

Data Structure and Algorithms

Introduction to Data Structures & 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**.

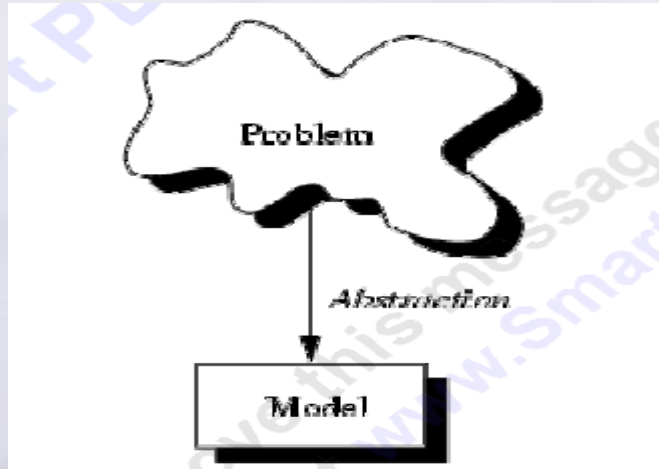
So,

 Data structure = organized data + Operations

 Program = Data structure + Algorithm

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.

Abstract Data Types

- An entity with the properties just described is called an *abstract data type* (ADT).
- An ADT consists of an **abstract data structure** and **operations**. Put in other terms, an ADT is an abstraction of a data structure.
- The ADT specifies:
 - What can be stored in the Abstract Data Type
 - 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.*

Abstraction

- ☐ 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
 - ☐ *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?

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.**

Cont.

- 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.

Cont.

- **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.

Operations on data Structures

☰ The various operations that can be performed over Data Structures are

- Create
- Destroy
- Access
- Update

Primitive Data Structures

- Integers, Real Numbers, Characters, Logical, Pointers,...

Non-Primitive Data Structure

- Non-Primitive data structures are those that can be obtained from the primitive data structures.
- The non-primitive data structures and number of such data structures are not fixed.
- If the allocation of memory is sequential or continuous then such a data structure is called a non-primitive linear structure. Example: arrays, stacks, queues, linked lists, etc.
- If the allocation of memory is not sequential but random or discontinuous then such a data structure is called a **Non-primitive non-linear structure**. Example, Trees, Graphs, etc.

Algorithm Analysis

- **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.
- 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.

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 .

Cont.

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.
- Generally, Complexity analysis is concerned with determining the efficiency of algorithms.

1. Empirical (Computational) Analysis

- ≡ Total running time of the program is considered.
- ≡ It uses the system time to calculate the running time and it can't be used for measuring efficiency of algorithms.
- ≡ This is because the total running time of the program algorithm varies on the:
 - Processor speed
 - Current processor load
 - Input size of the given algorithm
 - and software environment (multitasking, single tasking,...)

What is Efficiency depends on?

- Execution speed (most important)
- Amount of memory used
- Efficiency differences may not be noticeable for small data, but become very important for large amounts of data

Exercise One

- Write a program that displays the total running time of a given algorithm based on different situations such as processor speed, input size, processor load, and software environment (Windows).

2. Theoretical (Asymptotic Complexity) Analysis

- In most real-world examples it is not so easy to calculate the complexity function.
- But not normally necessary, e.g. consider $t = f(n) = n^2 + 5n$; For all $n > 5$, n^2 is largest, and for very large n , the $5n$ term is insignificant
- Therefore we can approximate $f(n)$ by the n^2 term only. This is called *asymptotic complexity*
- An approximation of the computational complexity that holds for large n .
- Used when it is difficult or unnecessary to determine true computational complexity
- Usually it is difficult/impossible to determine computational complexity. So, asymptotic complexity is the most common measure

How do we estimate the complexity of algorithms?

1. Algorithm analysis – Find $f(n)$ - helps to determine the *complexity of an algorithm*
2. Order of Magnitude – $g(n)$ belongs to **$O(f(n))$** – helps to determine the **category of the complexity** to which it belongs.

Analysis Rule:

1. Assume an arbitrary time unit
2. Execution of one of the following operations takes time unit 1
 - Assignment statement Eg. `Sum=0;`
 - Single I/O statement;. E.g. `cin>>sum; cout<<sum;`
 - Single Boolean statement.
 - Single arithmetic. E.g. `a+b`
 - Function return. E.g. `return(sum);`
3. selection statement
 - Time for condition evaluation + the maximum time of its clauses
4. Loop statement
 - $\sum(\text{no. of iteration of Body}) + 1 + n + 1 + n$ (initialization time + checking + update)
5. For function call
 - $1 + \text{time}(\text{parameters}) + \text{body time}$

Estimation of Complexity of Algorithm

- Calculate $T(n)$ for the following

- 1. $k=0;$

$\text{Cout} << \text{"enter an integer"};$

$\text{Cin} >> n;$

For $(i=0; i < n; i++)$

{

$K++;$

}

- $T(n) = 1 + 1 + 1 + (1 + n + 1 + n + n(1))$
 $= 5 + 3n$

- 2. $i=0;$
While ($i < n$)
{
 $x++;$
 $i++;$
}
 $J=1;$
While($j \leq 10$)
{
 $x++;$
 $j++$
}
- $T(n)=1+n+1+n+n+1+11+10+10$
 $=3n+34$

...

3. `for(i=1;i<=n;i++)`

`{`

`for(j=1;j<=n; j++)`

`{`

`k++;`

`}`

`}`

$$T(n)=1+n+1+n+n(1+n+1+n+n)=3n^2+4n+2$$

...

4. $Sum=0;$

$if(test==1)$

{

$for\ (i=1;i\leq n;i++)$

$sum=sum+i$

}

$else$

{

$cout<<sum;$

}

- $T(n)=1+1+Max(1+n+1+n+n+n,1)=4n+4$

Exercise

- Calculate $T(n)$ for the following codes

A. $sum=0;$
 $for(i=1;i\leq n;i++)$
 $for(j=1;j\leq m;j++)$
 $Sum++;$

B. $sum=0;$
 $for(i=1;i\leq n;i++)$
 $for(j=1;j\leq i;j++)$
 $sum++;$

Algorithm Analysis Categories

- Algorithm must be examined under different situations to correctly determine their efficiency for accurate comparisons
- **Best Case Analysis:** Assumes that data are arranged in the most advantageous order. It also assumes the minimum input size.
- E.g.
 - For **sorting** – the best case is if the data are arranged in the required order.
 - For **searching** – the required item is found at the first position.
- Note: Best Case computes the lower boundary of $T(n)$
- It causes fewest number of executions

Worst case analysis

- Assumes that data are arranged in the disadvantageous order.
- It also assumes that the input size is infinite.
- E.g.
 - For **sorting** – data are arranged in opposite required order
 - For **searching** – the required item is found at the end of the item or the item is missing
- It computes the upper bound of $T(n)$ and causes maximum number of execution.

Average case Analysis

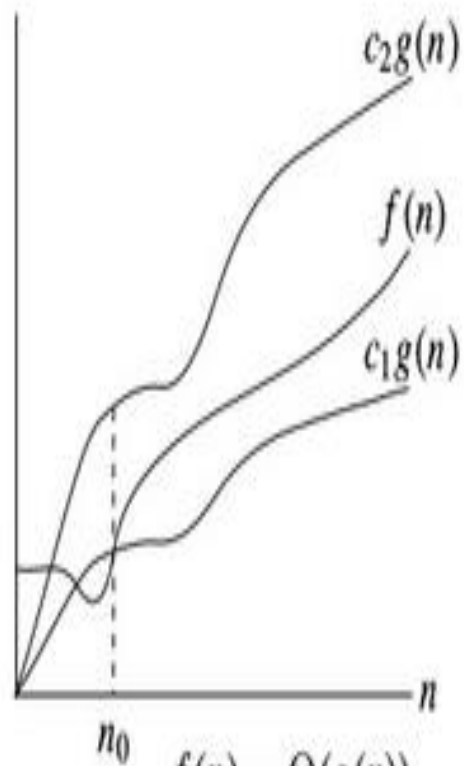
- Assumes that data are found in random order.
- It also assumes random or average input size.
- E.g.
 - For **sorting** – data are in random order.
 - For **searching** – the required item is found at any position or missing.
- It computes optimal bound of $T(n)$
- It also causes average number of execution.
- Best case and average case can not be used to estimate (determine) complexity of algorithms
- Worst case is the best to determine the complexity of algorithms.

Order Of Magnitude

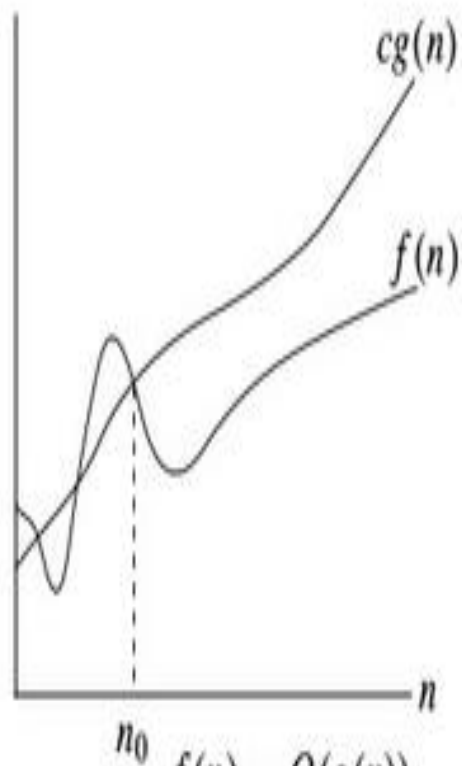
- Order Of Magnitude refers to the rate at which the storage or time grows as function of problem size (function n).
- It is expressed in terms of its relationship to some known functions – asymptotic analysis.
- Asymptotic complexity of an algorithm is an approximation of the computational complexity that holds for large amounts of input data.

Types of Asymptotic Notations

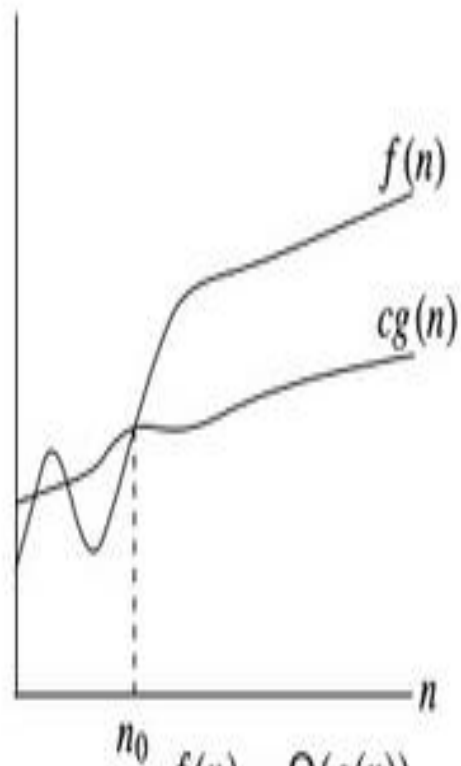
- 1. Big-O Notation
- 2. Big – Omega (Ω)
- 3. Big –Theta (Θ)



(a)



(b)



(c)

1. Big-O Notation

- Definition : The function $T(n)$ is $O(F(n))$ if there exist constants c and N such that $T(n) \leq c.F(n)$ for all $n \geq N$.
- As n increases, $T(n)$ grows no faster than $F(n)$ or in the long run (for large n) T grows at most as fast as F
- It computes the tight upper bound of $T(n)$
- Describes the worst case analysis.

...

, E.g. The growth rate of $f(n) = n^2 + 100n + \log_{10} n + 1000$

n	$f(n)$	n^2		$100n$		$\log_{10} n$		1,000	
	Value	Value	%	Value	%	Value	%	Value	%
1	1,101	1	0.1	100	9.1	0	0.0	1,000	90.83
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	10,000	47.6	10,000	47.6	2	0.001	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

Properties of Big-O Notation

- ☰ If $T(n)$ is $O(h(n))$ and $F(n)$ is $O(h(n))$ then $T(n) + F(n)$ is $O(h(n))$.
- ☰ The function $a \cdot n^k$ is $O(n^k)$ for any a and k
- ☰ The function $\log_a n$ is $O(\log_b n)$ for any positive numbers a and $b \neq 1$

Examples

Find $F(n)$ such that $T(n) = O(F(n))$ for $T(n) = 3n+5$

Solution:

$$3n+5 \leq cn$$

$$c=6, N=2$$

$F(n)=n$ because $c=6$ for $3n+5 \leq 6n$, for All $n \geq 2$

$\Rightarrow T(n) = O(n)$

Example


$$T(n) = n^2 + 5n$$



$$T(n) \leq c.F(n)$$

$$n^2 + 5n \leq c. n^2$$

$$1 + (5/n) \leq c$$

$$F(n) = n^2$$

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

- Therefore if we choose $N=5$, then $c=2$; if we choose $N=6$, then $c=1.83$, and so on.
-  So what are the ‘correct’ values for c and N ? The answer to this question, it should be determined for which value of N a particular term in $T(n)$ becomes the largest and stays the largest.
-  In the above example, the n^2 term becomes larger than the $5n$ term at $n>5$, so $N=5$, $c=2$ is a good choice.

2. Big – Omega

Definition : The function $T(n)$ is $\Omega(F(n))$ if there exist Constants c and N such that

$$T(n) \geq c.F(n) \text{ for all } n \geq N.$$

- As n increases $T(n)$ grows no slower than $F(n)$ or in the long run (for large n) T grows at least as fast as F*
- It computes the tight lower bound of $T(n)$.*
- Describes the best case analysis*

Examples

Find $F(n)$ such that $T(n) = \Omega(F(n))$

for $T(n) = 3n+5$

$F(n) = n$ as $3n+5 \geq c \cdot n$ for $c=1$ and $n=0$

3. Big Theta Notation

Definition: The function $T(n)$ is $\Theta(F(n))$ if there exist constants c_1 , c_2 and N such that $c_1.F(n) \leq T(n) \leq c_2.F(n)$ for all $n \geq N$.

As n increases, $T(n)$ grows as fast as $F(n)$

It computes the tight optimal bound of $T(n)$.

Describes the average case analysis.

Example:

● Find $F(n)$ such that $T(n) = \Theta(F(n))$ for $T(n) = 2n^2$.

● Solution: $c_1n^2 \leq 2n^2 \leq c_2n^2$

● $F(n) = n^2$ because for $c_1 = 1$ and $c_2 = 3$,

● $n^2 \leq 2n^2 \leq 3n^2$ for all n .

4. Little o (small o) notation

Defn: The function $T(n)$ is $o(F(n))$ if $T(n)$ is $O(F(n))$ but $T(n)$ is not $\Theta(F(n))$ or for some constants c, N : if $T(n) < cF(n)$ for all $n \geq N$

As n increases $F(n)$ grows strictly faster than $T(n)$.

It computes the non-tight upper bound of $T(n)$

Describes the worst case analysis

Example:

● Find $F(n)$ such that $T(n) = o(F(n))$ for $T(n) = n^2$.

● Solution: $n^2 < c n^2$


● $F(n) = n^2$ because for $c=2$,

● $n^2 < 2n^2$ for all $n \geq 1$

Big O Analysis

T(n)	Complexity Category Function (F(n))	Big Oh of T(n)
$10\log(n+5)$	<u>Logn</u>	$T(n)=O(\log n)$
C where C is constant	1	$T(n) = O(1)$
$5\sqrt{n}+5$	\sqrt{n}	$T(n) = O(\sqrt{n})$
$5/3(n)+2\log n+2$	N	$T(n) = O(n)$
$10n\log n+2n+2$	<u>nlogn</u>	$T(n)=O(n\log n)$
$3n^2+2n+2$	n^2	$T(n)=O(n^2)$
$5n^3-3$	n^3	$T(n) = O(n^3)$
$8n^n + 2^n n^2$	<u>n^n</u>	$T(n)=O(n^n)$

Assignment

 Write a program that displays the total running time for various input sizes of the complexity category function and plot a graph (input size Vs time) for each function. By observing the graph write functions in ascending order of complexity.

 **Deadline :**

Finding Asymptotic Complexity: Examples

Rules to find Big-O from a given $T(n)$

- Take highest order
- Ignore the coefficients

Reading Assignment

-  Big-O notation
-  Amortized complexity

Finding Big O of given Algorithm

1. for (i=1;i<=n;i++)

● Cout<<i;

● $T(n) = 1+n+1+n+n$

$$O = 3n+2$$

● $T(n) = O(n)$

...

2. For (i=1;i<=n;i++)

- For(j=1;j<=n;j++)

- Cout<<i;

- $T(n) = 1 + n + 1 + n + n(1 + n + 1 + n + n)$

$$O = 3n^2 + 4n + 2$$

- $T(n) = O(n^2)$

...

3. for (i=1; i<=n; i++)

● Cout<<i;

$T(n) = 1+n+1+n+n$

● = $3n+2$

$T(n) = O(n)$

Exercise

Find Big O of the following algorithm

1. for($i=1; i \leq n; i++$)

- $Sum = sum + i;$

- For ($i=1; i \leq n; i++$)

- For($j=1; j \leq m; j++$)

- $Sum++;$

2. if(k==1)

● {

● For (i=1;i<=100;i++)

○ *For (j=1;j<=1000;j++)*

○ Cout<<I

● }

● Else if(k==2)

● {

● For(i=1;i<=n;i=i*2)

● Cout<<i;

● }

● Else

● {

● {for (i=1;i<=n;i++)

● Sum++;

● }

3. for (i=1; i<=n;i++)

- {
- If (i<10)
 - For(j=1;j<=n;j=j*2)
 - Cout<<j;
- Else
 - Cout<<i;
- }

...

4. for (int i=1; i<=N; ++i)

for (int j=i; j>0; --j){}

5. for (int i=1; i<N; i=i*2){}

6. for (int i=0; i<N; ++i)

for (int j=i; j>0; j=j/2){}

...

```
7. for (int i=1; i<N; ++i)
    for (int j=0; j<N; j+=i){}
```

```
8. int i=0, sum=0;
    while (sum < N)
        sum = sum + i++;
9. for (int i=1; i<N; i=i*2)
    for (int j=0; j<i; ++j){}
```

...

```
10.  i=1;
      while (i<n)
      {
        i=i*2;
      }
```

.....

```
11. for ( int i = 0; i < n; i++ )  
    for ( int j = 0; j <= n - i; j++ )  
        int a = i;  
            for (int i = 0; i < n; i ++ )  
                for (int j = 0; j < n*n*n ; j++ )  
                    sum++;
```

...

```
12. void Test( int N){  
    for(int i =0 ; i < N ; i= i*4){  
        for(int j = N ; j > 2; j = j/3){  
            cout<<“I have no idea what this does”;  
        }  
    }  
}
```

...

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

Quiz one - 5 %

☰ 1. *Determine the big-O expression for*

● $T(n) = 3 \log n^2 + 2$

☰ 2 Big O computes the worst case analysis (T/F).