# Asymptotic Notation

## Asymptotic Notation

*The main idea of **asymptotic analysis** is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, mainly because this analysis doesn't require algorithms to be implemented and time taken by programs to be compared*

It is often used to describe how the size of the input data affects an algorithm's usage of computational resources. Running time of an algorithm is described as a function of input size n for large n.

- Asymptotic Notation
- Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm.
- But when we calculate the complexity of an algorithm it does not provide the exact amount of resource required.
- So instead of taking the exact amount of resource, we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm.
- We use that general form (Notation) for analysis process.

- Asymptotic notation of an algorithm is a mathematical representation of its complexity.
- Note - In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity can be Space Complexity or Time Complexity).

- For example, consider the following time complexities of two algorithms...

- Algorithm 1 : 5n2 + 2n + 1
- Algorithm 2 : 10n2 + 8n + 3
- Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. 'n' value).
- In above two time complexities, for larger value of 'n' the term '2n + 1' in algorithm 1 has least significance than the term '5n2', and the term '8n + 3' in algorithm 2 has least significance than the term '10n2'.
- Here, for larger value of 'n' the value of most significant terms ( 5n2 and 10n2 ) is very larger than the value of least significant terms ( 2n + 1 and 8n + 3 ).
- So for larger value of 'n' we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

# Asymptotic Notation

***Asymptotic analysis of an algorithm*** **-** refers to defining the mathematical framing of its run-time performance.

Usually, time required by an algorithm falls under three types:

- ✓ Best Case − Minimum time required for program execution.
- ✓ Average Case − Average time required for program execution.
- ✓ Worst Case − Maximum time required for program execution.

Asymptotic analysis are input bound i.e., if there's no input to the algorithm it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.

# Asymptotic Notation

***For example***, running time of one operation is computed as f(n) and may be for another operation it is computed as g(n2). Which means first operation running time will increase linearly with the increase in n and running time of second operation will increase exponentially when n increases.

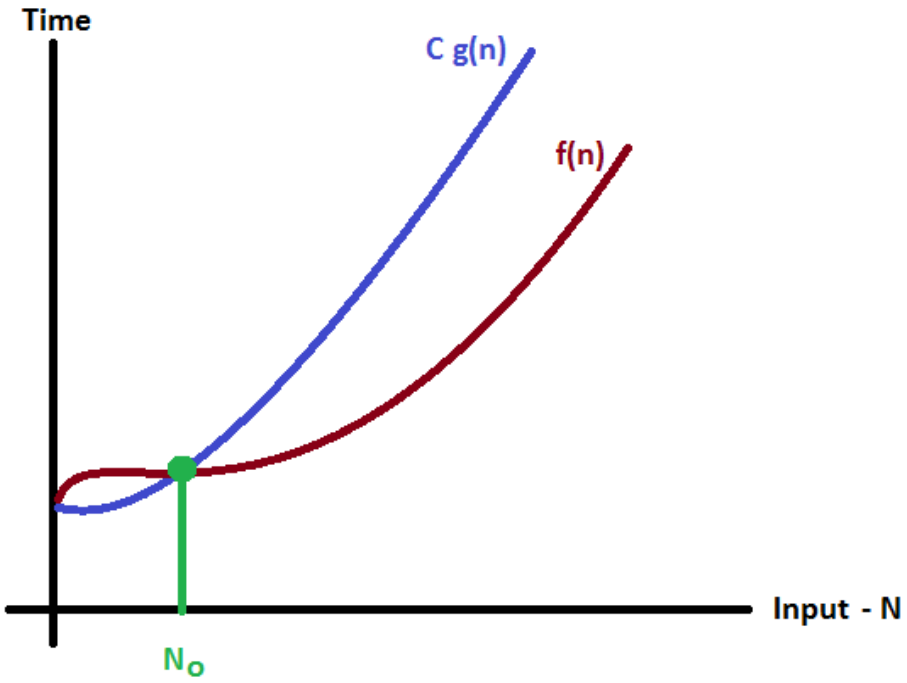
The Big-O Notation

Omega Ω Notation

Theta Θ Notation

- Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity.
- That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.
- Big - Oh Notation can be defined as follows...
- Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n0, C > 0 and n0 >= 1. Then we can represent f(n) as O(g(n)).
- f(n) = O(g(n))

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis




In above graph after a particular input value $n_0$, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

# Asymptotic Notation -The Big-O

## The Big-O

Big oh(O): Definition: f(n) = O(g(n)) (read as f of n is big oh of g of n) if there exist a positive integer n0 and a positive number c such that $|f(n)| \leq c|g(n)|$ for all $n \geq n0$ . Here g(n) is the upper bound of the function f(n).

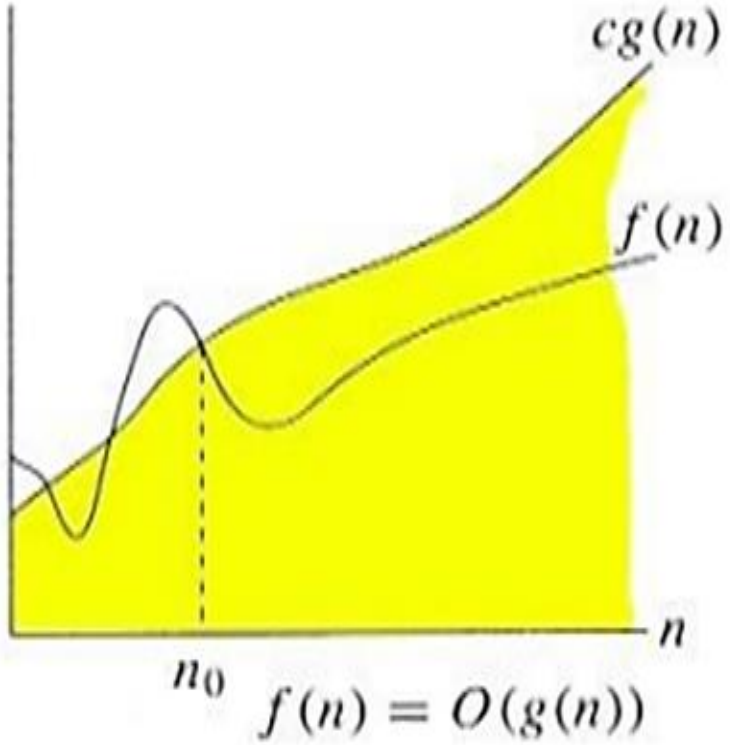
Big "oh": O

- ✓ The asymptotic running time of an algorithm is defined in terms of functions.
- ✓ The upper bound for the function 'f' is provided by the big oh notation (O).
- ✓ Considering 'g' to be a function from the non-negative integers to the positive real numbers, then it define O(g) as the set of function f such that for some real constant c>0 and some non-negative integers constant n0.

# The Big-O Notation

**Example**:

| f(n) | g(n) | |
|---|---|---|
| $16n^3 + 45n^2 + 12n$ | $n^3$ | $f(n) = O(n^3)$ |
| $34n - 40$ | $n$ | $f(n) = O(n)$ |
| 50 | 1 | $f(n) = O(1)$ |



- Example

- Consider the following f(n) and g(n)...
- f(n) = 3n + 2
- g(n) = n
- If we want to represent f(n) as O(g(n)) then it must satisfy f(n) <= C g(n) for all values of C > 0 and n0>= 1
- f(n) <= C g(n)
- ⇒3n + 2 <= C n
- Above condition is always TRUE for all values of C = 4 and n >= 2.
- By using Big - Oh notation we can represent the time complexity as follows...
- 3n + 2 = O(n)

# The Big-O Notation

Time complexity of an algorithm using O Notation:

| Running TIME | NAME |
|---|---|
| **O(1)** | constant |
| **O(n)** | linear |
| **$O(n^2)$** | quadratic |
| **$O(n^3)$** | cubic |
| **$O(2^n)$ ,$O(3^n)$, $O(k^n)$** | Exponential |
| **$O(n^k)$** | Polynomial |
| **O(log n)** | Logarithmic |
| **O(n log n)** | Log linear |

Some of the commonly occurring time complexities in their ascending orders of magnitude are listed as: $O(1) \leq O(\log n) \leq O(n) \leq O(n \log n) \leq O(n^2) \leq O(n^3) \leq O(2^n)$

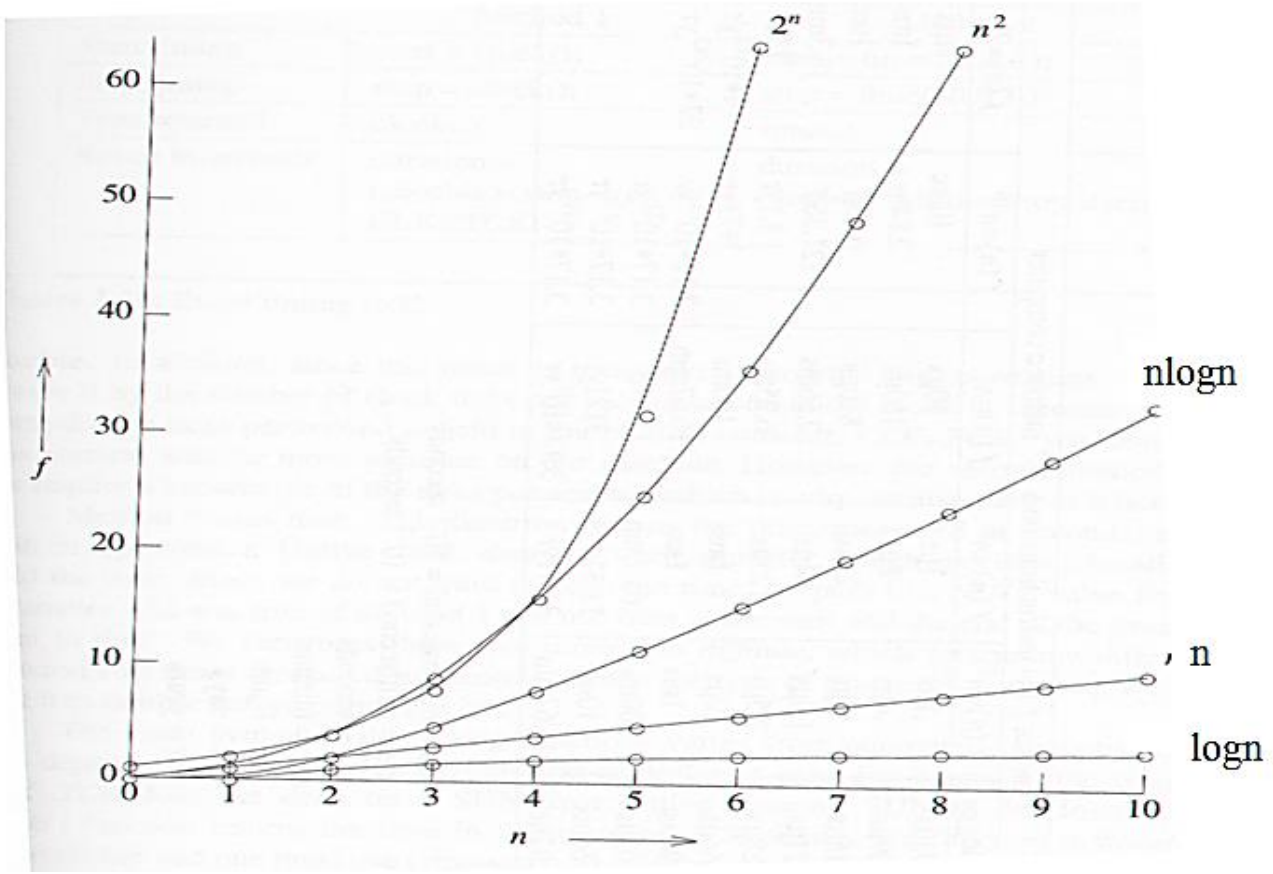
# The Big-O Notation

Let's draw the growth rates for the above functions and take a look at the following table.

| | *constant* | *logarithmic* | *linear* | *N-log-N* | *quadratic* | *cubic* | *exponential* |
|---|---|---|---|---|---|---|---|
| $n$ | $O(1)$ | $O(\log n)$ | $O(n)$ | $O(n \log n)$ | $O(n^2)$ | $O(n^3)$ | $O(2^n)$ |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 2 | 1 | 1 | 2 | 2 | 4 | 8 | 4 |
| 4 | 1 | 2 | 4 | 8 | 16 | 64 | 16 |
| 8 | 1 | 3 | 8 | 24 | 64 | 512 | 256 |
| 16 | 1 | 4 | 16 | 64 | 256 | 4,096 | 65536 |
| 32 | 1 | 5 | 32 | 160 | 1,024 | 32,768 | 4,294,967,296 |
| 64 | 1 | 6 | 64 | 384 | 4,069 | 262,144 | $1.84 \times 10^{19}$ |

# The Big-O Notation

Plot of function values



# Omega Ω Notation

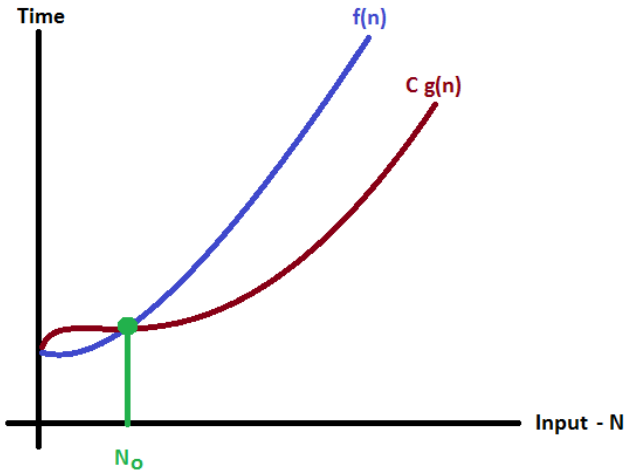
Omega: Ω

- ✓ The lower bound for the function ‘f’ is provided by the big omega notation (Ω).
- ✓ Considering ‘g’ to be a function from the non-negative integers to the positive real numbers, we define Ω(g) as the set of function f such that for some real constant c>0 and some non-negative integers constant n0.

f(n) = Ω (g(n)) iff there exist positive constants c and $n_0$ such that $f(n) \geq cg(n)$ for all n, $n \geq n_0$.

- Big - Omege Notation (Ω)
- Big - Omega notation is used to define the lower bound of an algorithm in terms of Time Complexity.
- That means Big-Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big-Omega notation describes the best case of an algorithm time complexity.
- Big - Omega Notation can be defined as follows...
- Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n0, C > 0 and n0 >= 1. Then we can represent f(n) as Ω(g(n)).
    - f(n) = Ω(g(n))

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis

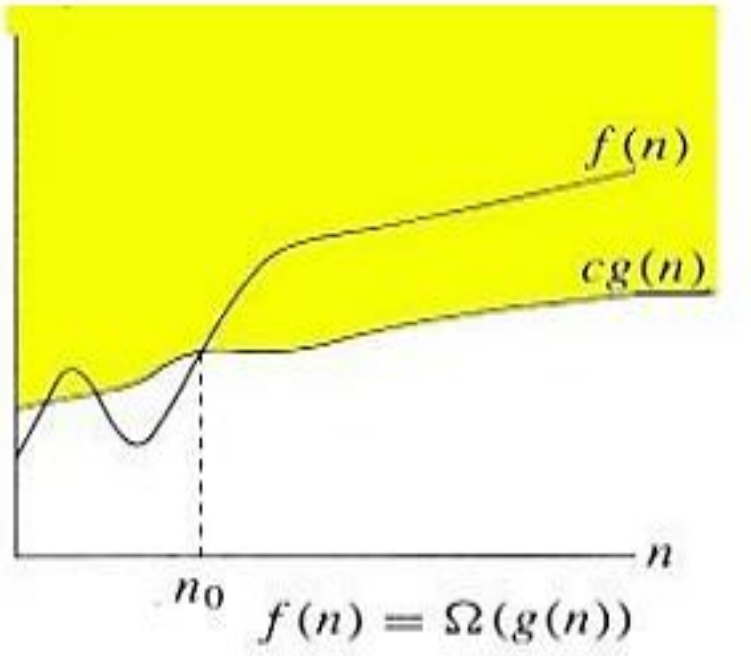



In above graph after a particular input value $n_0$, always C g(n) is less than f(n) which indicates the algorithm's lower bound.

# Omega Ω Notation

**Example:**

| f(n) | g(n) | |
|---|---|---|
| $16n^2 + 8n + 2$ | $n^2$ | $f(n) = \Omega(n^2)$ |
| $24n + 9$ | $n$ | $f(n) = \Omega(n)$ |



- Example

- Consider the following f(n) and g(n)...
- f(n) = 3n + 2
- g(n) = n
- If we want to represent f(n) as Ω(g(n)) then it must satisfy f(n) >= C g(n) for all values of C > 0 and n0>= 1
- f(n) >= C g(n)
- ⇒3n + 2 >= C n
- Above condition is always TRUE for all values of C = 1 and n >= 1.
- By using Big - Omega notation we can represent the time complexity as follows...
- 3n + 2 = Ω(n)

# Theta Θ Notation

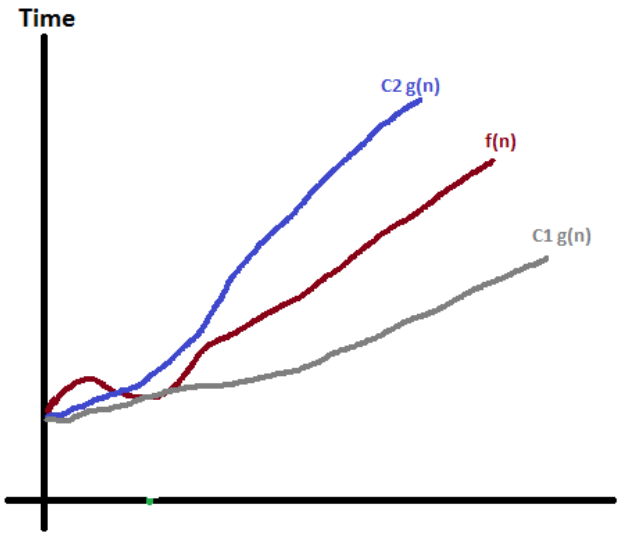
Definition: f(n) = Θ(g(n)) (read as f of n is theta of g of n), if there exists a positive integer n0 and two positive constants c1 and c2 such that c1 |g(n)| ≤ |f(n)| ≤ c2 |g(n)| for all n ≥ n0. The function g(n) is both an upper bound and a lower bound for the function f(n) for all values of n, n ≥ n0

Theta: Θ

- ✓ The upper and lower bound for the function ‘f’ is provided by the big oh notation (Θ). Considering ‘g’ to be a function from the non-negative integers to the positive real numbers, we define Θ(g) as the set of function f such that for some real constant c1 and c2 >0 and some non negative integers constant n0.

- Big - Theta notation is used to define the average bound of an algorithm in terms of Time Complexity.
- That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.
- Big - Theta Notation can be defined as follows...
- Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C1 g(n) <= f(n) <= C2 g(n) for all n >= n0, C1 > 0, C2 > 0 and n0 >= 1. Then we can represent f(n) as Θ(g(n)).
    - **f(n) = Θ(g(n))**

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis

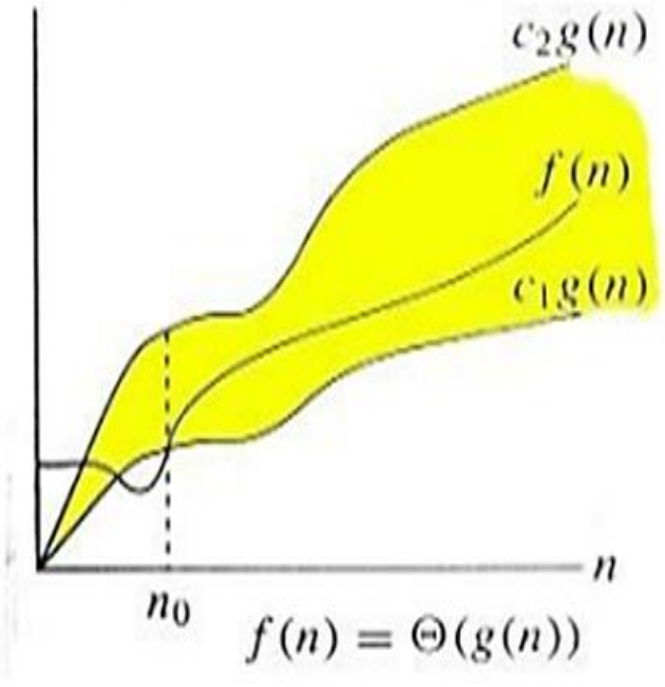



In above graph after a particular input value n0, always C1 g(n) is less than f(n) and C2 g(n) is greater than f(n) which indicates the algorithm's average bound.

# Theta Θ Notation

**Example:**

| f(n) | g(n) | |
|---|---|---|
| $16n^3 + 30n^2 - 90$ | $n^2$ | $f(n) = \Theta(n^2)$ |
| $7.2^n + 30n$ | $2^n$ | $f(n) = \Theta(2^n)$ |



- Example

- Consider the following f(n) and g(n)...
- f(n) = 3n + 2
- g(n) = n
- If we want to represent f(n) as Θ(g(n)) then it must satisfy C1 g(n) <= f(n) <= C2 g(n) for all values of C1 > 0, C2 > 0 and n0>= 1
- C1 g(n) <= f(n) <= C2 g(n)
- ⇒C1 n <= 3n + 2 <= C2 n
- Above condition is always TRUE for all values of C1 = 1, C2 = 4 and n >= 2.
- By using Big - Theta notation we can represent the time compexity as follows...
- 3n + 2 = Θ(n)

# Testing:-

- Technically Testing is a process to check if the application is working same as it was supposed to do, and not working as it was not supposed to do.
- Main objective of Testing is to find bugs and errors in an application which get missed during the unit testing by the developer.
- A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output.
- A unit may be an individual program, function, procedure, etc

- **Benefits:**

- Unit testing increases confidence in changing / maintaining code. If good unit tests are written and if they are run every time any code is changed, we will be able to promptly catch any defects introduced due to the change. Also, if codes are already made less interdependent to make unit testing possible, the unintended impact of changes to any code is less.
- Codes are more reusable. In order to make unit testing possible, codes need to be modular. This means that codes are easier to reuse.
- Debugging is easy. When a test fails, only the latest changes need to be debugged.

# Time-space Trade Off in algorithm

**Definition:**

In computer science, a space-time or time-memory tradeoff is a way of solving a problem in :

1.) Either in less time and by using more space

or

2.) By solving a problem in very little space by spending a long time.

# Time-space Trade Off

**Types of Trade Off:**

1. Compressed / Uncompressed Data
2. Re-Rendering / Stored Images
3. Smaller Code / Loop Unrolling
4. Lookup Table / Recalculation

# Types of Trade Off
# 1. Compressed / Uncompressed Data

- A space–time tradeoff can be applied to the problem of data storage.
  - If data is stored uncompressed, it takes more space but access takes less time than if the data were stored compressed
  - since compressing the data reduces the amount of space it takes, but it takes time to run the decompression algorithm
  - Depending on the particular instance of the problem, either way is practical.
  - There are also rare instances where it is possible to directly work with compressed data, such as in the case of compressed bitmap indices, where it is faster to work with compression than without compression.

# Types of Trade Off

2. **Re-Rendering / Stored Images**

- Storing only the SVG source of a vector image and rendering it as a bitmap image every time the page is requested would be trading time for space; more time used, but less space.
- Rendering the image when the page is changed and storing the rendered images would be trading space for time; more space used, but less time. This technique is more generally known as caching.

# Types of Trade Off

3. **Smaller Code(with loop) / Larger Code (without loop)**

- Smaller code occupies ***less space*** in memory but it requires ***high computation time*** which is required for jumping back to the beginning of the loop at the end of each iteration.
- Larger code or Loop unrolling can optimize execution speed at the cost of increased binary size. It occupies ***more space*** in memory but requires ***less computation time***. (as we need not perform jumps back and forth since there is no loop.)

- Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program.
- We basically remove or reduce iterations.
- Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

- // This program does not uses loop unrolling.

```
#include<stdio.h>

int main(void)
{
        for (int i=0; i<5; i++)
                printf("Hello\n"); //print hello 5 times

        return 0;
}
```

- // This program uses loop unrolling.

```
#include<stdio.h>

int main(void)
{
   // unrolled the for loop in program 1
   printf("Hello\n");
   printf("Hello\n");
   printf("Hello\n");
   printf("Hello\n");
   printf("Hello\n");

   return 0;
}
```

Output:

```
Hello
Hello
Hello
Hello
Hello
```

- Program 2 is more efficient than program 1 because in program 1 there is a need to check the value of i and increment the value of i every time round the loop.
- So small loops like this or loops where there is fixed number of iterations are involved can be unrolled completely to reduce the loop overhead.

- Advantages
  - Increases program efficiency.
  - Reduces loop overhead.
  - If statements in loop are not dependent on each other, they can be executed in parallel.
- Disadvantages
  - Increased program code size, which can be undesirable.
  - Possible increased usage of register in a single iteration to store temporary variables which may reduce performance.
  - Apart from very small and simple codes, unrolled loops that contain branches are even slower than recursions.

# Types of Trade Off

**4. Lookup Table / Recalculation**

- In lookup table, an implementation can include the entire table which ***reduces computing time*** but ***increases*** the amount of ***memory needed***.
- It can recalculate i.e. compute table entries as needed, ***increasing computing time*** but ***reducing memory requirements***.

- A lookup table or lookup file holds static data and is used to look up a secondary value based on a primary value.

- It can be used to translate encoded information into a user-friendly description, to validate input values by matching them to a list of valid items, or to translate a shorthand entry into something more detailed.

- A lookup action retrieves values from a related table. Specify the starting item, where to locate its value, and the column whose value is to be retrieved.

- For example, using the following table, if you want to display the name of a sales manager for a particular sales team, specify the name of the sales team as the primary value, for example, Rhode Island, and the lookup table returns the name of the sales manager, Mike Lucas, as the secondary value.

- Typically, a lookup table is one of the following file types:

- Text file (*.txt)
- Lookup table (*.tbl)
- Tab delimited file (*.tab)
- Lookup table (*.lup)

# Types of Trade Off

**Example**

| More time, Less space | More Space, Less time |
|---|---|
| 1.int a,b;<br>2. printf(“enter value of a\ n”);<br>3. scanf(“%d”,&a);<br>4. printf(“enter value of b \n”);<br>5. scanf(“%d”,&b);<br>6. b=a+b;<br>7. printf(“output is:%d”,b); | 1. int a,b,c;<br>2.printf(“enter values of a,b and c\ n”);<br>3. scanf(“%d%d%d”,&a,&b,&c);<br>4. printf(“output is:%d”,c=a+b); |

# Recursive Algorithms

- A recursive algorithm calls itself which generally passes the return value as a parameter to the algorithm again. This parameter indicates the input while the return value indicates the output.
- Recursive algorithm is defined as a method of simplification that divides the problem into sub-problems of the same nature.
- The result of one recursion is treated as the input for the next recursion. The repletion is in the self-similar fashion manner.
- The algorithm calls itself with smaller input values and obtains the results by simply accomplishing the operations on these smaller values.

- ***Examples of recursive algorithms***

  Generation of factorial

  Fibonacci number series

*Example: Writing factorial function using recursion*

```
intfactorialA(int n)
{
        return n * factorialA(n-1);
}
```

# Refinement of coding

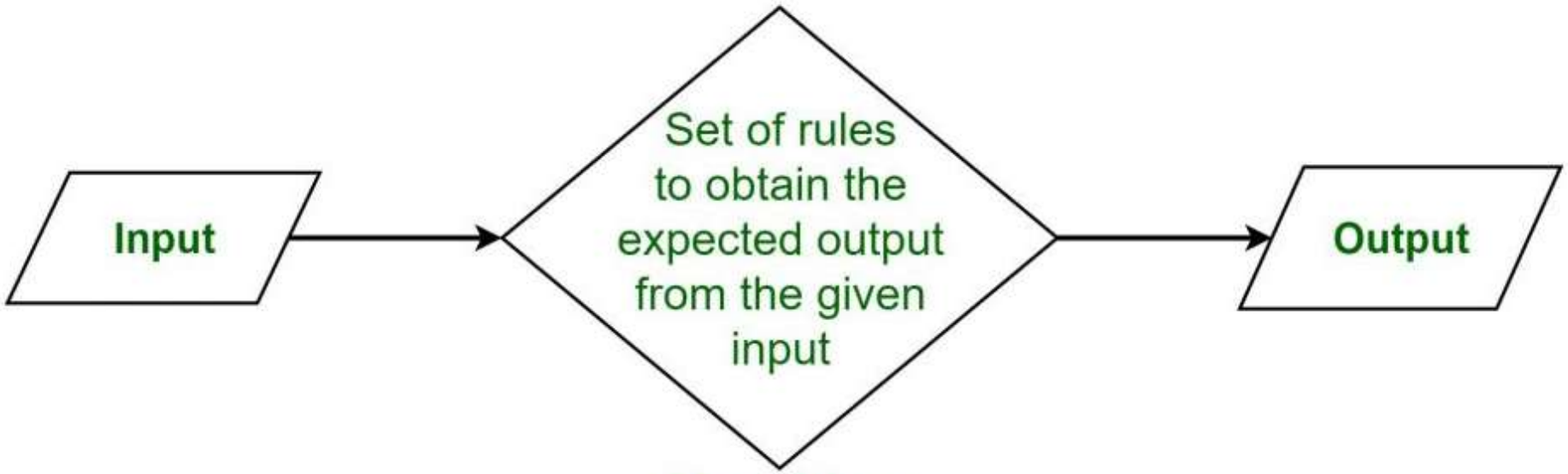
- Stepwise refinement is the idea that software is developed by moving through the levels of abstraction, beginning at higher levels and, incrementally refining the software through each level of abstraction, providing more detail at each increment.
- At higher levels, the software is merely its design models; at lower levels there will be some code; at the lowest level the software has been completely developed.

- At the early steps of the refinement process the software engineer does not necessarily know how the software will perform what it needs to do. This is determined at each successive refinement step, as the design and the software is elaborated upon.

- Refinement can be seen as the compliment of abstraction.
- Abstraction is concerned with hiding lower levels of detail; it moves from lower to higher levels.
- Refinement is the movement from higher levels of detail to lower levels. Both concepts are necessary in developing software.

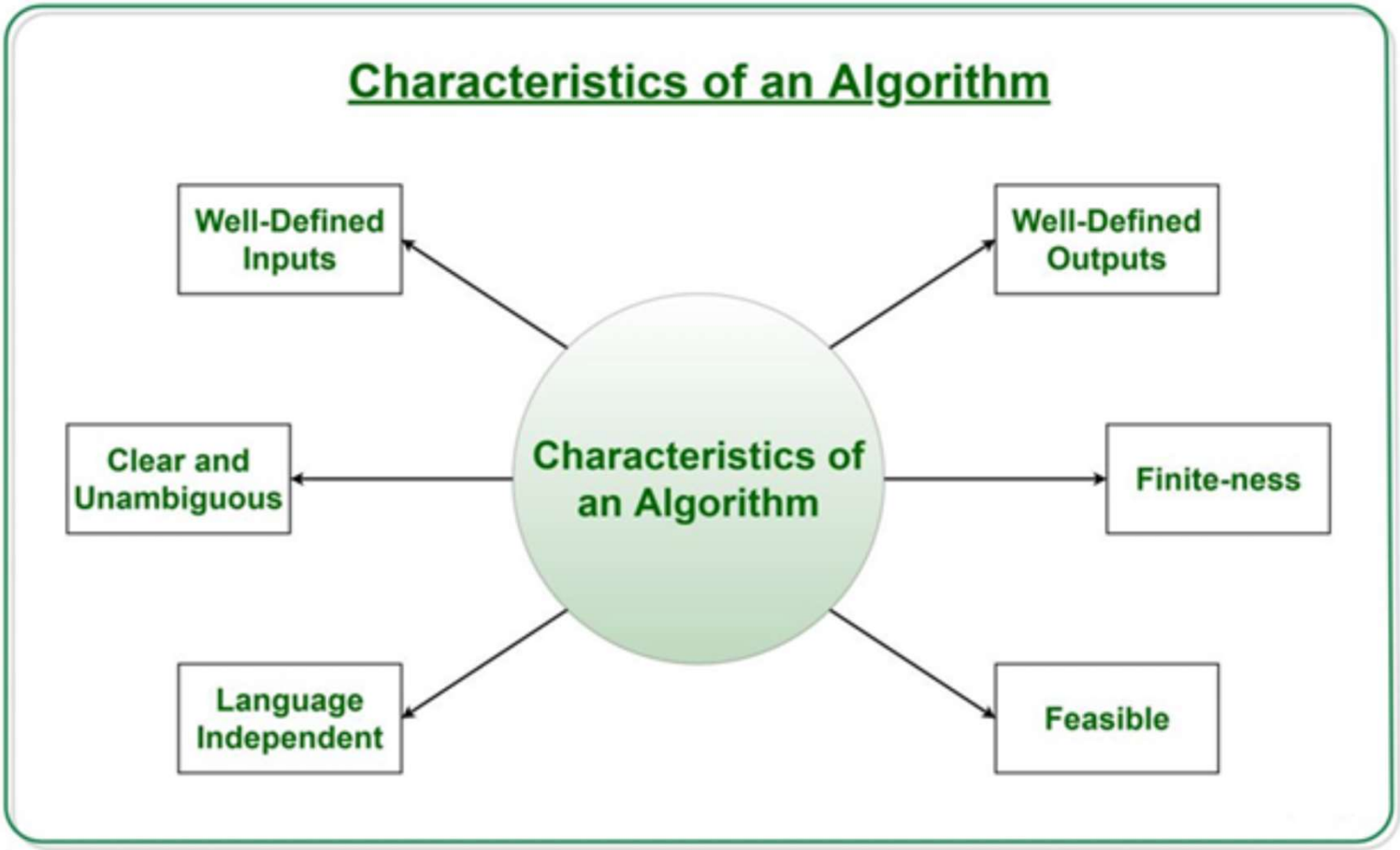
- Algorithm specification
- An algorithm is defined as a finite set of instructions that, if followed, performs a particular task.
- Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.

# What is Algorithm?



All algorithms must satisfy the following criteria

- Input - An algorithm has zero or more inputs, taken or collected from a specified set of objects.
- Output - An algorithm has one or more outputs having a specific relation to the inputs.
- Definiteness - Each step must be clearly defined; each instruction must be clear and unambiguous.
- Finiteness - The algorithm must always finish or terminate after a finite number of steps.
- Effectiveness - All operations to be accomplished must be sufficiently basic that they can be done exactly and in finite length.


Characteristics of an Algorithm
Well-Defined Inputs
Well-Defined Outputs
Clear and Unambiguous
Characteristics of an Algorithm
Finite-ness
Language Independent
Feasible

- Clear and Unambiguous: Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.

- Well-Defined Inputs: If an algorithm says to take inputs, it should be well-defined inputs.

- Well-Defined Outputs: The algorithm must clearly define what output will be yielded and it should be well-defined as well.

- Finite-ness: The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.

- Feasible: The algorithm must be simple, generic and practical, such that it can be executed upon will the available resources. It must not contain some future technology, or anything.

- Language Independent: The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

- Advantages of Algorithms
  - It is easy to understand.
  - Algorithm is a step-wise representation of a solution to a given problem.
  - In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

- Disadvantages of Algorithms
  - Writing an algorithm takes a long time so it is time-consuming.
  - Branching and Looping statements are difficult to show in Algorithms.

- We can depict an algorithm in many ways.
- Natural language: implement a natural language like English
- Flow charts: Graphic representations denoted flowcharts, only if the algorithm is small and simple.
- Pseudo code: This pseudo code skips most issues of ambiguity; no particularity regarding syntax programming language.

# Performance Analysis of an algorithm

- If we want to go from city "A" to city "B", there can be many ways of doing this.
- We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us.
- Similarly, in computer science, there are multiple algorithms to solve a problem.
- When we have more than one algorithm to solve a problem, we need to select the best one.
- Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.
- When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements. The formal definition is as follows...

- **Performance of an algorithm is a process of making evaluative judgement about algorithms.**
- It can also be defined as follows...
- **Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.**

- Generally, the performance of an algorithm depends on the following elements...
  - Whether that algorithm is providing the exact solution for the problem?
  - Whether it is easy to understand?
  - Whether it is easy to implement?
  - How much space (memory) it requires to solve the problem?
  - How much time it takes to solve the problem? Etc.,

- Analysis of efficiency of an algorithm can be performed at two different stages, before implementation and after implementation, as
- **A priori analysis** −
  - This is defined as theoretical analysis of an algorithm.
  - Efficiency of algorithm is measured by assuming that all other factors e.g. speed of processor, are constant and have no effect on implementation.
- **A posterior analysis** −
  - This is defined as empirical analysis of an algorithm. The chosen algorithm is implemented using programming language.
  - Next the chosen algorithm is executed on target computer machine. In this analysis, actual statistics like running time and space needed are collected.
- Algorithm analysis is dealt with the execution or running time of various operations involved. Running time of an operation can be defined as number of computer instructions executed per operation.

# Programming Style

- Programming style refers to the technique used in writing the source code for a computer program.
- Most programming styles are designed to help programmers quickly read and understands the program as well as avoid making errors. (Older programming styles also focused on conserving screen space.)
- A good coding style can overcome the many deficiencies of a first programming language, while poor style can defeat the intent of an excellent language.

- The goal of good programming style is to provide understandable, straightforward, elegant code.
- The programming style used in a various program may be derived from the coding standards or code conventions of a company or other computing organization, as well as the preferences of the actual programmer.

- Example:
- To find the average of 3 numbers, the algorithm is as shown below.
- Step1: Read the numbers a, b, c, and d.
- Step2: Compute the sum of a, b, and c.
- Step3: Divide the sum by 3.
- Step4: Store the result in variable of d.
- Step5: End the program.

- Development of an Algorithm
- The steps involved in the development of an algorithm are as follows:
  - Specifying the problem statement.
  - Designing an algorithm.
  - Coding.
  - Debugging
  - Testing and Validating
  - Documentation and Maintenance.

- Specifying the problem statement: The problem which has to be implemented in to a program must be thoroughly understood before the program is written.
- Problem must be analyzed to determine the input and output requirements of the program.
- Designing an Algorithm: Once the problem is cleared then a solution method for solving the problem has to be analyzed.
- There may be several methods available for obtaining the required solution.
- The best suitable method is designing an Algorithm.
- To improve the clarity and understandability of the program flowcharts are drawn using algorithms.

- Coding: The actual program is written in the required programming language with the help of information depicted in flowcharts and algorithms.
- Debugging: There is a possibility of occurrence of errors in program. These errors must be removed for proper working of programs. The process of checking the errors in the program is known as Debugging‘.

- There are three types of errors in the program.
- Syntactic Errors: They occur due to wrong usage of syntax for the statements.
- Ex: x=a*%b
- Here two operators are used in between two operands.
- Runtime Errors : They are determined at the execution time of the program
- Ex: Divide by zero
- Range out of bounds.
- Logical Errors: They occur due to incorrect usage of instructions in the program. They are neither displayed during compilation or execution nor cause any obstruction to the program execution. They only cause incorrect outputs.

- Testing and Validating: Once the program is written, it must be tested and then validated.i.e., to check whether the program is producing correct results or not for different values of input
- Documentation and Maintenance: Documentation is the process of collecting, organizing and maintaining, in written the complete information of the program for future references.
- Maintenance is the process of upgrading the program, according to the changing requirements.

# General rules or guidelines in respect of programming style

1. Clarity and simplicity of Expression: The programs should be designed in such a manner so that the objectives of the program is clear.

**2. Naming:** In a program, you are required to name the module, processes, and variable, and so on. Care should be taken that the naming style should not be cryptic and non-representative.

**For Example:** a = 3.14 * r * r

area of circle = 3.14 * radius * radius;

3. Control Constructs: It is desirable that as much as a possible single entry and single exit constructs used.

4. Information hiding: The information secure in the data structures should be hidden from the rest of the system where possible. Information hiding can decrease the coupling between modules and make the system more maintainable.

5. Nesting: Deep nesting of loops and conditions greatly harm the static and dynamic behavior of a program. It also becomes difficult to understand the program logic, so it is desirable to avoid deep nesting.

6. User-defined types: Make heavy use of user-defined data types like enum, class, structure, and union. These data types make your program code easy to write and easy to understand.

7. Module size: The module size should be uniform. The size of the module should not be too big or too small. If the module size is too large, it is not generally functionally cohesive. If the module size is too small, it leads to unnecessary overheads.

8. Module Interface: A module with a complex interface should be carefully examined.

9. Side-effects: When a module is invoked, it sometimes has a side effect of modifying the program state. Such side-effect should be avoided where as possible.

# Data Abstraction

- An ADT is a mathematical model of a data structure that specifies the type of **data** stored, the **operations** supported on them, and the types of **parameters of the operations**.
- An ADT specifies what each operation does, but not how it does it.
- Typically, an ADT can be implemented using one of many different data structures.
- A useful first step in deciding what data structure to use in a program is to specify an ADT for the program.

- In general, the steps of building ADT to data structures are:
  - Understand and clarify the nature of the target information unit.
  - Identify and determine which data objects and operations to include in the models.
  - Express this property somewhat formally so that it can be understood and communicate well.
  - Translate this formal specification into proper language. In C++, this becomes a .h file. In Java, this is called "user interface".
  - Upon finalized specification, write necessary implementation. This includes storage scheme and operational detail. Operational detail is expressed as separate functions (methods).

- Consider two alternate data structures for the above ADT:
- (1) **an unordered array** of records and
- (2) **an ordered array** of records, ordered by IDs. These different data structures greatly influence the implementation details and how fast and efficient the program runs.
- **unordered** array- Assume that the student records are stored in an array with no particular order. We can use a variable *n* to keep track of the number of students currently in the array.
  - **ADD**: Simply take the record and store it in slot *n* of the array and increment *n*. This takes constant amount of time since the time is independent of *n*.

- **SEARCH**: Since the array is not ordered, we have to scan through the whole array to find the requested record.
- The result could be either the record is found or the record doesn't exist. The time taken to perform the search is proportional to *n* and the worst case scenario is *n*.
- if the record I am looking for happens to be the first item in the array, it only takes constant time. However, when determining the running time of an algorithm, we are often interested in worst case analysis.

- **DELETE**: This operation requires us to first search for the given record.
- Once it is found, the algorithm can simply replace it by the last record and decrement *n*.
- Once the record is found, it only takes a constant amount of time to delete it. But time spent searching for the record is the same as above, proportional to *n*. Therefore, in all, this operation also takes time proportional to *n*, in the worst case.

- **ordered** array - Assume that the student records are sorted in an array, with an increasing order of student IDs. We can also use a variable *n* to keep track of the number of students currently in the array.
- **ADD**: Since the array is sorted, we first need to find out where should we insert the record. This can be done by scanning through the array, comparing the current record in the array with the record we want to insert, and finding the smallest index *i* of the record whose ID is larger than the new ID.
- Then the new record should be put to the *i*th slot in the array. To do that, all the records in slots *i*, *i*+1, *i*+2, ..., *n* need to be moved down one slot to create an empty slot for the new record. We can then put the new record into the *i*th slot and increment *n*. This operation takes time proportional to *n*.

- **SEARCH**: Since the array of records is sorted by IDs, we can use a **binary search** to find the given record.
- In general, a binary search algorithm first compares the given ID with the ID in the middle of the array (with index $n/2$ or $(n+1)/2$).
- Then the algorithm branches based on different conditions: if the two IDs are the same, we find the record; if the given ID is smaller, the algorithm continues to search the first half of the current array, ignoring the second half; otherwise, the algorithm continues to search the second half of the current array, ignoring the first half.
- The operation time is $\log_2 n$. Finding the time for binary search will be discussed in the future.

- **DELETE**: This operation requires us to first search for the given record. This can be done in $\log_2 n$ time as shown above. Since the array is ordered, we need to fill in the empty slot $i$ with a record, which means the records in slots $i$+1, $i$+2, ..., $n$ need to be moved up one slot to cover the empty one. The time is also proportional to $n$. In all, this operation takes time proportional to $n$, in the worst case.

- Some examples of ADT are Stack, Queue, List etc.
- operations of those above ADT
- Stack –
  - isFull(), This is used to check whether stack is full or not
  - isEmpry(), This is used to check whether stack is empty or not
  - push(x), This is used to push x into the stack
  - pop(), This is used to delete one element from top of the stack
  - peek(), This is used to get the top most element of the stack
  - size(), this function is used to get number of elements present into the stack

- Queue –
  - isFull(), This is used to check whether queue is full or not
  - isEmpty(), This is used to check whether queue is empty or not
  - insert(x), This is used to add x into the queue at the rear end
  - delete(), This is used to delete one element from the front end of the queue
  - size(), this function is used to get number of elements present into the queue

- List
  - size(), this function is used to get number of elements present into the list
  - insert(x), this function is used to insert one element into the list
  - remove(x), this function is used to remove given element from the list
  - get(i), this function is used to get element at position i
  - replace(x, y), this function is used to replace x with y value