MODULE 1

Introduction to programming methodologies

Programming methodology deals with the analysis, design and implementation of programs.

Algorithm

Algorithm is a step-by-step finite sequence of instruction, to solve a well-defined computational problem.

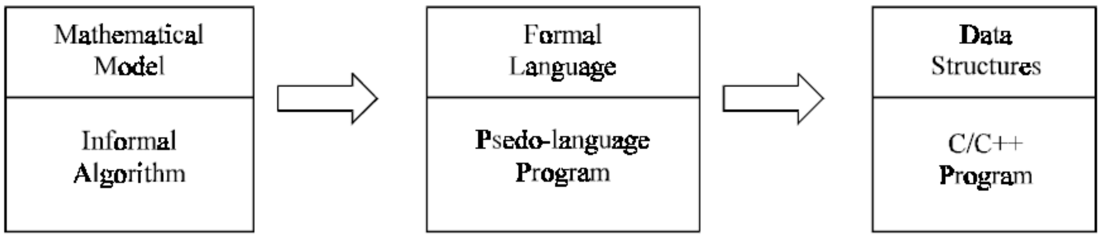
That is, in practice to solve any complex real life problems; first we have to define the problems. Second step is to design the algorithm to solve that problem.

Writing and executing programs and then optimizing them may be effective for small programs. Optimization of a program is directly concerned with algorithm design. But for a large program, each part of the program must be well organized before writing the program. There are few steps of refinement involved when a problem is converted to program; this method is called ***stepwise refinement method***. There are two approaches for algorithm design; they are ***top-down*** and ***bottom-up*** algorithm design.

Stepwise Refinement Techniques

We can write an informal algorithm, if we have an appropriate mathematical model for a problem. The initial version of the algorithm will contain general statements, *i.e.*, informal instructions. Then we convert this informal algorithm to formal algorithm, that is, more definite instructions by applying any programming language syntax and semantics partially. Finally a program can be developed by converting the formal algorithm by a programming language manual. From the above discussion we have understood that there are several steps to reach a program from a mathematical model. In every step there is a refinement (or conversion).
That is to convert an informal algorithm to a program, we must go through several stages of formalization until we arrive at a program — whose meaning is formally defined by a programming language manual — is called stepwise refinement techniques.

There are three steps in refinement process, which is illustrated in Figure



1. In the first stage, modeling, we try to represent the problem using an appropriate mathematical model such as a graph, tree etc. At this stage, the solution to the problem is an algorithm expressed very informally.
2. At the next stage, the algorithm is written in pseudo-language (or formal algorithm) that is, a mixture of any programming language constructs and less formal English statements. The operations to be performed on the various types of data become fixed.
3. In the final stage we choose an implementation for each abstract data type and write the procedures for the various operations on that type. The remaining informal statements in the pseudo-language algorithm are replaced by (or any programming language) C/C++ code.

Programming style

Following sections will discuss different programming methodologies to design a program.

1. Procedural
2. Modular
3. Structured
4. Object oriented

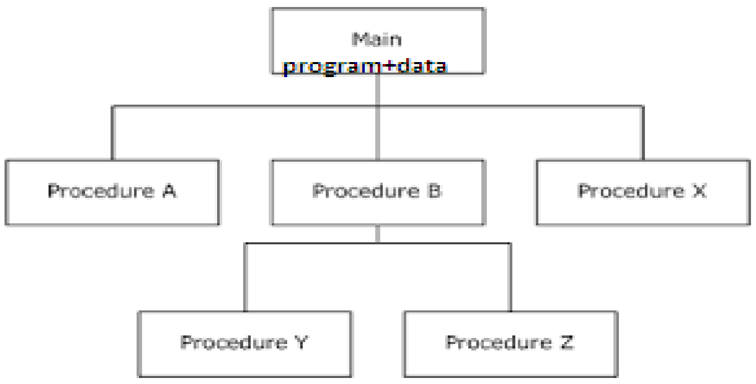
1.Procedural Programming

Procedural programming is a paradigm based on the concept of using procedures. Procedure (sometimes also called subprogram, routine or method) is a sequence of commands to be executed. Any procedure can be called from any point within the general program, including other procedures or even itself (resulting in a recursion).

Procedural programming is widely used in large-scale projects, when the following benefits are important:

- re-usability of pieces code designed as procedures
- ease of following the logic of program;
- Maintainability of code.
- Emphasis is on doing things (algorithms).
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.

Procedural programming is a sub-paradigm of imperative programming, since each step of computation is described explicitly, even if by the means of defining procedures.



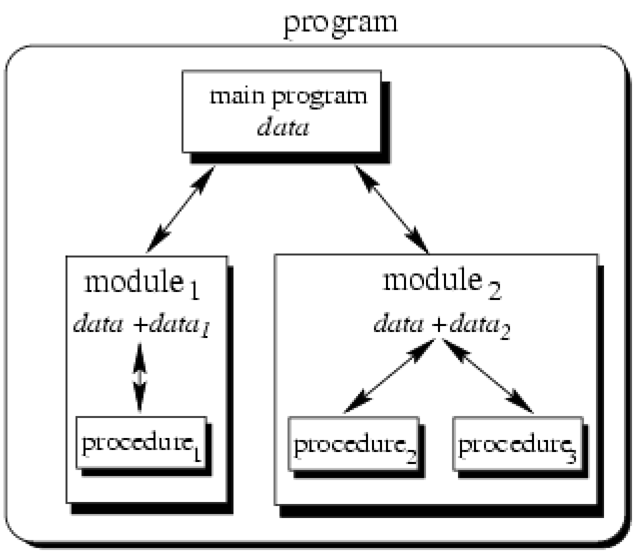
2.Modular Programming

The program is progressively decomposed into smaller partition called modules. The program can easily be written in modular form, thus allowing an overall problem to be decomposed into a sequence of individual sub programs. Thus we can consider, a module decomposed into successively subordinate module. Conversely, a number of modules can combined together to form a superior module.

A sub-module, are located elsewhere in the program and the superior module, whenever necessary make a reference to subordinate module and call for its execution. This activity on part of the superior module is known as a calling, and this module is referred as calling module, and sub module referred as called module. The sub module may be subprograms such as function or procedures.

The following are the steps needed to develop a modular program

1. Define the problem
2. Outline the steps needed to achieve the goal
3. Decompose the problem into subtasks
4. Prototype a subprogram for each sub task
5. Repeat step 3 and 4 for each subprogram until further decomposition seems counter productive



Modular Programming is heavily procedural. The focus is entirely on writing code (functions). Data is passive in Modular Programming. Modular Programming discourages the use of control variables and flags in parameters; their presence tends to indicate that the caller needs to know too much about how the function is implemented.

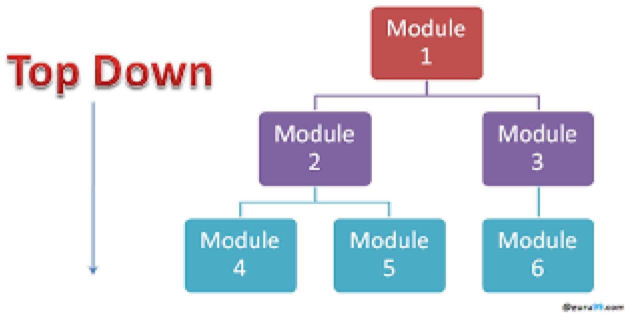
Two methods may be used for modular programming. They are known as **top-down and bottom-up**. Regardless of whether the top-down or bottom-up method is used, the end result is a modular program. This end result is important, because not all errors may be detected at the time of the initial testing. It is possible that there are still bugs in the program. If an error is discovered after the program supposedly has been fully tested, then the modules concerned can be isolated and retested by them. Regardless of the design method used, if a program has been written in modular form, it is easier to detect the source of the error and to test it in isolation, than if the program were written as one function.

Advantages of modular programming

1. Reduce the complexity of the entire problem
2. Avoid the duplication of code
3. debugging program is easier and reliable
4. Improves the performance
5. Modular program hides the use of data structure
6. Global data also hidden in module
7. Reusability- modules can be used in other program without rewriting and retesting
8. Modular program improves the portability of program
9. It reduces the development work

Top- down modular programming

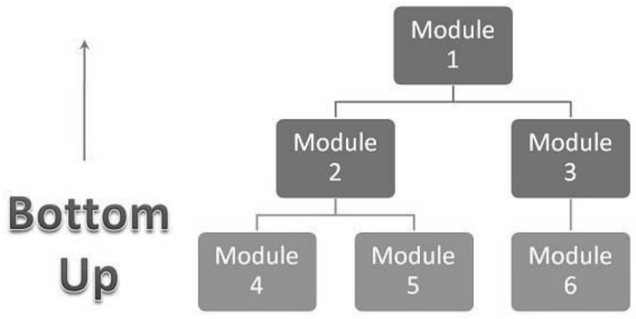
The principles of top-down design dictate that a program should be divided into a main module and its related modules. Each module should also be divided into sub modules according to software engineering and programming style. The division of modules processes until the module consists only of elementary process that are intrinsically understood and cannot be further subdivided.



Top-down algorithm design is a technique for organizing and coding programs in which a hierarchy of modules is used, and breaking the specification down into simpler and simpler pieces, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. That is top-down programming tends to generate modules that are based on functionality, usually in the form of functions or procedures or methods.

Bottom-Up modular programming

Bottom-up algorithm design is the opposite of top-down design. It refers to a style of programming where an application is constructed starting with existing primitives of the programming language, and constructing gradually more and more complicated features, until the all of the application has been written. That is, starting the design with specific modules and build them into more complex structures, ending at the top. The bottom-up method is widely used for testing, because each of the lowest-level functions is written and tested first. This testing is done by special test functions that call the low-level functions, providing them with different parameters and examining the results for correctness. Once lowest-level functions have been tested and verified to be correct, the next level of functions may be tested. Since the lowest-level functions already have been tested, any detected errors are probably due to the higher-level functions. This process continues, moving up the levels, until finally the *main* function is tested.



3.Structured Programming

It is a programming style; and this style of programming is known by several names: Procedural decomposition, Structured programming, etc. Structured programming is not programming with structures but by using following types of code structures to write programs:

1. Sequence of sequentially executed statements
2. Conditional execution of statements (*i.e.*, "if" statements)
3. Looping or iteration (*i.e.*, "for, do...while, and while" statements)
4. Structured subroutine calls (*i.e.*, functions)

In particular, the following language usage is forbidden:

• "GoTo" statements

• "Break" or "continue" out of the middle of loops

• Multiple exit points to a function/procedure/subroutine (*i.e.*, multiple "return" statements)

• Multiple entry points to a function/procedure/subroutine/method

In this style of programming there is a great risk that implementation details of many data structures have to be shared between functions, and thus globally exposed. This in turn tempts other functions to use these implementation details; thereby creating unwanted dependencies between different parts of the program. The main disadvantage is that all decisions made from the start of the project depend directly or indirectly on the high-level specification of the application. It is a well known fact that this specification tends to change over a time. When that happens, there is a great risk that large parts of the application need to be rewritten.

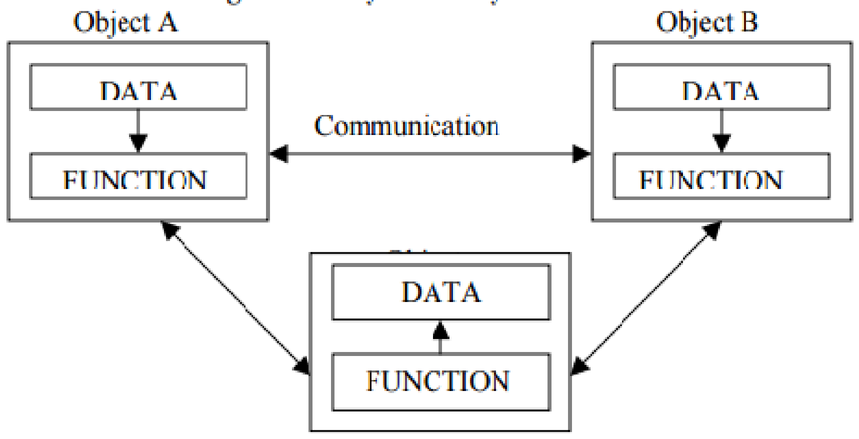
Advantages of structured programming

1. clarity: structured programming has a clarity and logical pattern to their control structure and due to this tremendous increase in programming productivity
2. another key to structured programming is that each block of code has a single entry point and single exit point.so we can break up long sequence of code into modules
3. Maintenance: the clarity and modularity inherent in structured programming is of great help in finding an error and redesigning the required section of code.

4.Obect oriented programming

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural or modular approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function. OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects. The organization of data and function in object-oriented programs is shown in fig. The data of an object can be accessed only by the function associated with that object. However, function of one object can access the function of other objects.

Organization of data and function in OOP



Some of the features of object oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are ties together in the data structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows bottom up approach in program design.

Analysis of Algorithm

After designing an algorithm, it has to be checked and its correctness needs to be predicted; this is done by analyzing the algorithm. The algorithm can be analyzed by tracing all step-by-step instructions, reading the algorithm for logical correctness, and testing it on some data using mathematical techniques to prove it correct. Another type of analysis is to analyze the simplicity of the algorithm. That is, design the algorithm in a simple way so that it becomes easier to be implemented. However, the simplest and most Straight forward way of solving a problem may not be sometimes the best one. Moreover there may be more than one algorithm to solve a problem. The choice of a particular algorithm depends on following performance analysis and measurements:

1. Space complexity
2. Time complexity

Space Complexity

Analysis of space complexity of an algorithm or program is the amount of memory it needs to run to completion. Some of the reasons for studying space complexity are:

1. If the program is to run on multi user system, it may be required to specify the amount of memory to be allocated to the program.

2. We may be interested to know in advance that whether sufficient memory is available to run the program.

3. There may be several possible solutions with different space requirements.

4. Can be used to estimate the size of the largest problem that a program can solve.

The space needed by a program consists of following components.

• *Instruction space* : Space needed to store the executable version of the program and it is fixed.

• *Data space* : Space needed to store all constants, variable values and has further two components :

(*a*) Space needed by constants and simple variables. This space is fixed.

(*b*) Space needed by fixed sized structural variables, such as arrays and structures.

(*c*) Dynamically allocated space. This space usually varies.

• ***Environment stack space****:* This space is needed to store the information to resume the suspended (partially completed) functions. Each time a function is invoked the following data is saved on the environment stack :

(*a*) Return address : *i.e.*, from where it has to resume after completion of the Called function.

(*b*) Values of all lead variables and the values of formal parameters in the function being invoked.

The amount of space needed by recursive function is called the recursion stack space. For each recursive function, this space depends on the space needed by the local variables and the formal parameter. In addition, this space depends on the maximum depth of the recursion *i.e.*, maximum number of nested recursive calls.

Time Complexity

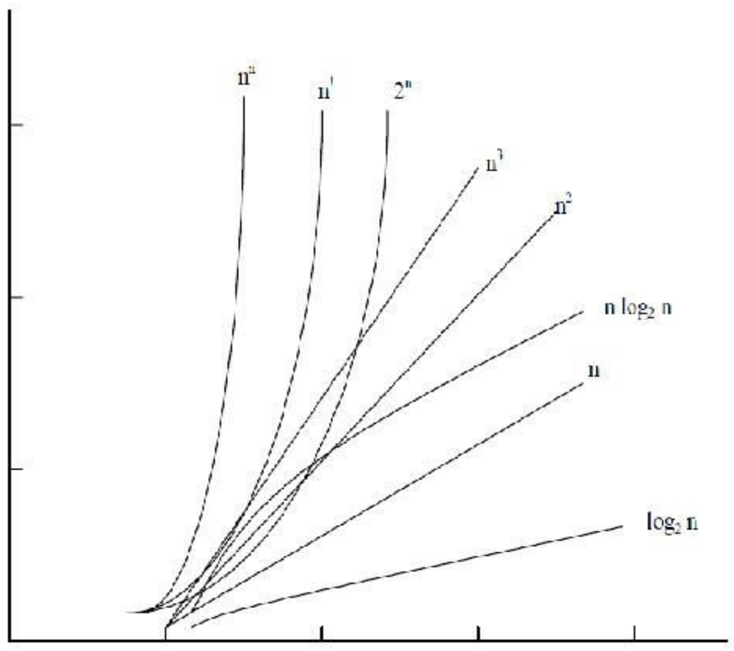
The time complexity of an algorithm or a program is the amount of time it needs to run to completion. The exact time will depend on the implementation of the algorithm, programming language, optimizing the capabilities of the compiler used, the CPU speed, other hardware characteristics/specifications and so on. To measure the time complexity accurately, we have to count all sorts of operations performed in an algorithm. If we know the time for each one of the primitive operations performed in a given computer, we can easily compute the time taken by an algorithm to complete its execution. This time will vary from machine to machine. By analyzing an algorithm, it is hard to come out with an exact time required. To find out exact time complexity, we need to know the exact instructions executed by the hardware and the time required for the instruction. The time complexity also depends on the amount of data inputted to an algorithm. But we can calculate the order of magnitude for the time required.

The time complexity also depends on the amount of data input to an algorithm, but we can calculate the order of magnitude for the time required. That is, our intention is to estimate the execution time of an algorithm irrespective of the computer machine on which it will be used.

Some of the reasons for studying time complexity are

a) We may be interest to know in advance that whether an algorithm or program will provide a satisfactory real time response
b) There may be several possible solutions with different time requirements

Here, the more sophisticated method is to identify the key operations and count such operations performed till the program completes its execution. A key operation in our algorithm is an operation that takes maximum time among all possible operations in the algorithm. Such an abstract, theoretical approach is not only useful for discussing and comparing algorithms, but also it is useful to improve solutions to practical problems. The time complexity can now be expressed as function of number of key operations performed. Before we go ahead with our discussions, it is important to understand the rate growth analysis of an algorithm, as shown in Figure.



The function that involves '*n*' as an exponent, *i.e.*, 2^n, n^n, $n\,!$ are called exponential functions, which is too slow except for small size input function where growth is less than or equal to n^c,(where '*c*' is a constant) *i.e.*; n^3, n^2, $n \log_2 n$, n, $\log_2 n$ are said to be polynomial. Algorithms with polynomial time can solve reasonable sized problems if the constant in the exponent is small.

When we analyze an algorithm it depends on the input data, there are three cases :

1. Best case
2. Average case
3. Worst case

In the best case, the amount of time a program might be expected to take on best possible input data.

In the average case, the amount of time a program might be expected to take on typical (or average) input data.

In the worst case, the amount of time a program would take on the worst possible input configuration.

Frequency Count

Frequency count method can be used to analyze a program .Here we assume that every statement takes the same constant amount of time for its execution. Hence the determination of time complexity of a given program is is the just matter of summing the frequency counts of all the statements of that program Consider the following examples

```
........              for(i=0;I,n;i++)       for(i=0;i<n;i++)
.........             X++;                   for(j=0;j<n;j++)
X++;                  .......                x++;
    (a)               (b)                    (c)
```

In the example (a) the statement x++ is not contained within any loop either explicit or implicit. Then its frequency count is just one. In example (b) same element will be executed n times and in example (3) it is executed by $n^{2.}$ From this frequency count we can analyze program

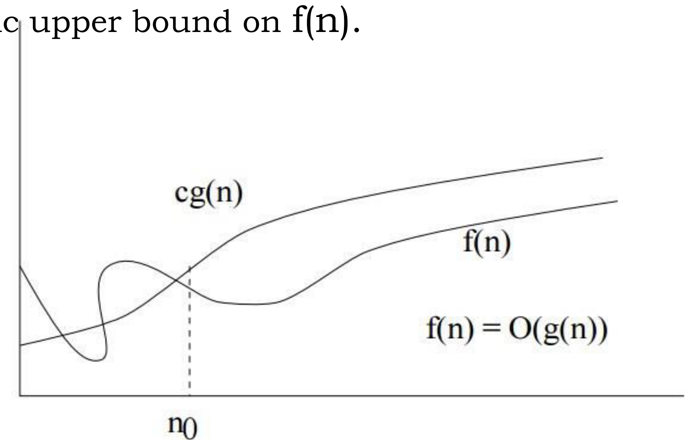
Growth of Functions and Asymptotic Notation

When we study algorithms, we are interested in characterizing them according to their efficiency. We are usually interesting in the order of growth of the running time of an algorithm, not in the exact running time. This is also referred to as the asymptotic running time. We need to develop a way to talk about rate of growth of functions so that we can compare algorithms. Asymptotic notation gives us a method for classifying functions according to their rate of growth.

Big-O Notation

Definition:

$f(n) = O(g(n))$ iff there are two positive constants c and n_0 such that $|f(n)| \leq c\ |g(n)|$ for all $n \geq n_0$. If f(n) is nonnegative, we can simplify the last condition to $0 \leq f(n) \leq c\ g(n)$ for all $n \geq n_0$. then we say that **"f(n) is big-O of g(n)."** . As n increases, f(n) grows n_o faster than g(n). In other words, g(n) is an asymptotic upper bound on f(n).



Example: $n^2 + n = O(n^3)$

Proof: • Here, we have $f(n) = n^2 + n$, and $g(n) = n^3$

• Notice that if $n \geq 1$, $n \leq n^3$ is clear.

• Also, notice that if $n \geq 1$, $n^2 \leq n^3$ is clear.

• In general, if $a \leq b$, then $n^a \leq n^b$ whenever $n \geq 1$. This fact is used often in these types of proofs.

• Therefore, $n^2 + n \leq n^3 + n^3 = 2n^3$

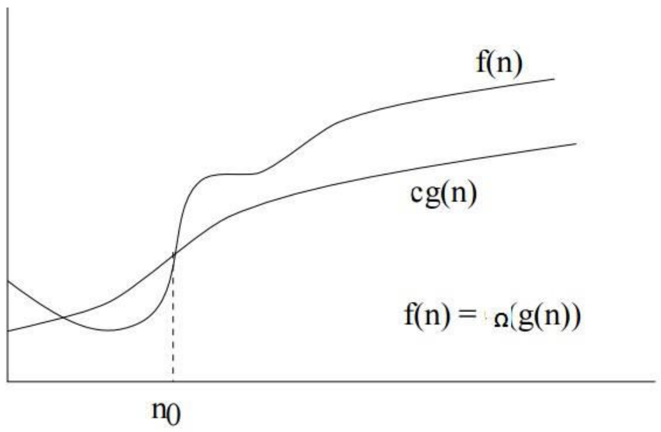
• We have just shown that $n^2 + n \leq 2n^3$ for all $n \geq 1$

• Thus, we have shown that $n^2 + n = O(n^3)$ (by definition of Big-O, with $n_0 = 1$, and $c = 2$.)

Big-Ω notation

Definition:

$f(n) = \Omega(g(n))$ iff there are two positive constants c and n_0 such that $|f(n)| \geq c\,|g(n)|$ for all $n \geq n0$ .If $f(n)$ is nonnegative, we can simplify the last condition to $0 \leq c\, g(n) \leq f(n)$ for all $n \geq n0$ • then we say that **"f(n) is omega of g(n)."** • As n increases, $f(n)$ grows no slower than $g(n)$. In other words, $g(n)$ is an asymptotic lower bound on $f(n)$



Example: $n^3 + 4n^2 = \Omega(n^2)$

Proof: • Here, we have $f(n) = n^3 + 4n^2$, and $g(n) = n^2$

• It is not too hard to see that if $n \geq 0$, $n^3 \leq n^3 + 4n^2$

• We have already seen that if $n \geq 1$, $n^2 \leq n^3$

• Thus when $n \geq 1$, $n^2 \leq n^3 \leq n^3 + 4n^2$

• Therefore,

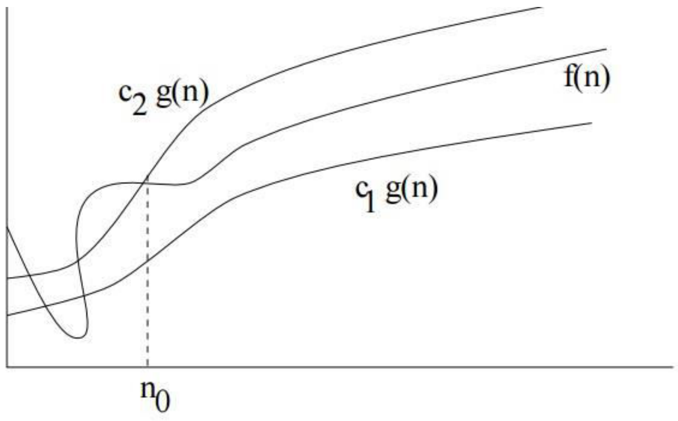
$$1n^2 \leq n^3 + 4n^2 \text{ for all } n \geq 1$$

• Thus, we have shown that $n^3 + 4n^2 = \Omega(n^2)$ (by definition of Big-Ω, with $n_0 = 1$, and $c = 1$.)

Big-Θ notation

Definition:

$f(n) = \Theta(g(n))$ iff there are three positive constants c_1, c_2 and n_0 such that $c1|g(n)| \leq |f(n)| \leq c2|g(n)|$ for all $n \geq n_0$. If f(n) is nonnegative, we can simplify the last condition to $0 \leq c1\ g(n) \leq f(n) \leq c2\ g(n)$ for all $n \geq n_0$ .then we say that **"f(n) is theta of g(n)."** . As n increases, f(n) grows at the same rate as g(n). In other words, g(n) is an asymptotically tight bound on f(n).



Example: $n^2 + 5n + 7 = \Theta(n^2)$

Proof: •

When $n \geq 1$, $n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2$

- When $n \geq 0$, $n^2 \leq n^2 + 5n + 7$
- Thus, when $n \geq 1$

$$1n^2 \leq n^2 + 5n + 7 \leq 13n^2$$

Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big-Θ, with $n_0 = 1$, $c1 = 1$, and $c2 = 13$.)

Comparison of different Algorithm

Algorithm	**Best case**	**Average case**	**Worst case**
Quick sort	O(n log n)	O(n log n)	$O(n^2)$
Merge sort	O(n log n)	O(n log n)	O(n log n)
Heap sort	O(n log n)	O(n log n)	O(n log n)
Bubble sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Binary search	O(1)	O(log n)	O(log n)
Linear search	O(1)	O(n)	O(n)