Chapter 2: Algorithm Analysis

Algorithm Analysis Concepts

Algorithm analysis refers to the process of determining how much computing time and storage that algorithms will require. In other words, it's a process of predicting the resource requirement of algorithms in a given environment.

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

Running time is usually treated as the most important since computational time is the most precious resource in most problem domains.

- The objective of algorithm analysis is
  - to determine how quickly an algorithm executes in practice
  - To measure either time or space requirements of an algorithm
- What to measure
  - Space utilization- amount of memory required
  - Time efficiency- amount of time required to process the data
- What to analyze
  - The most important recourse to analyze is generally the running time
  - Several factors affect the running time of an a program
    - Compiler used
    - Computer used
    - The algorithm used
    - The input to the algorithm
  - The first two are beyond the scope of theoretical model
  - Although they are important, we will not deal with them in the course
  - The last two are the main factors that we deal
  - Typically, the size of the input is an important consideration
- Space utilization and Time efficiency depends on many factors
  - Size of input
  - Speed of machine (computer used)
  - Quality of source code (used algorithm)
  - Quality of compiler
- Focuses on measuring the running time of an algorithm
- For most algorithm, the running time depends on the input

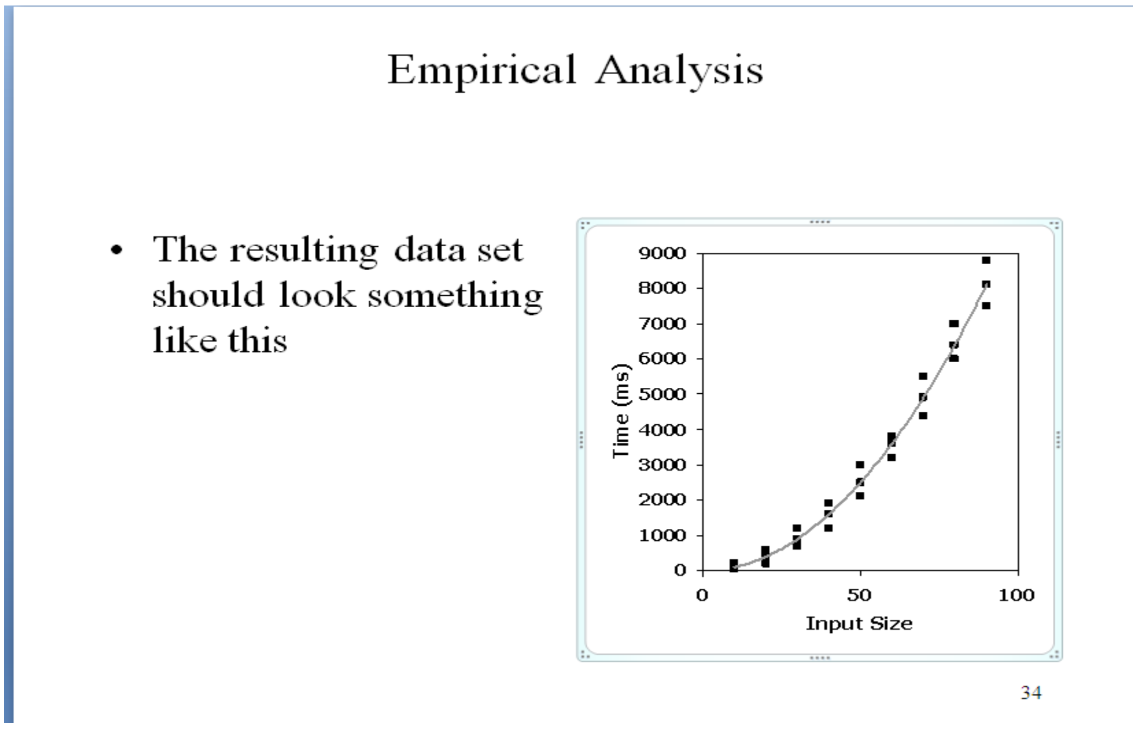
  - An already sorted sequence is easier to sort
- for most algorithm, the running time depends on size of the input
  - Short sequences are easier to sort than long ones
  - That is the running time of most algorithms varies with the input and typically grows with the input size
- Generally, we seek an upper bound on the running time, because everybody likes a guarantee

There are two approaches to measure the efficiency of algorithms:

- Empirical: Programming competing algorithms and trying them on different instances.
- Theoretical: Determining the quantity of resources required mathematically (Execution time, memory space, etc.) needed by each algorithm.

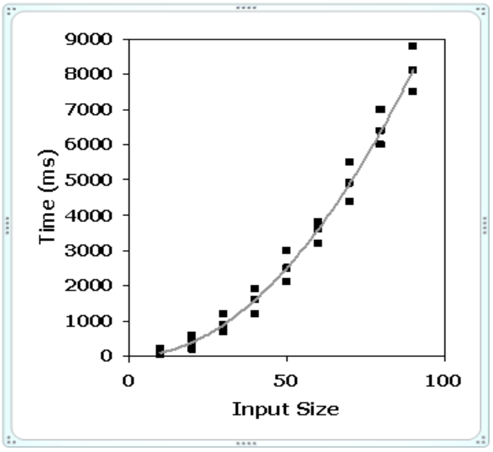
Empirical Analysis

- What you should do
  - Write a program that implements the algorithm
  - Run the program with the data sets (the inputs) of varying size and composition
  - Use the method like clock() (or System.CurrentTime.Millis()) to get an accurate measure of the running time
  - Plot the results



Empirical Analysis

- The resulting data set should look something like this



34

- Is an experimental study
- The result comes from the running time of the program
- It uses system time

Limitations of Empirical Analysis

- It can’t be used in estimating the efficiency of algorithms because the result varies due to variations in
  - Specific processor speed
  - Input size
  - Current processor load
  - SW environment
  - So, in order to compare two algorithms, the same hardware and software environment must be used
- Involves implementing the algorithm and testing it on various instances
- It is necessary to implement and test the algorithm in order to determine its running time
  - But, implementing the algorithm may be difficult
  - This is not the case for theoretical analysis
- The difficulty to know which instances to test it on
- Experiments can be done only on a limited set of inputs, and may not be indicative of the running time on other inputs not included in the experiment
- That is, results may not be indicative of the running time on other inputs not included in the experiment

Theoretical Analysis

- Is in contrast to the “experimental approach”
- Allows us to evaluate the speed (efficiency) of an algorithm in a way that is independent from the HW and SW environment
- Is a general methodology for analyzing the running time of an algorithm
- Uses a high-level description of the algorithm instead of testing one of its implementation
- Takes into account all possible inputs
- Takes an algorithm and produces a function T(n) which depends on the number of operations
- That is, running time is expressed as T(n) for some function T on input size n
- The output comes from the quantity of resources using mathematical concepts
- Does not use system time
- Rather it uses the number of operations (which are expressed in time units) because the number of operations do not vary with
  - Processor speed
  - Input size
  - Current processor load

    - SW environment
- Theoretical analysis allows us to separate the effect of these factors
- Is an approach that can be used to measure or estimate the complexity of an algorithm as the approach do not vary due to variations in the computer systems
- The only factor that affects measure of complexity is the input size of the algorithm
- To avoid this, measure complexity of algorithm for arbitrary number n as

n $\rightarrow \infty$

How do measure complexity of algorithms

- Two steps or phases for this
    - Analysis of algorithm (i.e., theoretical analysis)
    - Order of magnitude
- Step one - analysis of algorithm
    - At this stage we should have an algorithm or select an algorithm
    - By analyzing this algorithm we end up with a function T(n), which is the computing time of an algorithm for input size n
    - T(n)= the number of operations of time units
    - This first step is used to measure the complexity of the algorithm
    - In analyzing an algorithm written in psedocode to determine its complexity, determine the number of operations on the underlying model of computation
    - To obtain an estimate of this number, we count primitive operations

    Primitive operations:

    - Basic computations performed by an algorithm
    - Identifiable in pseudocode
    - Largely independent from the programming language
    - Exact definition not important
    - chosen to reflect the complexity of the total cost of all operations

    Examples of basic operations could be

        - An assignment (Assigning a value to a variable)
        - Calling a method
        - Returning from a method
        - Evaluating an expression
            - An arithmetic operation between two variable (e.g., Performing an arithmetic operations such as addition)
        - A Comparison between to variables (two numbers)
        - Indexing into an array
- We measure execution (running) time in terms of any of the following

  - Arithmetic operations
  - Assignment operations
  - Loop iterations
  - Comparisons
  - Procedure calls (calling a method)
  - Returning from a method

Order of Magnitude

- Refers to the rate at which the storage or time grows as a function of problem size
- Input: T(n), the timing function
- Output: order of T(n) written as O(T(n))
- The output of helps to determine the category to which the algorithm belongs to some known functions (functions whose complexity is known through research and experiment**)**
- Thus, the output is expressed in terms of its relationship to some known functions

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.

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

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.

1.2.3.1. 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 + 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 * 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 + the time for any parameter calculations + the time required for the execution of the function body.
6. Consecutive Statements
   Add the time complexities of each statement
7. Switch statement: Take the complexity of the most expensive case.

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

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 an addition.
	1 for the return statement.
	T (n)= ***1+ (1+n+1+n)+2n+1 = 4n+4 = O(n)***

3. 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)*

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.
	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 so 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 $n=10^8$?

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).

Best, Worst, and Average-Case Complexity

We can say that we are looking for the most suitable algorithm for a specific purpose. For this, we need to analysis the algorithm under specific constraints. An algorithm can be analyzed under three specific cases:

Best case analysis

We analyze the performance of the algorithm under the circumstances on which it works best. In that way, we can determine the upper-bound of its performance. However, you should note that we may obtain these results under very unusual or special circumstances and it may be difficult to find the optimum input data for such an analysis.
The *best case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size *n*.

- Best case is defined as which input of size n is the cheapest among all inputs of size n
- Best case is obtained from the in the input data set that results in best possible performance

Input

- Assumes the input data are found in the most advantageous order for the algorithm
- **For sorting**, the most advantageous order is that data are arranged in the required order
- **For searching**, the best case is that the required item is found at the first position
- Question
    - Which order is the most advantageous for an algorithm that sorts 2 5 8 1 4 (ascending order)

Input size
Assumes that the input size is the minimum possible

- This case executes or causes the fewest number of executions
- So, it computes the lower bound of T(n) and You can not do better

- Is the amount of time the algorithm takes on the smallest possible set of inputs
- Best case behavior is usually not informative and hence usually uninteresting (why?)
- The function is denoted by $T_{best}(n)$ – best case running time of an algorithm for n inputs

Average case analysis

This gives an indication on how the algorithm performs with an average data set. It is possible that this analysis is made by taking all possible combinations of data, experimenting with them, and finally averaging them. However, such an analysis may not reflect the exact behavior of the algorithm you expect from a real-life data set. Nevertheless, this analysis gives you a better idea how this algorithm works for your problem.
The *average-case complexity* of the algorithm is the function defined by an average number of steps taken on any instance of size *n*.

- Is the complexity of an algorithm averaged over all inputs of each size
- Varies between the best case and worst case
- Gives the average performance of an algorithm
- Computes the **optimal** (the one which is found most of the time) **bound of T(n)**
- The average case can be as bad as the worst case

Input (i.e., arrangement of data)

- Assumes that data are arranged in random order
- **For searching**, it assumes that the required item is found at any position or can be missing
- **For sorting**, it assumes that data are arranged in a random order
- **Input size**
- The input size can be any random number as small as the best case or as large as the worst case
- Problems with performing an average case analysis
- It may not be apparent what constitutes an “average” input for a particular problem
- There is no clue as to what can be an average input
- It is difficult to envision the data that would cause an algorithm (e.g., alg. for insertion sort) to exhibit its average behavior
- Average case is difficult to determine
- Problems with performing an average case analysis
    - The average operation count is often
        - Quite difficult to determine or define
        - Quite difficult to analyze
    - Often we assume that all inputs of a given size are equally likely
        - In practice, this assumption may be violated, but randomized algorithms can sometimes force this to hold
- As a result, in several of the algorithms, we limit our analysis to determining the best and worst counts
- We focus on the worst case running time
    - Easier to analyze
    - Crucial to applications such as games, finance and robotics

Worst case analysis.

In contrast to the best-case analysis, this gives you an indication on how bad this algorithm can go, or in other words, gives a lower-bound for its performance. Sometimes, this could be useful in determining the applicability of an algorithm on a mission-critical application. However, this analysis may be too pessimistic for a general application, and it may be difficult to find a test data set that produces the worst case.

- Is the complexity of an algorithm based on worst input of each size
- Is the amount of time the algorithm takes on the worst possible set of inputs
- Is the longest running time for any input of size n

  Input (i.e., arrangement of data)
  - Assumes that the data are arranged in the most **disadvantageous** order
  - **For sorting**, the worst case assumes that data are arranged in opposite order
  - **For searching**, the worst case assumes that the required item is found at last position

Input size
  - Assumes the input size to be infinite
  - That is, it assumes that the input size is a very large number
  - That is, it considers n $\rightarrow$infinity
  - or missing
- This case, causes highest number of executions to be performed by the algorithm
- So, it computes the upper bound of T(n), the longest running time for any input size n
- Commonly used for computing T(n)
- Is a function and is denoted by $T_{worst}(n)$ – the worst case running time of the algorithm for n inputs

The *worst case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*.

We are interested in the worst-case time, since it provides a bound for all input – this is called the "Big-Oh" estimate.

Typically, the worst case analysis is used to estimate the complexity of an algorithm (That is, we compare algorithms at their worst case)

Why use a worst case analysis

The following are reasons for using worst case analysis as an estimate for the complexity of an algorithm
  - The worst case running time of an algorithm provides an upper bound on the running time for all inputs (i.e., all cases)
  - Thus, there is no bad operation than worst case
  - So, knowing it gives us guarantee that the algorithm will never take any longer

- For some algorithms, the worst case occurs fairly often
- That is , Worst case "may" be a typical case
- The average case is often roughly as bad as the worst case
- Average case may be approximately the worst case (($t_j = j / 2$) is roughly quadratic)
- Often it is hard to pick average cases and compute expected times.
- The average case bounds are usually more difficult to compute
- **Conclusion**
- Unless otherwise specified, running time is the worst case

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.

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: $f(n) = O(g(n))$ if there exist $c, k \in \mathscr{R}^+$ such that for all $n \geq k$, $f(n) \leq c.g(n)$.

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	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 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:

$f(n) = (3/2)n^2+(5/2)n-3$
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 $O(\log_d n)$ □$b, d > 1$

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 nth power
n!
n^n

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

Properties of the O Notation

Higher powers grow faster

•n^r □is □□O(n^s) if $0 <= r <= s$

Fastest growing term dominates a sum

• If f(n) is O(g(n)), then f(n) + g(n) is O(g)

E.g $5n^4 + 6n3$ is $O(n^4)$

Exponential functions grow faster than powers, i.e. □is □□O(b^n) □$\forall b > 1$ and $k >= 0$
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 $O(n^{0.5})$

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 $\in \mathscr{R}+$ 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 that 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)= $\Theta(n^2)$.

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: $f(n)= \omega (g(n))$ 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 $\Omega(n)$.

1.5. **Relational Properties of the Asymptotic Notations**

Transitivity

- if $f(n)=\Theta(g(n))$ and $g(n)= \Theta(h(n))$ then $f(n)=\Theta(h(n))$,
- if $f(n)=O(g(n))$ and $g(n)= O(h(n))$ then $f(n)=O(h(n))$,
- if $f(n)=\Omega(g(n))$ and $g(n)= \Omega(h(n))$ then $f(n)=\Omega (h(n))$,
- if $f(n)=o(g(n))$ and $g(n)= o(h(n))$ then $f(n)=o(h(n))$, and
- if $f(n)=\omega (g(n))$ and $g(n)= \omega(h(n))$ then $f(n)=\omega (h(n))$.

Symmetry

- $f(n)=\Theta(g(n))$ if and only if $g(n)=\Theta(f(n))$.

Transpose symmetry

- $f(n)=O(g(n))$ if and only if $g(n)=\Omega(g(n)$,
- $f(n)=o(g(n))$ if and only if $g(n)=\omega(g(n))$.

Reflexivity

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

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;
```
What is the value of sum if n=100?

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?