignment (i = 0), *n+1* tests, and *n* increments.
  - *n* loops of 2 units for an assignment and addition.
- ✓ 1 for the return statement.

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

Example 2:

```
 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 addition.
- ✓ 1 for the return statement.

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

Example 3:

```
void func( )
   {      int x=0, i=0, 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)*

Example 4:

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

Time Units to Compute

- 1 for the assignment.
- 1 assignment, n+1 tests, and n increments for the loop expression (for).
- n loops of 4 units for an assignment, addition, and two multiplications.
- 1 for the return statement.

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

Example 5:

```
void func ( )
{
  int i =1, sum = 0;
  while (i < = n)
  {
   for (int j =0;j<n; j++)
    {
     sum = i + j;
    }
   i ++;
  }
}
```

Time Units to Compute

- 1 for the first assignment (i = 1).
- 1 for the second assignment (sum = 0).
- In the while loop
  - n+1 tests
  - n loops of the following
    - For the for loop

- 1 assignment, n+1 tests, and n increments.
- n loops of 2 units for an assignment and addition.

○ 1 for the increment operation.

T (n) = $1+1+(n+1)+n[(1+n+1+n+2n)+1] = \mathbf{4n^2+4n+3} = O(n^2)$

Example 6:

```
int k=0;
for (int i=1; i<n; i*=2)
  for(int j=1;j<=n;j++)
    k++;
```

Time Units to Compute

- 1 for the first assignment (k = 0).
- For the first loop (for)
  - 1 assignment, $1+\log_2 n$ tests, and $\log_2 n$ multiplication of i*=2.
  - $\log_2 n$ iterations of the following
    - For the second loop (for)
      - 1 assignment, n+1 tests, and n increments.
      - n loops of one unit (increment operation)

T (n) = $1+1+1+\log_2 n +\log_2 n+\log_2 n[(1+n+1+n)+n] = \mathbf{3}n\log_2 n+\mathbf{4}\log_2 n+\mathbf{3} = O(n\log_2 n)$

1.2.3. 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;
}}
```

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

Example:

Suppose we have hardware capable of executing 10^6 instructions per second. How long would it take to execute an algorithm whose complexity function was T (n) = $2n^2$ on an input size of n=10^8?

Solution

The total number of operations to be performed would be T (10^8):

$T(10^8) = 2*(10^8)^2 = 2*10^{16}$

The required number of seconds required would be given by

$T(10^8)/10^6$ so:

Running time $=2*10^{16}/10^6 = 2*10^{10}$

The number of seconds per day is 86,400 so this is about 231,480 days (634 years).

Exercises

Determine the run time equation and complexity of each of the following code segments.

1.
```
for (i=0;i<n;i++)
    for (j=0;j<n; j++)
        sum=sum+i+j;
```

2.
```
for(int i=1; i<=n; i++)
        for (int j=1; j<=i; j++)
        sum++;
```

What is the value of the sum if n=20?

3.
```
int  k=0;
for (int i=0; i<n; i++)
    for (int j=i; j<n; j++)
        k++;
```

What is the value of k when n is equal to 20?

4.
```
int k=0;
for (int i=1; i<n; i*=2)
    for(int j=1; j<n; j++)
        k++;
```

What is the value of k when n is equal to 20?

5.
```
int x=0;
for(int i=1;i<n;i=i+5)
    x++;
```
What is the value of x when n=25?

6.
```
int x=0;
for(int k=n;k>=n/3;k=k-5)
    x++;
```
What is the value of x when n=25?

7.
```
int x=0;
for (int i=1; i<n;i=i+5)
   for (int k=n;k>=n/3;k=k-5)
      x++;
```
What is the value of x when n=25?

8.
```
int x=0;

for(int i=1;i<n;i=i+5)
    for(int j=0;j<i;j++)
       for(int k=n;k>=n/2;k=k-3)
            x++;
```
What is the correct big-Oh Notation for the above code segment?

1.3. 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.

1.4. Asymptotic Analysis

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.

The seven most important functions used in the analysis of algorithms are *constant*, *linear*, *logarithm*, *N log N*, *quadratic*, *cubic* and *exponential* functions. We use only these seven simple functions for almost all the analysis we do in this course.

The simplest function we can think of is the **constant function**. This is the function, f (n) = c, for some fixed constant c, such as c = 5, c = 27, or c = 2^{10}. That is, for any argument n, the constant function f (n) assigns the value c. In other words, it doesn't matter what the value of n is, f (n) is always be equal to the constant value c. Since we are most interested in integer functions, the most fundamental constant function is g(n) = 1, and this is the typical constant function we use in this course. Note that any other constant function, f (n) = c, can be written as a constant c times g(n). That is, f (n) = c g(n) in this case.

One of the interesting and sometimes even surprising aspects of the analysis of data structures and algorithms is the ubiquitous presence of the **logarithm function**, f(n) = $\log_b n$, for some constant b > 1. This function is defined as $x = log_b n$ if and only if $b^x = n$. By definition, $log_b 1 = 0$. The value *b* is known as the base of the logarithm.

Another simple yet important function is the **linear function**, $f(n) = n$. That is, given an input value n, the linear function f assigns the value n itself. This function arises in algorithm

analysis any time we have to do a single basic operation for each of n elements. For example, comparing a number x to each element of an array of size n requires n comparisons. The linear function also represents the best running time we can hope to achieve for any algorithm that processes a collection of n objects that are not already in the computer's memory, since reading in the n objects itself requires n operations.

The next function is the **n-log-n function**, $f(n) = n \log n$. That is, the function that assigns to an input n the value of n times the logarithm base of n. This function grows a little faster than the linear function and a lot slower than the quadratic function. Thus, as we show on several occasions, if we can improve the running time of solving some problem from quadratic to n-log-n, we have an algorithm that runs much faster in general.

Another function that appears quite often in algorithm analysis is the **quadratic function**, $f(n) = n^2$. That is, given an input value n, the function f assigns the product of n with itself (in other words, "n squared"). The main reason why the quadratic function appears in the analysis of algorithms is that there are many algorithms that have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. Thus, in such cases, the algorithm performs $n \cdot n = n^2$ operations.

The other function is the **cubic function**, $f(n) = n^3$, which assigns to an input value n the product of n with itself three times. This function appears less frequently in the context of algorithm analysis than the constant, linear, and quadratic functions previously mentioned, but it does appear from time to time.

Another function used in the analysis of algorithms is the **exponential function**, $f(n) = b^n$, where b is a positive constant, called the base, and the argument n is the exponent. That is, function f (n) assigns to the input argument n the value obtained by multiplying the base b by itself n times. In algorithm analysis, the most common base for the exponential function is $b = 2$. For instance, if we have a loop that starts by performing one operation and then doubles the number of operations performed with each iteration, then the number of operations performed in the n^{th} iteration is 2^n.

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)

1.4.1. 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: f (n)= O (g (n)) if there exist c, k ∈ $\mathscr{R}^+$ such that for all n≥ k, f (n) ≤ c.g (n).

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

- 1<=n for all n>=1
- $n<=n^2$ for all n>=1
- 2^n <=n! for all n>=4
- $\log_2 n<=n$ for all n>=2
- $n<=n\log_2 n$ for all n>=2

1) 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 k such that
f(n) <=c.g(n) for all n>=k Or 10n+5<=c.n for all n>=k
Try c=15. Then we need to show that 10n+5<=15n
Solving for n we get: 5<5n or 1<=n.
So f(n) =10n+5 <=15.g(n) for all n>=1. (c=15,k=1).

2) f(n) = $3n^2$ +4n+1. Show that f(n)=O(n^2).

4n <=$4n^2$ for all n>=1 and 1<=n^2 for all n>=1
$3n^2$ +4n+1<=$3n^2$+$4n^2$+n^2 for all n>=1
<=$8n^2$ for all n>=1
So we have shown that f(n)<=$8n^2$ for all n>=1
Therefore, f (n) is O(n^2) (c=8,k=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 n)	O(n)	O(n log n)	O(n^2)	O(n^3)
1	1	0	1	0	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 k for which the inequality holds (and show that the inequality does in fact hold).

Big-O 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. " What is the best that we know we can do?"

Exercise: if $f(n) = (3/2)n^2+(5/2)n-3$, then show that $f(n)= O(n^2)$

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

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(logn). 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
$\log_b n$
n
$n\log_b n$
n^2
n to higher powers
2^n
3^n
larger constants to the n^{th} 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 $O(2^n)$

More Properties of the O Notation

1) Higher powers grow faster
   - n^r is $O(n^s)$ if $0 <= r <= s$
2) 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 $O(n^4)$
3) Exponential functions grow faster than powers, i.e. n^k is $O(b^n)\ \forall b > 1$ and $k >= 0$
   E.g. n^{20} is $O(1.05^n)$
4) Logarithms grow more slowly than powers
   - $\log_b n$ is $O(n^k)\ \forall\ b > 1$ and $k >= 0$
   E.g. $\log_2 n$ is $O(n^{0.5})$

1.4.2. 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 ∈ $\mathcal{R}$+ 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).

1.4.3. 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).

1.4.4. 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).

1.4.5. 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 $\omega(n^2)$.

1.5. Relational Properties of the Asymptotic Notations

i. 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)).

ii. Symmetry

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

iii. 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)).

iv. Reflexivity

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