ARRAY REPRESENTATION IN MEMORY

An array is consecutive set of memory locations storing homogeneous data.

One dimensional array (interger type) is defined in C language as -

int arr[10], *ptr[10];

In C, all arrays start at index **0**. The address of the first element arr[0], is called the **base address**.

The memory address of **arr[i]**, is

Addr(arr[i]) := α + i * sizeof(int), where α denotes the base address.

The study of a data structure is a logical concept whereas its representation of data in computer memory is a physical concept.

Thus, an array can be defined as a set of **index–value pairs**, such that for each index of an array, there exists and associated value.

Indexing Function:

Since a data structure represents an abstract form of an item alongwith operations, it is essesntial to provide mapping between the abstract representation of the data structure and its physical representation.

For arrays, the mapping is the function of indexing variable(s) and it provides a unique integer that represents the exact location of an element with respect to the base address of the array under consideration. This function is known as indexing function.

All arrays are represented in main memory (RAM) as linear array of consecutive memory cells, irrespective of the number of dimensions in their abstract representation.

ADDRESS CALCULATION OF 1D ARRAY (i.e., a Vector)

Addr(A[i]) = Base_Address + (i – lb) * Width_of_Element

Base_Address	:= **b**	
Lower Limit/Bound	:= **lb**	//for a subscript or dimension
Subscript of Element	:= **i**	//address to be found
Width_of_Element	:= **w**	//storage size in bytes for an element

Addr(A[i]) := b + (i – lb) * w

Address Calculation

While storing the elements of a 2D array in memory, they are allocated contiguous memory locations and requires linearization to enable their storage.

There are two alternatives to achieve linearization:

Row-Major linearization and Column-Major linearization.

Fortran (for scientific computing) uses column-major layout. Column vectors are more commonly used in linear algebra computations. Matlab, R, Scilab, GNU Octave, IDL [Interactive Data Language] and Julia use column-major layout to benefit from LAPACK - a fast Fortran library for linear algebra.
C and C++ use row-major layout. Other popular languages using row-major layout - Python, Pascal, PL/I and Mathematica.

Consider a 3D array represented as **row** x **column** x **depth**.

In C layout (row-major), the first dimension (rows) changes the slowest while the third dimension (depth) changes the fastest.

In Fortran layout (column-major) the first dimension changes the fastest while the third dimension changes the slowest.

The row-major order and column-major order are methods for storing multi-dimensional arrays in linear storage such as random access memory. Row-major grouping starts from the leftmost index and column-major from the rightmost index, leading to lexicographic and colexicographics orders, respectively.

Row Major Order

This is the most natural way of storing an array into the memory. The elements in the first row are followed by elements in the second, third rows and so on. In row-major order, for a 2D matrix we traverse it row-by-row, then within each row enumerate the column.

Column Major Order

In column-major order, for a 2D matrix we traverse it column-by-column, then enumerate the rows within each column.

ADDRESS CALCULATION IN 2D ARRAY (i.e., a Matrix)

For a 2-d array with **m** [lr .. ur] rows and **n** [lc .. uc] columns,

Row-Major Ordering [C Style | Lexocographic]

$$\textbf{Addr(A[i][j])} = \textbf{b} + (\ (\textbf{i} - \textbf{lr})\ *\ \textbf{n} + (\textbf{j} - \textbf{lc})\ )\ *\ \textbf{w}$$

$$n = (\ uc - lc\ ) + 1$$

Column-Major Ordering [Fortran Style | Colexicographic]

$$\textbf{Addr(A[i][j])} = \textbf{b} + (\ (\textbf{i} - \textbf{lr}) + (\textbf{j} - \textbf{lc})\ *\ \textbf{m}\ )\ *\ \textbf{w}$$

$$m = (\ ur - lr\ ) + 1$$

Python/NumPy refers to the orderings in array flags as C_CONTIGUOUS and F_CONTIGUOUS, respectively.

```
import numpy as np
np.reshape(np.arange(1,10), (3,3), "F")
np.reshape(np.arange(1,10), (3,3), "C")
```

```
ar2D = [[1,2,3],[4,5,6],[7,8,9]]
rm2D = np.array(ar2D, dtype='uint8',order='C');
' '.join(str(ord(x)) for x in rm2D.data)
```

```
'1 2 3 4 5 6 7 8 9'
```

```
cm2D = np.array(ar2D, dtype='uint8',order='F');
' '.join(str(ord(x)) for x in cm2D.data)
```

```
'1 4 7 2 5 8 3 6 9'
```

```
ar3D = [[[1,2,3],[4,5,6],[7,8,9]],[[10,11,12],[13,14,15],
         [16,17,18]],[[19,20,21],[22,23,24],[25,26,27]]]
```

```
rm3D = np.array(ar3D, dtype='uint8',order='C');
' '.join(str(ord(x)) for x in rm3D.data)
```

```
'1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27'
```

```
cm3D = np.array(ar3D, dtype='uint8',order='F');
' '.join(str(ord(x)) for x in cm3D.data)
```

```
'1 10 19 4 13 22 7 16 25 2 11 20 5 14 23 8 17 26 3 12 21 6 15 24 9 18 27'
```

```
np.reshape(np.arange(1,17), (2,2,2,2), "C")
    array([[[[ 1,  2],
             [ 3,  4]],
            [[ 5,  6],
             [ 7,  8]]],
           [[[ 9, 10],
             [11, 12]],
            [[13, 14],
             [15, 16]]]])

np.reshape(np.arange(1,17), (2,2,2,2), "F")
    array([[[[ 1,  9],
             [ 5, 13]],
            [[ 3, 11],
             [ 7, 15]]],
           [[[ 2, 10],
             [ 6, 14]],
            [[ 4, 12],
             [ 8, 16]]]])
```

Significance:

Data layout is critical for correctly passing arrays between programs written in different programming languages. Modern CPUs process sequential data more efficiently than nonsequential data [CPU caching]. Contiguous access enables use of SIMD instructions to operate on vectors of data.

Neither Row-Major nor Column-Major

For Dense Arrays [use of Iliffe Vectors]

Java [Arrays-of-Arrays], C#, .Net, Scala, CLI

For Less Dense Arrays

Python [use of Lists-of-lists], Lua [Tables-of-Tables].

External Libraries

Row-major order is the default in NumPy [Python].

Column-major order is the default in Eigen [C++].

ADDRESS CALCULATION OF N-DIMENSIONAL ARRAY

For a d-dimensional N_1 x N_2 x ... x N_d array with dimensions N_k(k = 1..d), a given element of this array is specified by a tuple (n_1,n_2,..,n_d) of **d** (zero-based) indices $n_k \in [0, N_k\text{-}1]$.

In row-major order, the last dimension is contiguous, so the memory-offset of this element is given by:

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\cdots + N_2 n_1) \cdots))) = \sum_{k=1}^{d} \left(\prod_{\ell=k+1}^{d} N_\ell \right) n_k$$

In column-major order, the first dimension is contiguous, so the memory-offset of this element is given by:

$$n_1 + N_1 \cdot (n_2 + N_2 \cdot (n_3 + N_3 \cdot (\cdots + N_{d-1} n_d) \cdots))) = \sum_{k=1}^{d} \left(\prod_{\ell=1}^{k-1} N_\ell \right) n_k$$

where the empty product is the multiplicative identity element, i.e., .

$$\prod_{\ell=1}^{0} N_\ell = \prod_{\ell=d+1}^{d} N_\ell = 1$$

For d = 2

CM_offset := $n_1 + N_1 * n_2$

RM_offset := $n_2 + N_2 * n_1$

In image processing, the first dimension of an image array is the column, and the second dimension is the row. IDL is widely used for image processing.

In the mathematical notation used for linear algebra, the first dimension of an array (or matrix) is the row, and the second dimension is the column.

EFFICIENCY

Accessing memory in the wrong order can impose a severe performance penalty if the data is larger than the physical memory of computer.

Accessing elements of an array along the contiguous dimension minimizes the amount of memory paging required by the virtual memory subsystem of the computer hardware, and will therefore be the most efficient.

Accessing memory across the non-contiguous dimension can cause each such access to occur on a different page of system memory.

This forces the virtual memory subsystem into a cycle in which it must continually force current pages of memory to disk in order to make room for new pages, each of which is only momentarily accessed. This inefficient use of virtual memory is commonly known as thrashing.

ALGORITHMS & COMPLEXITY

Algorithm

Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

An algorithm can be viewed as a tool for solving a well-specified computational problem. The statement of the problem specifies the desired input/output relationship. The algorithm describes a specific computational procedure for achieving this input/output relationship.

An **algorithm** is a finite set of instructions that, if followed, accomplishes a particular task. It must satisfy following criteria -

1. Input: There are zero or more quantities that are externally supplied.
2. Output: At least one quantity is produced.
3. Definiteness: Each instruction is clear and unambigous.
4. Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. Effectiveness: Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3), but also must be feasible.

Algorithms exist for many common problems. It is very crucial for development of large-scale computer systems [Industrial Strength Software – ISS] to have an efficiently designed algorithm.

A program and an algorithm differs computationally on the ground that, the former does not have to satisfy condition [4]. For example, an Operating System may continue in the wait loop until more jobs are entered. It terminates when the system crashes or encounter a fatal error.

Algorithms as a Technology

Suppose computers were infinitely fast and computer memory was free. Would it still be necessary to study algorithms? The answer is YES. In absense of no other reason, the programmer would still like to demonstrate that his/her solution method terminates and does so with the correct answer.

Of course, computers may be fast, but they are not infinitely fast. And memory may be inexpensive, but it is not free. Computing time is therefore a bounded resource, and so is space in memory. The programmer/developer should use these resources wisely, and algorithms that are efficient in terms of time or space will facilitate achieving this goal.

Efficiency

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

For example, an Insertion sort, takes time roughly equal to $\mathbf{c_1.n^2}$ to sort $\mathbf{n}$ items, where $\mathbf{c_1}$ is a constant that does not depend on $\mathbf{n}$. That is, it takes time roughly proportional to $\mathbf{n^2}$.

The merge sort, takes time roughly equal to $\mathbf{c_2.(n.lg(n))}$, where $\mathbf{lg(n)}$ stands for $\mathbf{log_2(n)}$ and $\mathbf{c_2}$ is another constant that also does not depend on $\mathbf{n}$. insertion sort typically has a smaller constant factor than the merge sort, so that $\mathbf{c_1 < c_2}$.

Although insertion sort usually runs faster than merge sort for small input sizes, once the input size **n** becomes large enough, merge sort's advantage of **lg(n)** versus **n** will more than compensate for the difference in constant factors. No matter how much smaller $\mathbf{c_1}$ is than $\mathbf{c_2}$, there will always be a crossover point beyond which merge sort is faster.

Comparative Example

Let us put a faster computer (Comp_A) running insertion sort against a slower computer (Comp_B) running merge sort. They each must sort an array of **10M** numbers. (8-byte; input size **80M**).

Suppose that Comp_A executes 10 BIPS and Comp_B executes only 10 MIPS [Comp_A is 1000 times faster than Comp_B].

Let world's craftiest programmer codes **insertion sort in machine language** for Comp_A, requiring $\mathbf{2n^2}$ instructions to sort **n** numbers.

An average programmer implements **merge sort in HLL** with an inefficient compiler requiring $\mathbf{50.n(lg(n))}$ instructions.

To sort 10M numbers, Comp_A takes

$(2*10^{7})^2$ instructions / 10^{10} instructions/sec := 20000 sec [= **5.5** hrs]

while Comp_B takes

$50*(10^7 \mathbf{lg} 10^7)$ instructions / 10^7 instructions/sec := 1163 sec [= **20** min]

By using an algorithm whose running time grows more slowly, even with a poor compiler, Comp_B runs more than 17 times faster than Comp_A.

In general, as the problem size increases, so does the relative advantage of merge sort.

PERFORMANCE ANALYSIS

To evaluate a program following criteria are used -

1. Does the program meet the original specification of the task?
2. Does it work correctly?
3. Does the program contain documentation that shows how to use it and how it works?
4. Does the program effectievely use functions to create logical units?
5. Is the program's code readable?

Criteria 1-5 are associated with the development of a good programming style.

Two additional and concrete criteria to judge a program are -

6. Does the program efficiently use primary and secondary **storage**?
7. Is the **program's running time** acceptable for the task?

SPACE COMPLEXITY

The space complexity of a program is the amount of memory that it needs to run to completion. The space requirement of the program is the sum of following components -

- **Fixed Space Requirements**: This component referes to space requirement that do not depend on the number and size of program's inputs and outputs. It includes instruction space (to store the program code), storage for simple variables, fixed-size structured variables (such as struct, class, union), and constants.
- **Variable Space Requirements**: This component consists of the space needed by structured variables whose size depends on the particular instance, I, of the problem being solved. It also includes the additional space required when a function uses recursion.

S(I) denotes the variable space requirement of a program **P** working on an instance **I**. It is a function of the characteristics of an instance I, namely, number, size and values of inputs and outputs associated with I.

The total space requirement of any program is -

S(P) := c + S(I) ; where **c** denotes the fixed space requirements.

Example-01

```
float abc(float a, float b, float c){
    return (a + b - c) / (3 * b);
}
```

Space Requirement Computation

The function **abc()** incorporates only simple variables for input and output, therefore it accounts for only fixed space requirements. Hence,

S(I) = 0

Example-02

```
float sum(float list [], int n){
    float psum = 0;
    int i;
    for(i = 0; i < n; i++)
        psum = psum + list[i];
    return psum;
}
```

Space Requirement Computation

Here the space requirement depends on how the array is passed into the function. C always passes arrays by reference (or address), hence the array is not copied. Therefore, it accounts for a fixed space requirements.

$S(I) = 0$

The PASCAL scenario:

Pascal may pass arrays by value. Thus, the array is copied into temporary storage before the function is executed.

$$\mathbf{S(I) = S(n) = n}$$

where **n** denotes size of the array.

Example-03

```
float rsum(float list [], int n){
    if(n)
        return rsum(list, n-1) + list[n-1];
    return 0;
}
```

Space Requirement Computation

Each recursive call requires space for two parameters and a return address. In **gcc** an integer and an address pointer requires 4 bytes.

Thus, storage for one Recursive Call

= storage for (array pointer, list[ ] + integer, n + return address)

= 4 bytes + 4bytes + 4 bytes = 12 bytes

For n = 100,

S(I) = S(n) = 12 * 100 = <u>1200 bytes</u>

It may be noted that the iterative counterpart of rsum() has no variable space requirements.

TIME COMPLEXITY

The time compexity of a program is the amount of computer time that it needs to run to completion. The time, **T(P)**, taken by a program, P, is the sum of its **compile time** and its **run** (or **execution**) **time**.

The compile time does not depends on the instance characteristics and is more or less constant. Also, when it is verified that the program ran correctly, it may be re-executed without recompiling thereafter.

Thus, time complexity depend on program's execution time, $\mathbf{T_p}$. Computation of $\mathbf{T_p}$ necessitates in-depth knowledge of the compiler's attributes regarding how it transaltes the source code into the object code.

STEP COUNT

It is perfectly feasible to count the number of operations the program performs. This provides us the machine-independent estimate of running time. But this depends on segregating the program into distinct steps.

The approach involves determining the step count for a program.

Program Step: It is a syntactically or symantically meaningful program segment whose execution time is independent of the instance characteristics.

For example, the instructions

`kount = psum;` and `psum = psum * 2 + kount;`

can be individually considered as single step, unless they are independent of instance characteristics.

Example-04 (Function sum() with count statements)

```
float sum(float list [], int n){
    float psum = 0; count++;    // for assignment
    int i;
    for(i = 0; i < n; i++){
        count++;                // for the for loop
        psum = psum + list[i];
        count++;                // for assignment
    }
    count++;                    // last execution of for
    return psum;
    count++;                    // for return value
}
```

Transforming above code by eiminating program statements provides the estimate of program step count.

Example-05 (Simplified Example-04)

```
float sum(float list [], int n){
    float psum = 0;
    int i;
    for(i = 0; i < n; i++){
        count += 2;
    }
    count += 3;
}
```

Thus, step count for **sum()** := **2n + 3**.

Example-06 (Function rsum() with count statements)

```
float rsum(float list [], int n){
    count++;              // for if conditional
    if(n){
        count++;          // for return and rsum invocation
        return rsum(list, n-1) + list[n-1];
    }
    count++;
    return 0;
}
```

For the boundary condition, when n = 0, only the if conditional and the outer return statement are executed. Thus, the step count for n = 0 is 2.

For n > 0, the if conditional and the inner return statement are executed. Each recursive call with n > 0 adds 2 to step count. As there are **n** such function calls, the total contribution to step count is 2n.

Thus, step count for **rsum()** := **2n + 2**.

^**

The recursive function may have a lower step count than its iterative counterpart. But it must be mentioned that the step count only tells us about how many steps are execute, and does not reveal the time taken by each step.

It is established (to be seen later) although recursive function has fewer steps, it typically runs more slowly than the iterative version. This is due to the recursive steps taking longer to execute than iterative steps.

**^

Consider the iterative function for adding two matrices of size **rows * cols**.

Example-07 (Matrix Addition)

```
float addMat(int a[][MAX_SZ], int b[][MAX_SZ]
             int c[][MAX_SZ], int rows, int cols){
    int i, j;
    for(i = 0; i < rows; i++){
        count++;                    // for i for loop
        for(j = 0; j < cols; j++){
            count++;                // for j for loop
            c[i][j] = a[i][j] + b[i][j];
            count++;                // for assignment  stmt
        }
        count++;                    // last time of j for loop
    }
    count++;          // last time of i for loop
}
```

Example-08 (Simplified **addMat()** with count statements)

```
float addMat(int a[][MAX_SZ], int b[][MAX_SZ]
             int c[][MAX_SZ], int rows, int cols){
    int i, j;
    for(i = 0; i < rows; i++){
        for(j = 0; j < cols; j++){
            count += 2;
        }
        count += 2;
    }
    count++;
}
```

Total Steps := step count (**j for** loop) + step count(**i for** loop) + 1

:= (2 * rows * cols) + (2 * rows) + 1

Step Count : the Tabular Approach

The step count for each statement is **s/e** (a short for steps/execution)

The sum() function ...

program statements	**s/e**	**frequency**	**tot_steps**
`float sum(float list [], int n){`	0	0	0
`  float psum = 0;`	1	1	1
`  int i;`	0	0	0
`  for(i = 0; i < n; i++)`	1	n+1	n+1
`  psum = psum + list[i];`	1	n	n
`  return psum;`	1	1	1
`}`	0	0	0
			2n+3

The rsum() function ...

program statements	s/e	frequency	tot_steps
`float rsum(float list [], int n){`	0	0	0
`  if(n)`	1	n+1	n+1
`     return rsum(list, n-1) + list[n-1];`	1	n	n
`  return 0;`	1	1	1
`}`	0	0	0
			2n+2

The addMat() function ...

program statements	s/e	frequency	tot_steps
`float addMat(int a[][MAX_SZ],`	0	0	0
`  int b[][MAX_SZ], int rows,`			
`  int c[][MAX_SZ], int cols){`			
`  int i, j;`	0	0	0
`  for(i = 0; i < rows; i++)`	1	rows+1	rows+1
`    for(j = 0; j < cols; j++)`	1	rows*(cols+1)	rows*cols+rows
`     c[i][j] = a[i][j] + b[i][j];`	1	rows*cols	rows*cols
`}`	0	0	0
			2 rows . cols + 2 rows + 1

The Best, Average and Worst case step counts

The **best case step count** is the minimum number of steps that can be executed for the given parameters.

The **worst case step count** is the maximum number of steps that can be executed for the given parameters.

The **average step count** is the average number of steps executed on instances with given parameters.

ASYMPTOTIC NOTATION (O, Ω, Θ)

For the comparative analyses of algorithms the step count approach is not suitable due to inexactness in iterpretation of the notion of a step.

The Big "oh" Notation (O)

$f(n) = O(g(n))$ ... f of n is big oh of g of n

iff ... if and only if

there exist positive constants c and n_0 such that

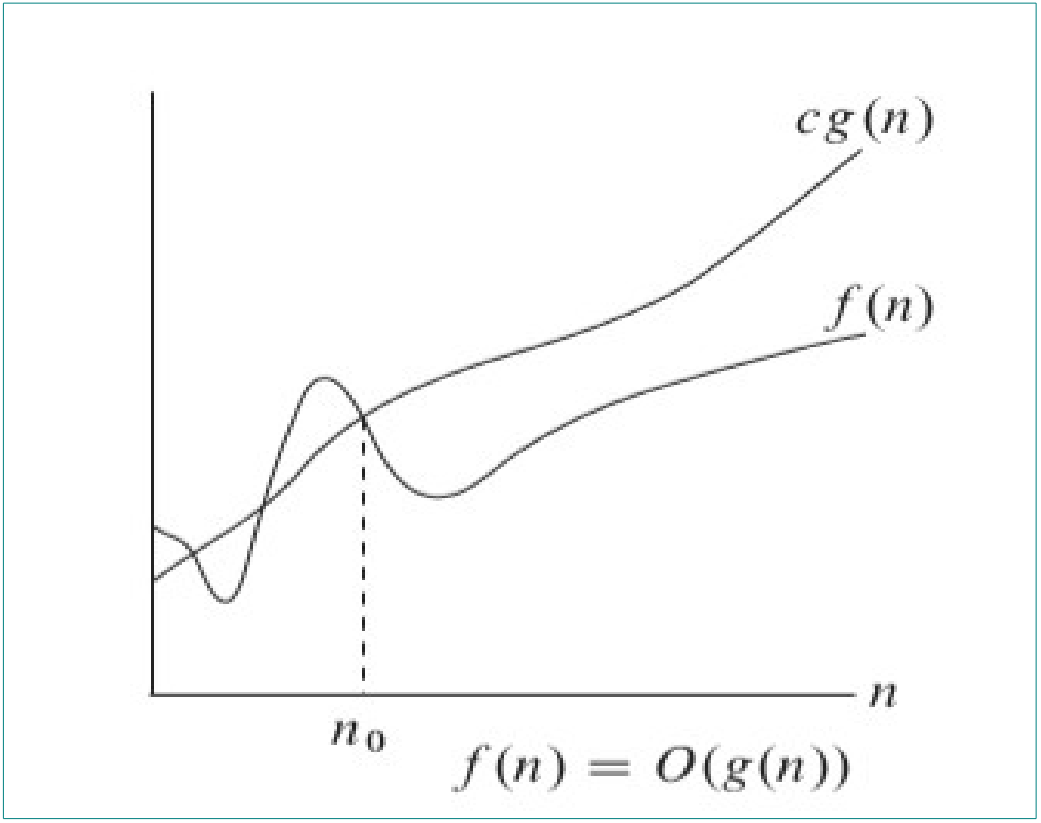
$f(n) \leq cg(n)$ for **all n, $n \geq n_0$**.

$3n + 2 = O(n)$ as $3n + 2 \leq 4n$ for all $n \geq 2$.

$3n + 3 = O(n)$ as $3n + 3 \leq 4n$ for all $n \geq 3$.

$100n + 6 = O(n)$ as $100n + 6 \leq 101n$ for all $n \geq 10$.

$10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$.


cg(n)
f(n)
n
n0
f(n) = O(g(n))

Computing Time and Growth of Functions

$O(1)$:= Constant $O(n)$:= Linear $O(n^2)$:= Quadratic

$O(n^3)$:= Cubic $O(2^n)$:= Exponential

$$O(1) \leq O(\log n) \leq O(n) \leq O(n \log n) \leq O(n^2) \leq O(n^3) \leq O(2^n)$$

Remember -

- The statement $f(n) = O(g(n))$ only states that $g(n)$ is an **upper bound** on the value of $f(n)$ for all n, $n \geq n_0$. It does not reveal specifics about the goodness of this bound.
- For the statement $f(n) = O(g(n))$ to be informative, $g(n)$ should be as small a function of n as can be, for which $f(n) = O(g(n))$.
- Hence, although $\mathbf{3n + 3 = O(n^2)}$ is a correct statement (it is seldom stated that way), there exists a better approximation $\mathbf{3n + 3 = O(n)}$ which therefore is stated and used.

The Omega Notation (Ω)

$f(n) = \Omega(g(n))$ ... f of n is omega of g of n

iff ... if and only if

there exist positive constants c and n_0 such that

$f(n) \geq cg(n)$ for all n, $n \geq n_0$.

$3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for all $n \geq 1$. [it is required that $n_0 > 0$]

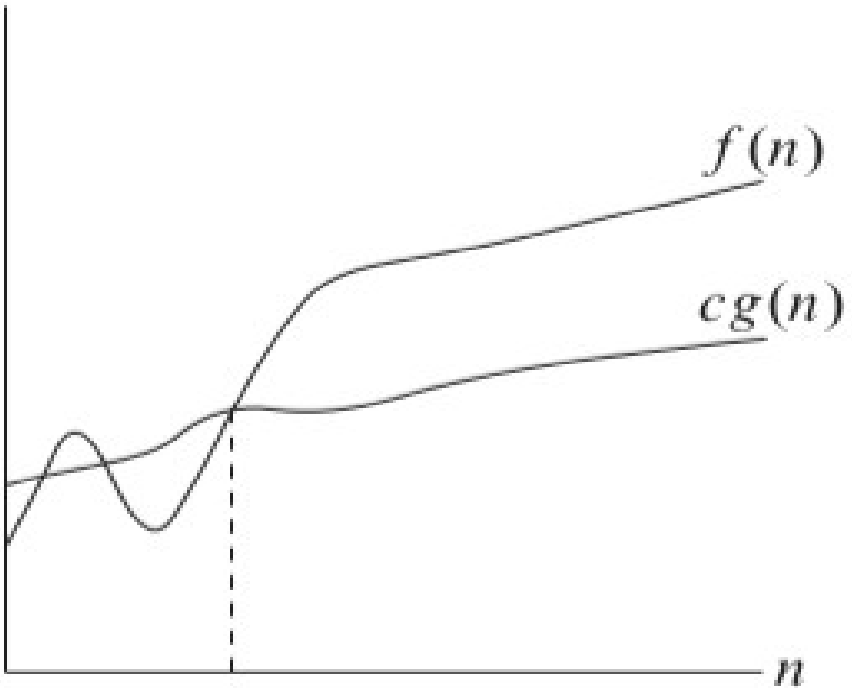
$3n + 3 = \Omega(n)$ as $3n + 3 \geq 3n$ for all $n \geq 1$.

$100n + 6 = \Omega(n)$ as $100n + 6 \geq 100n$ for all $n \geq 1$.

$10\,n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for all $n \geq 1$.

Remember -

- There exists several functions g(n) for which $f(n) = \Omega(g(n))$. g(n) is only a **lower bound** on f(n).
- For the statement $f(n) = \Omega(g(n))$ to be informative, g(n) should be as large a function of n as possible for which $f(n) = \Omega(g(n))$ is true.



$f(n) = \Omega(g(n))$

The Theta Notation (Θ)

$f(n) = \Theta(g(n))$ ... f of n is theta of g of n

iff ... if and only if

there exist positive constants c_1, c_2 and n_0 such that

$\mathbf{c_1\, g(n) \leq f(n) \leq c_2\, g(n)}$ for all n, $n \geq n_0$.

$3n+2 = \Theta(n)$ as $3n+2 \geq 3n$ for all $n \geq 2$ and $3n+2 \leq 4n$ for all $n \geq 2$;

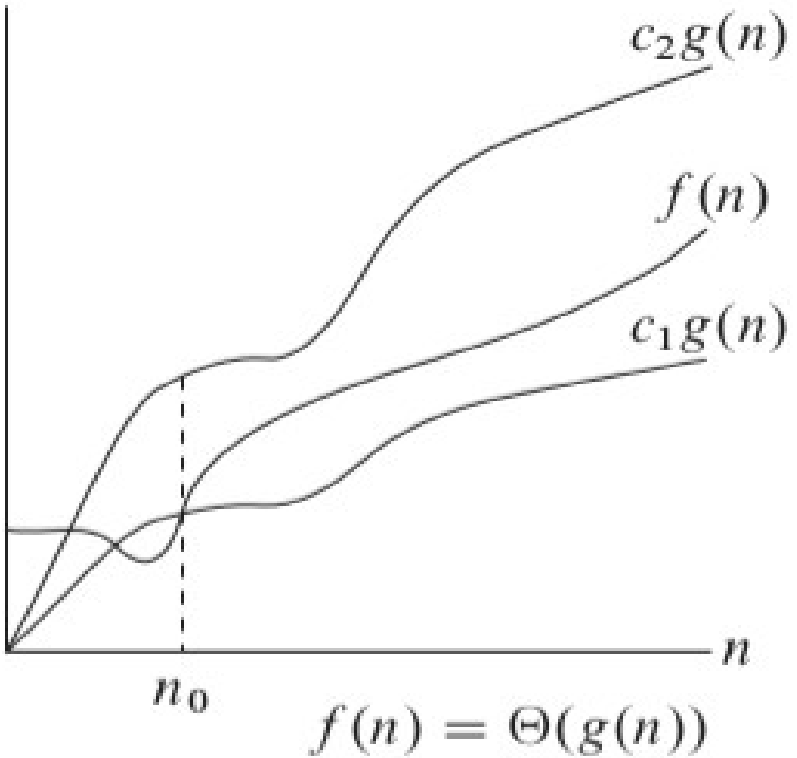
so $c_1 = 3$, $c_2 = 4$, and $n_0 = 2$.

$3n + 3 = \Theta(n)$; $10n^2 + 4n + 2 = \Theta(n^2)$

$6*2^n + n^2 = \Theta(2^n)$; $10*\log n + 4 = \Theta(\log n)$

Remember -

- The theta notation is **more precise** as $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper and lower bound on $f(n)$.


$c_2g(n)$
$f(n)$
$c_1g(n)$
n
n_0
$f(n) = \Theta(g(n))$