Design and Analysis of Algorithms (DAA)

Basic introduction and time and space complexity analysis

Dr. Priyadarshan Dhabe,

Ph.D (IIT Bombay)

Syllabus

- **Basic introduction and time and space complexity analysis:**

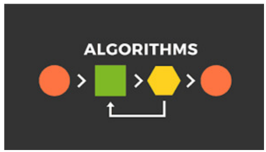
Asymptotic notations (Big Oh, small oh, Big Omega, Theta notations). Best case, average case, and worst-case time and space complexity of algorithms. Overview of searching, sorting algorithms. Using Recurrence relations and Mathematical Induction to get asymptotic bounds on time complexity. Proving correctness of algorithms.

What is an Algorithm?

- **Definition Of Algorithm**

(Ref-https://en.wikipedia.org/wiki/Algorithm

- In mathematics and computer science, an **algorithm** is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation.
- Algorithms are always unambiguous and are used as specifications for performing calculations, data processing, automated reasoning, and other tasks.
- It has an input (can also be empty) and produces output (goal)



What is an 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- *Book of Cormen and et.all*
- Algorithm can be used as a *tool* solve a ***computational problem***.

Computational problem Example

- **Sorting problem**

$$Input :- \text{A sequence of n numbers} < \mathrm{a}_1, a_2, \ldots, a_n >$$

$$Output :- \text{A permutation (reordering)} < \mathrm{a}_1', a_2', \ldots, a_n' >$$

$$\text{such that } \mathrm{a}_1' \leq a_2' \leq, \ldots, \leq a_n'$$

- Input can be **<31; 41; 59; 26; 41; 58>** and output produced by algorithm will be

  <26; 31; 41; 41; 58; 59>

- Input sequence is call ***instance*** of sorting program Or ***instance of a problem.***

Correctness of Algorithm

- Algorithm is said to be ***correct***, if for **every** input instance it **halts** with **correct output**.
- We say that the **correct algorithm** solves computational problem.
- **Incorrect Algorithms**
  - **May not halt at all for some inputs**
  - **Or halts with incorrect output**
- Algorithm can be specified in English, a Computer program or even as a hardware design.
- Generally a **pseudocode** is commonly used which looks like C, C++ code.

What is Analysis of the algorithm?

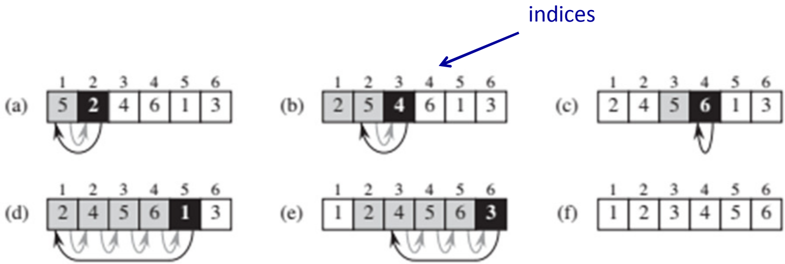
- **Analyzing** an algorithm means finding out
  - **Execution time-** arithmetic operations
  - **Memory requirement-** space requirement
- Since, execution time is machine dependent (depends on RAM, Processor speed, Cache, OS and etc.), we are interested in finding number of basic operations in terms of input size (n)
- The amount of memory needed is also counted in terms of input size n.

Insertion sort Analysis

- It works like **how we add a new card in left hand ?** at its appropriate location. We are inserting the card at correct place by comparing it with previous sorted cards.

Insertion sort Working

- Sort Array A=[5,2,4,6,1,3]



Ref- introduction to algorithms- By Cormen et.al

Insertion sort Algorithm

A.length=n=6

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]        // j th element to be placed at its proper place
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1            shift item A[i] at A[i+1]
    A[i + 1] = key    Copy key at its proper location in A
```

Insertion sort time analysis

INSERTION-SORT(A)	cost	times
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $\quad key = A[j]$	c_2	$n-1$
3 $\quad$ // Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
4 $\quad i = j - 1$	c_4	$n-1$
5 $\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $\quad\quad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $\quad\quad i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $\quad A[i+1] = key$	c_8	$n-1$

c_i – is the execution time needed for line i

t_j – denote the number of times the while loop runs in line 5 for that value of j

$$\sum_{j=2}^{n} t_j = t_2 + t_3 + \ldots + t_n$$

Insertion sort time analysis

- The *running time* of insertion sort *T(n)* is then

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) . \end{aligned}$$

- For **the best case** (A is sorted) the while loop will run once for each value of j, thus tj=1 . So the T(n) can be

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

- The above formula is in the type ***an+b,*** where a and b are constants, which is a *linear function*

What is a linear function?

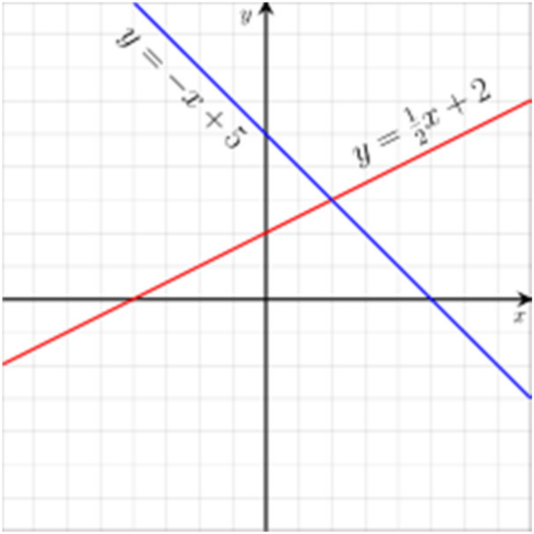
- A **linear function** is an algebraic equation in which each term is either a constant or the **product of a constant and (the first power of) a single variable**.
- Its graph is a line of form ***y=mx+c***, for constants m and c.
- Mathematically, algebraic equation that satisfy superposition theorem/principle

For constants a and b, and variables x and y

function f is linear iff,

$$f(ax+by)=a\cdot f(x)+b\cdot f(y)$$

Graph of linear function



Insertion sort time analysis

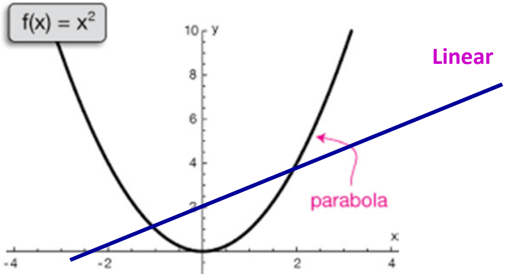
- **The worst case** is, A is in decreasing order, we must compare each element A[j] with each element of A[1,..,j-1] so tj=j. Using $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$

Worst case running time can be

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\
&\quad - (c_2 + c_4 + c_5 + c_8)\ .
\end{aligned}
$$

Worst case running time is in the form $an^2 + bn + c$,
for constants a, b and c, which is quadratic in nature

Quadratic function



$$Linear --- f(x) = ax + b$$

$$Quadratic -- f(x) = ax^2 + bx + c$$

Insertion sort time analysis

- The **Average case running time**.
- In this case half of the elements in **A [1,…j-1]** are less than A[j] and remaining are greater than A[j], Thus, tj=j/2. It also turns out to be a **Quadratic function**.
- We generally interested in "*rate of growth*" or "*order of growth*" of running time functions.

Thus, from $an^2 + bn + c$, we remove the lower ordered term and eliminate even the constant of highest ordered term and use it as n^2. we write that insertion sort has worst case running time of $\Theta(\mathrm{n}^2)$. Read it as Theta of n square.

Linear searching from unsorted array

- We have an unsorted array of n elements and we want to search a key=x. Find the best case, average case and worst case running time, assuming that k is constant time required for a single comparison.

A=[4,6,1,3,8,4] and x=4

Asymptotic notations

- **Asymptotic** – means approaching a **value** or **curve** arbitrary closely. (also called limiting behavior)
- We are interested in understanding **3 notations**

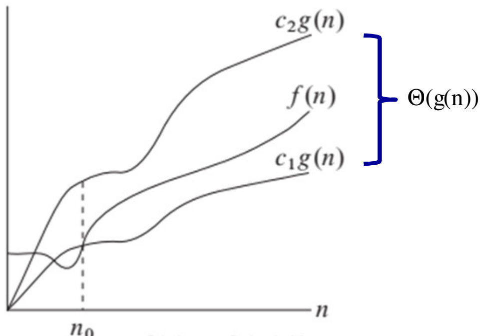
1. Θ – Theta - provide asymptotic tight bounds (lower and upper both)
2. O - Big Oh - provide asymptotic upper bound
3. Ω - Big Omega - provide asymptotic lower bound

Asymptotic notations

$$\Theta(g(n)) = \{f(n) : \text{ there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.^1$$

Read it as $\Theta(g(n))$ is set of all the functions $f(n)$ such that.....

We will write $f(n) \in \Theta(g(n))$ as $f(n) = \Theta(g(n))$



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

n is input size of Algorithm

Asymptotic notations

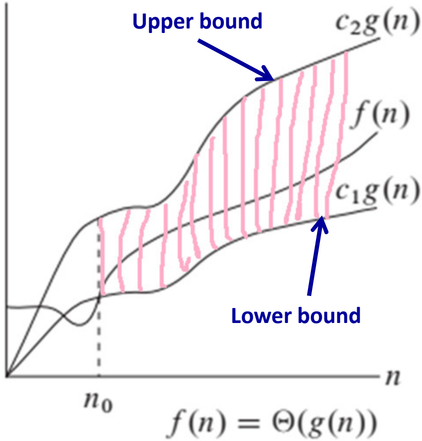
$f(n) = \Theta(g(n))$ means $f(n)$ can be any function that lie in marked pink region. There can be many such functions

$f(n)$ must be greater $\geq c_1 g(n)$ and $\leq c_2 g(n)$ for all $n \geq n_0$

For insertion sort running time

$T(n) = an^2 + bn + c$

Thus, $T(n) = \Theta(n^2)$



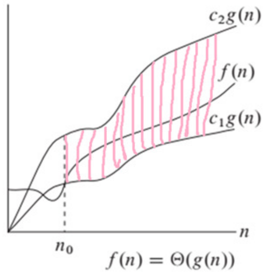
Asymptotic notations

Following all the functions are belonging to $\Theta(n^2)$

$$f(n) = 8n^2 + 3n + 4$$

$$f(n) = 106n^2 + 300n + 56$$

$$f(n) = n^2 + 33n + 400$$

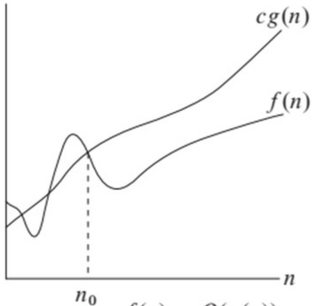


We say that $g(n)$ is asympotically tight bound for $f(n)$ for sufficiently large values of n and $f(n)$ is non - negative

Asymptotic notation- Big-Oh

$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$

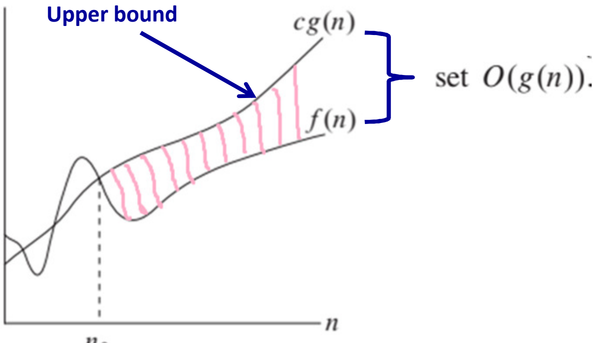
Big-Oh (O) represents asymptotic upper bound



$f(n) = O(g(n))$

We use O-notation to give an upper bound on a function, to within a constant factor. Figure [redacted] shows the intuition behind O-notation. For all values n at and to the right of n_0, the value of the function $f(n)$ is on or below $cg(n)$.

Asymptotic notation- Big-Oh

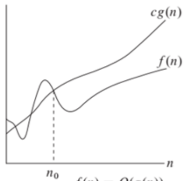


$f(n) = O(g(n))$

We write $f(n) = O(g(n))$ to indicate that a function $f(n)$ is a member of the set $O(g(n))$. Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since Θ-notation is a stronger notion than O-notation. Written set-theoretically, we have $\Theta(g(n)) \subseteq O(g(n))$. Thus, our proof that any quadratic function $an^2 + bn + c$, where $a > 0$, is in $\Theta(n^2)$ also shows that any such quadratic function is in $O(n^2)$.

Asymptotic notation- Big-Oh

Using O- notation , we can often describe the running time of an algorithm merely by inspecting its overall structure. E.g. The doubly nested loop structure of insertion sort Indicates that its worst case upper bound is *O(n^2)*.



$f(n) = O(g(n))$

generalization

Since O-notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm, we have a bound on the running time of the algorithm on every input—the blanket statement we discussed earlier. Thus, the $O(n^2)$ bound on worst-case running time of insertion sort also applies to its running time

Asymptotic notation- Big-Oh

input of size n. When we say "the running time is $O(n^2)$," we mean that there is a function $f(n)$ that is $O(n^2)$ such that for any value of n, no matter what particular input of size n is chosen, the running time on that input is bounded from above by the value $f(n)$. Equivalently, we mean that the worst-case running time is $O(n^2)$.

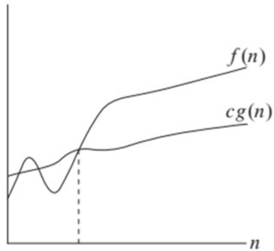
```
for  (i=1 to n)
     for (j=1 to n)
         for (k=1 to n)
           ..........
         end
      end
end
```

$$O(n^3)$$

Asymptotic notation- Big-Omega

Just as O-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions

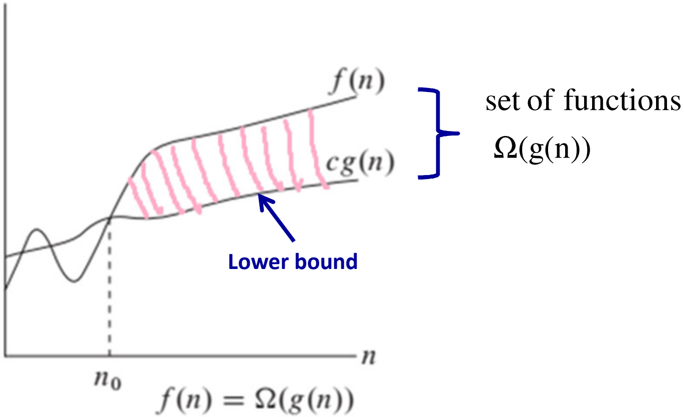
$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \} .$$



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

Figure shows the intuition behind Ω-notation. For all values n at or to the right of n_0, the value of $f(n)$ is on or above $cg(n)$.

Asymptotic notation- Big-Omega



For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ■

Asymptotic notation- Big-Omega

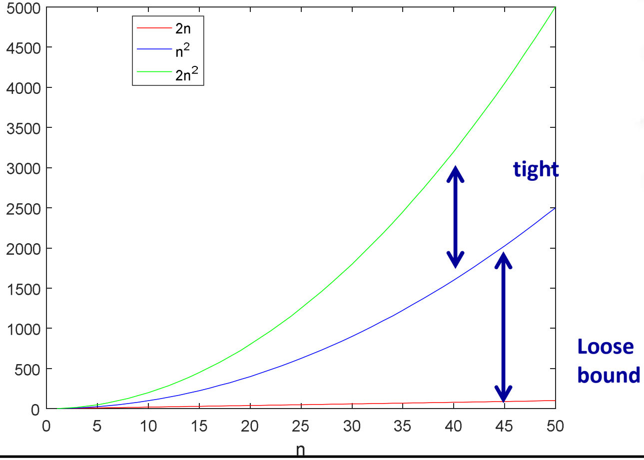
When we say that the *running time* (no modifier) of an algorithm is $\Omega(g(n))$, we mean that *no matter what particular input of size n is chosen for each value of n*, the running time on that input is at least a constant times $g(n)$, for sufficiently large n. Equivalently, we are giving a lower bound on the best-case running time of an algorithm. For example, the best-case running time of insertion sort is $\Omega(n)$, which implies that the running time of insertion sort is $\Omega(n)$.

contradictory, however, to say that the *worst-case* running time of insertion sort is $\Omega(n^2)$, since there exists an input that causes the algorithm to take $\Omega(n^2)$ time.

Asymptotic notation- little - o

The bound $2n^2 = O(n^2)$ is asymptotically tight

but $2n = O(n^2)$ is not tight



Asymptotic notation- little - oh

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight. We formally define $o(g(n))$ ("little-oh of g of n") as the set

$$o(g(n)) = \{ f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0 \} .$$

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \le f(n) \le cg(n)$ holds for *some* constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \le f(n) < cg(n)$ holds for *all* constants $c > 0$.

Asymptotic notation- little - oh

- Some authors define it using limit

$$f(n) = o(g(n))$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 .$$

If f(n) = n^2 and g(n) = n^3 then check whether f(n) = o(g(n)) or not.

$$\lim_{n \to \infty} \frac{n^2}{n^3}$$

$$= \lim_{n \to \infty} \frac{1}{n}$$

$$= \frac{1}{\infty}$$

$$= 0$$

Asymptotic notation- little – oh provides loose upper bound

Which bound we generally refers????

Mostly we are interested in computing worst case upper bound *O(g(n))* to compare algorithms and call it as worst case **time complexity**

Complexity-Refers- **quality of difficulty** or **complications, quality of being complex**

Computing time complexities

```
int Add(n, m)
{
   int sum=0;
   sum=n+m;
  return sum;
}
```

Total operations needed

T= 1 Addition
+ 1 assignment
+1 return

T=3=constant number of operations for every possible values of *n* and *m*.

Thus the time complexity is O(1)

Computing time complexities

Find an element in array A of size n=5

```
int Search(m)
{
   int i;
 for (i=I to  n)
     if (A[i]==m)
        Break;
     return i;
 end
}
```

Total operations needed in worst case

T= n comparisons

+ n assignments

+1 return

T=2n+1

Thus the time complexity is O(n)

Computing time complexities

Addition of two matrices of size (n x n)

```
MatAdd (A,B,C)
{
   int I, j;
 for (i=I to  n)
       for (j=I to  n)
              C(I,j)=A(I,j)+B(I,j);
       end

end
}
```

Total number of operations needed

T= (n x n) assignments for loops i, and j

+ (nxn) additions in loop A+B

+(n x n) assignments in C=A+B

T=3*(nx n)

$$T = 3n^2$$

$$Thus, O(n^2\ )$$

Even if we ignore assignment operations it has the same time complexity

Addition of two matrices of size (m x n) is thus, O(m x n)

Find the time complexity

- *Pseudocode*:

list_Sum(A,n) //A->array and n->elements

total =0

for i=0 to n-1

  sum = sum + A[i]

return sum

O(n)

```
for (int i = 1; i <=m; i += c) {
// some work
 }
for (int i = 1; i <=n; i += c) {
// some work
 }
```

If m!= n   O(m+n)

If m=n O(2n)=O(n)

Computing time complexities

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

-Innermost loop with k runs n times

- In this loop we require n additions and n multiplications i.e (n+n)
- This loop runs up to (nxn) times for above two loops
- Thus, the total number of operations needed
- T=(nxn)*(n+n)

$$T = (n \times n) * (n + n) = n^3 + n^3 = 2n^3$$

$$Thus, \text{is } O(n^3)$$

What is space complexity of an algorithm?

- The **space complexity** of an algorithm or a computer program is the amount of memory space required to solve an instance of the computational problem as a function of the input (size). It is the memory required by an algorithm to execute a program and produce output.- Wikipedia
- An algorithm/ Program need memory (main/RAM) for
  - Variables
  - Input and output data
  - Program stack for function calls (Auxiliary memory)
  - Instructions

What is space complexity of an algorithm?

- The space complexity is also expressed asymptotically in big O-h notation
- Space complexity is computed using

input size+ **auxiliary memory (additional)** required

Computing space complexity

```
int Add(n, m)
{
   int sum=0;
   sum=n+m;
  return sum;
}
```

-We need to store 3 integers *n*, *m* and *sum*.

- If each integer requires 4 bytes to store , hence we need
- 3X4=12 bytes+ some constant auxiliary memory (k)
- Thus total space needed
- ***S=(12 +k) bytes*** = constant, irrespective of any values of *n* and m
- Thus space complexity is *O(1)* i.e constant

Computing space complexity

Find an element in array A of size n=5

```
int Search(m)
{
   int i;
 for (i=I to  n)
     if (A[i]==m)
      break
end
     return i;

}
```

- We need to store *n* integers in A and thus need= 4xn bytes
- 4 bytes are need to store input *m*
- 4 bytes to store integer *i*
- Thus, total memory needed is
- **S=(4n+8) byes**
- Thus, the *space complexity* is **O(n)**

Computing space complexity

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

- 4 bytes for storing n of line 1
- 4*(nxn) bytes for storing matrix C in line2
- 4*(nxn) bytes for storing matrix A
- 4*(nxn) bytes for storing matrix B
- 4x3=12 bytes needed to store loop counters ***i, j, k***
- Thus, *total memory* needed *S* can be computed as

$$S = 4 + 12 * (nxn) + 12$$

$$S = 12n^2 + 16$$

Thus, space complexity is $O(n^2)$

- https://www.youtube.com/watch?v=yOb0BL-84h8
- Space complexity

Computing space complexity

- **Iterative version of factorial**

```
Int factorial (int n)
{
   int  i, fact = 1;
    for ( i = 1; i <= n; i++)
        fact = fact * i;
   return fact;
}
```

- We need 4 bytes for saving ***fact***
- 4 bytes for storing ***i***
- 4 bytes for storing **n**
- Some constant bytes **K**, as auxiliary space for initializing for loop and return statement
- Thus total space needed
- **S=12 bytes+ Auxiliary space(K)**
- **S=constant bytes**, irrespective of **value of n**

Thus, space complexity is O(1)

Computing space complexity

- **Recursive version of factorial**

```
int fact(int n)
{
   if(n<=1)
    return(1);
else
  // recursion
    return(n*fact(n-1));
}
```

fact(5) call stack

1
2*fact(1)
3*fact(2)
4*fact(3)
5*fact(4)

-We need some constant K bytes for the stack element for each call and have n calls

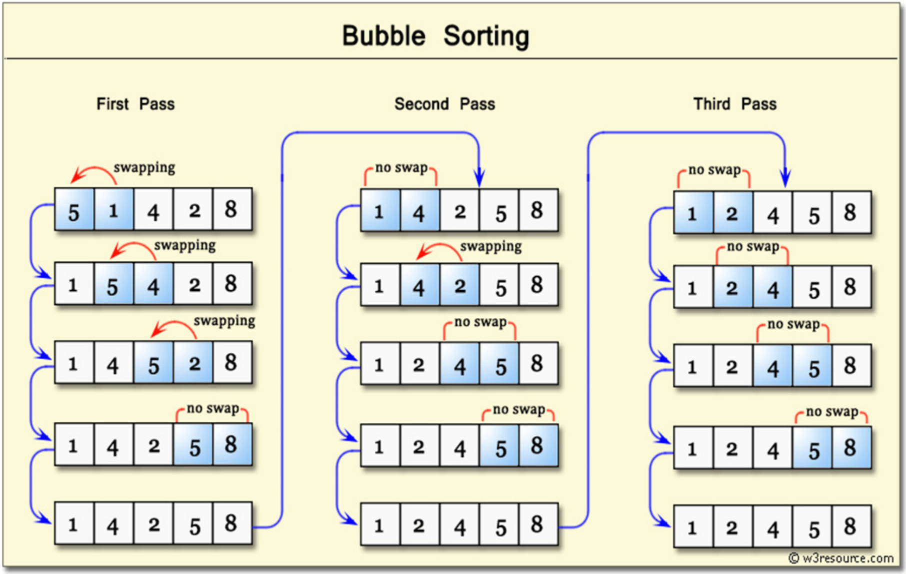
- 4Bytes for storing value of n

Thus total space **S = input size + Auxiliary space**

= 4 bytes+ (K*n)

Hence, space complexity is O(n). Input size don't have direct impact on space complexity.

Time complexity of bubble sort



Bubble sort Time Analysis

```
integer i, j,n;
n=A.length (Array  length )
for (i=0;i<n;i++)
   for(j=0;j<n-i-1;j++)
           if(A[j]>A[j+1])
               temp=A[j];
               A[j]=A[j+1];
               A[j+1]=temp;
          endIf
     endFor
endFor
```

Computing number of operations

-for each value of i and j we need

Comparison=1

Assignments=3

Add=3

Total=3+3+1=7

For i=0 inner loop runs upto (n-1) times. Since outer loop runs for n times. Total work done is n*(n-1). Thus, the order of operations required will be, roughly, 7*n^2. Hence it is O(n^2)

```
integer i, j, n;
n=A.length (Array  length )
for (i=0;i<n;i++)
   for(j=0;j<n-i-1;j++)
          if(A[j]>A[j+1])
              temp=A[j];
              A[j]=A[j+1];
              A[j+1]=temp;
          endIf
     endFor
endFor
```

For n=5 Analysis

Outer loop value i	Inner loop runs up to
i=0	(n-1)=4
i=1	(n-2)=3
i=2	(n-3)=2
i=3	(n-4)=1
i=4	(n-5)=0
Max- n times	Max-(n-1) times

Rate of growth =n*(n-1)

Total operations=7(n-1)+7(n-2)+……..+7(n-4)+…7x1

Bubble sort requires maximum (n-1) passes

Modified Bubble sort

```
integer i, j, n,swap;
n=A.length (Array  length )
for (i=0;i<n;i++)
      swap=0;
   for(j=0;j<n-i-1;j++)
           if(A[j]>A[j+1])
               temp=A[j];
               A[j]=A[j+1];
               A[j+1]=temp;
               swap=1;
          endIf
     endFor
   if (swap==0)
     break;
endFor
```

Worst case:- Array requires all the **(n-1)** passes, thus the order of work done is **n*(n-1)**=**n^2-n**. Thus, it is **O(n^2)**

Best case:- Array (already sorted) requires only **1** pass, thus the order of work done is **1x (n-1).** Thus, it is **O(n)**

Average case:- Array requires half of the passes (n-1)/2, thus the order of work done is **n x ((n-1)/2).** Thus, it is **O(n^2)**

Find the space complexity of both bubble sorts

```
integer i, j,n;
n=A.length (Array  length )
for (i=0;i<n;i++)
   for(j=0;j<n-i-1;j++)
          if(A[j]>A[j+1])
              temp=A[j];
              A[j]=A[j+1];
              A[j+1]=temp;
          endIf
     endFor
endFor
```

A

```
integer i, j, n,swap;
n=A.length (Array  length )
for (i=0;i<n;i++)
     swap=0;
   for(j=0;j<n-i-1;j++)
          if(A[j]>A[j+1])
              temp=A[j];
              A[j]=A[j+1];
              A[j+1]=temp;
              swap=1;
          endIf
     endFor
   if (swap==0)
     break;
endFor
```

B

Compare the space needed for both the versions

<u>Time memory trade off</u>, important principle in computer science

Find total number Add, Mult, and Assignment operations needed

SQUARE-MATRIX-MULTIPLY(A, B)

$n = A.rows$
let C be a new $n \times n$ matrix
for $i = 1$ **to** n
  for $j = 1$ **to** n
    $c_{ij} = 0$
    for $k = 1$ **to** n
      $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return C

Recurrence relations

- Running time of many recursive algorithms is, naturally, written by using recurrence relations.
- **Recurrence** is equation or inequality that describes a function in terms of its value on smaller inputs.
- E.g. Running Time T(n) of factorial method is given by

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1), & \text{if } n > 1 \end{cases}$$

- There are **3 approaches** to solve the recurrence relations, for obtaining the **asymptotic bounds** on the solutions (Time complexity)
- **1. Substitution method**
- **2. Recursion tree method**
- **3. Master method**

Recurrence relations- Substitution method

- In this method, we **substitute** the value of a term in terms of smaller input size and guess the form of solution and using **induction** find the constants and show that solution works.

```
Fact(n)
{   if (n<=1)
        return 1;
Else
      return n*fact(n-1);
}
```

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1)+1, & \text{if } n > 1 \end{cases}$$

Recurrence relations- Substitution method

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

$$= (T(n\text{-}2) + 1) + 1$$

$$= T(n\text{-}2) + 2$$

$$= (T(n\text{-}3) + 1) + 2$$

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

$$= \ldots\ldots$$

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

$$= \mathrm{T}(1) + (\mathrm{n} - 1) = 1 + n\text{-}1 = n$$

$$\Rightarrow O(n)$$

$T(n) = O(n)$

Prove by induction that $T(n) \le c * n$

Using induction

1. Assume that it is true for $T(1)$
2. Assume it is true for some n
3. prove that it is true for $c*n$

$$
\begin{aligned}
T(n) &= T(n-1)+1 \\
&\leq T(cn\text{-}1)+1 \quad \textit{putting } n = cn \\
&\leq (T(cn\text{-}2)+1)+1 \\
&\leq T(cn\text{-}2)+2 \\
&\ldots\ldots \\
&\leq T(cn\text{-}(cn\text{-}1))+(cn\text{-}1) \\
&\leq T(1)+(cn\text{-}1) \\
&\leq 1+cn\text{-}1 \\
&\leq c*n \quad \textit{hence proved}
\end{aligned}
$$

Recurrence relations- Substitution method

Binary search

```
BS(a, i, j, x)
{  mid=(i+j)/2;
   if (a[mid]==x)
       return mid;
Else
    if (a[mid]>x)
        BS(a,l,mid-1,x);
    else
        BS(a,mid+1,j,x);
}
```

$$T(n)=\begin{cases}1 & \text{if } n=1\\ T(n/2)+C & \text{if } n>1\end{cases}$$

$C -$ constant time needed for comparison and computing mid can be taken as 1

$$T(n)=\begin{cases}1 & \text{if } n=1\\ T(n/2)+1 & \text{if } n>1\end{cases}$$

Array a

Recurrence relations- Substitution method

$$T(n) = T(n/2) + 1$$

$$= (T(n/4) + 1) + 1$$

$$= 2 + T(n/4)$$

$$= 2 + (T(n/8) + 1)$$

$$= 3 + T(n/8)$$

....

$$= k + T(n/2^k) \qquad \max \ \mathit{vaue\ of\ k\ can\ be}\ \log n$$

$$= \log n + T(n/2^{\log n})$$

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

$$= \log n$$

$$\Rightarrow O(\log n)$$

Recurrence relations- Substitution method

To prove that our guess is correct $T(n) = O(\log n)$

we have to prove that $T(n) \leq c * \log n$ using induction

$T(n) = T(n/2) + 1$ known, since $T(n/2) \leq c * \log n/2$

$$\leq c * \log n/2 + 1$$

$$= c * (\log n - \log 2) + 1$$

$$= c * \log n - c * \log 2 + 1$$

$$\boxed{T(n) \leq c * \log n}$$

Recurrence relations- Recursion tree method

- ***Recursion tree***
- Each node represents cost of a **single subproblem** somewhere in the set of recursive function invocation.
- We sum the costs within each level of the tree to obtain a set of **per level costs**.
- We then sum all the per level costs to find the **total cost** of all the levels of recursion.

Recursion tree method

$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$$

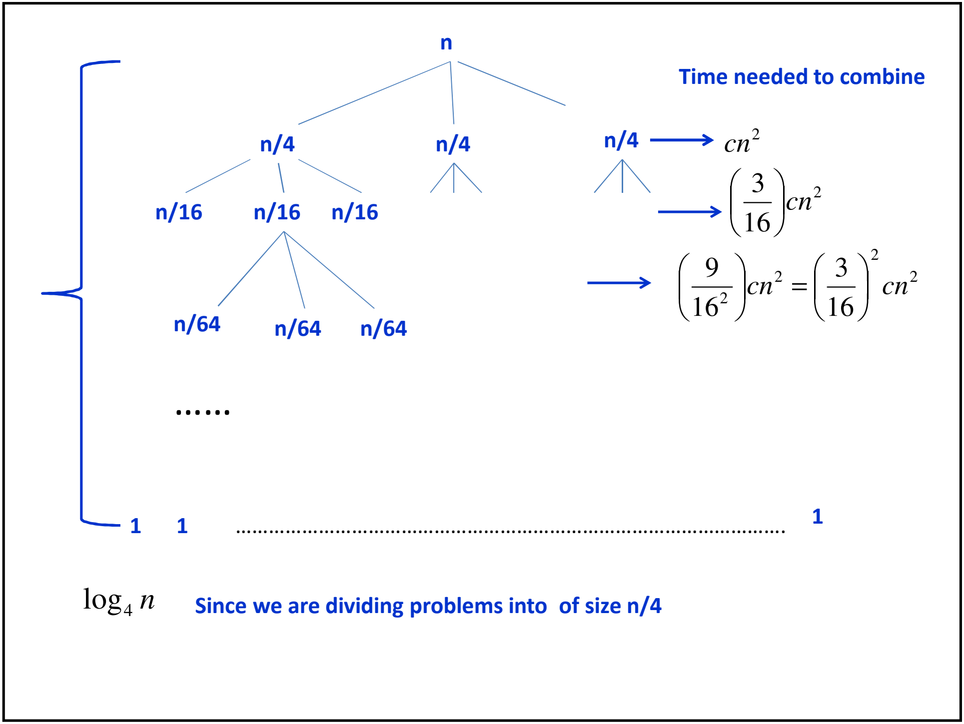
We can ignore **floor** operation since it is insignificant in finding time complexity again when n is divisible by 4, **floor(n/4)=n/4**

$$T(n) = 3T(n/4) + cn^2$$

Dividing a problem of size n into 3 sub-problems of size n/4 each

We need this much time to combine sub problems

https://www.youtube.com/watch?v=JPAA1FbM7jk



n
Time needed to combine
n/4
n/4
n/4
cn^2
n/16
n/16
n/16
$\left(\frac{3}{16}\right)cn^2$
$\left(\frac{9}{16^2}\right)cn^2 = \left(\frac{3}{16}\right)^2 cn^2$
n/64
n/64
n/64
......
1
1
1
$\log_4 n$
Since we are dividing problems into of size n/4

Sum of time required at each level will be

$$T(n) = cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \left(\frac{3}{16}\right)^3 cn^2 \ldots\ldots\ldots\ldots + 1$$

$$= cn^2 \left\{ 1 + \left(\frac{3}{16}\right) + \left(\frac{3}{16}\right)^2 + \left(\frac{3}{16}\right)^3 + \ldots\ldots\ldots \right\}$$

Geometric series

$$1 + r + r^2 + r^3 + \ldots\ldots = \frac{1}{1-r} \quad for\ r < 1$$

$$T(n) = cn^2 \left(\frac{1}{1-(3/16)} \right)$$

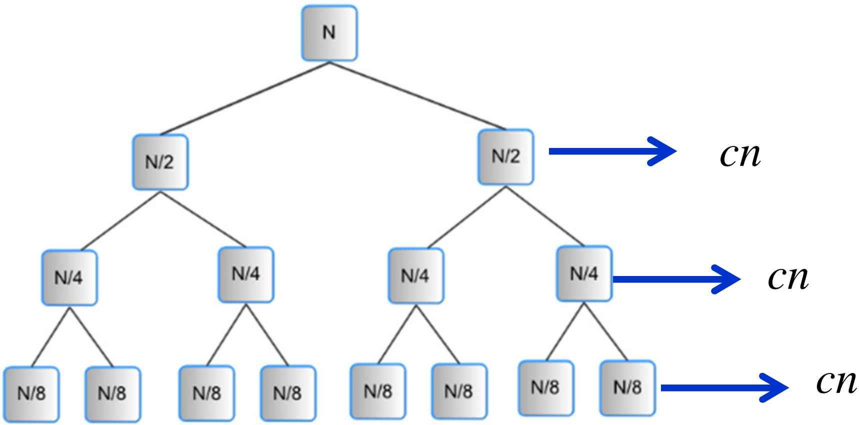
$$= cn^2 (16/13)$$

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

Recursion tree method –merge sort

$T(n) = 2T(n/2) + cn$

Time needed to combine



Since we are dividing the problem into 2, tree will have height of log2 to the base 2

https://www.youtube.com/watch?v=C4JjXc0htp0

Recursion tree method –merge sort

Since there are $\log_2 n$ levels are there and we need cn time to each layer, thus total running time will be

$$T(n) = cn \log_2 n$$

$$\therefore T(n) = O(n \log n)$$

Recursion tree method –Binary search

$$T(n) = T(n/2) + 1$$

tree	Time needed for comparison
n	1
|	|
n/2	1
|	|
n/4	1
|	|
n/8	1
|	|
.	.
.	.
.	.

Log n

T(n)=1xlog n

Thus

T(n)=**O(logn)**

Master method/theorem

- This is the direct method of solving the recurrence relations by remembering 3 cases, of the form

$$T(n) = aT(n/b) + f(n)$$

$$where\ a \geq 1 \text{ and } b > 1 \text{ are constants and } f(n) \text{ is positive}$$

- The problem of size ***n*** is divided into ***a*** sub-problems of size **n/b**.
- The **a** sub-problems are solved recursively , each in time ***T(n/b)***
- The cost of **dividing** the problems and **combining** the results of the sub-problems is described by function ***f(n)***

Master method/theorem- case 1

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is positive

$\boxed{\text{Case1:}}$ if $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$

Then $T(n) = \Theta(n^{\log_b a})$

In each of the case we are comparing $f(n)$ and $n^{\log_b a}$.

In case 1 $n^{\log_b a}$ is larger than $f(n)$

Master method/theorem- case 1 example

$$T(n) = 9T(n/3) + n$$

$$a = 9, b = 3 \text{ satisfies a} \geq 1 \text{ and b} > 1$$

$$\text{check f(n)} = \text{O}(\text{n}^{\log_b \text{a-}\in}) \text{ foe some} \in > 0$$

$$\log_b \text{a} = \log_2 9 = 2$$

$$\text{n} = \text{O}(\text{n}^{2\text{-}\in}) \quad \text{ if} \in = 1 \text{ condition can satisfy}$$

$$\text{n} = \text{O(n)} \quad \text{for} \in = 1$$

$$\text{Thus, can conclude that T(n)} = \Theta(\text{n}^{\log_b \text{a}}) = \Theta(n^2)$$

Means that T(n) is lower and upper bounded by **n^2**, i.e **T(n)=O(n^2)** and **T(n)=Omega(n^2)**

Master method/theorem- case 2

$$T(n) = aT(n/b) + f(n)$$

$where\ a \geq 1$ and $b > 1$ are constants and $f(n)$ is positive

Case2 : if $f(n) = \Theta(n^{\log_b a})$

Then $T(n) = \Theta(n^{\log_b a} \log n)$

In this case we are comparing $f(n)$ and $n^{\log_b a}$.

In case 2 : $n^{\log_b a}$ is of same as $f(n)$

Master method/theorem- case 2 example

$T(n) = T(2n/3) + 1$

where $a = 1,\ b = 3/2, f(n) = 1$; thus condition $a \geq 1$ and $b > 1$ satisfied

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$, since log 1 to any base is 0.

$f(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_{3/2} 1}) = \Theta(n^0) = \Theta(1)$

$1 = \Theta(1)$ is satisfied

Thus,

$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n)$

$\boxed{T(n) = \Theta(\log n)}$

Master method/theorem- case 3

$$T(n) = aT(n/b) + f(n)$$

$where\ a \geq 1$ and $b > 1$ are constants and $f(n)$ is positive

Case3 : if $f(n) = \Omega(n^{\log_b a + \epsilon})$ for $\epsilon > 0$

and $af(n) \leq c \cdot f(n)$ for $c < 1$, and $\forall n$

then $T(n) = \Theta(f(n))$

Master method/theorem- case 3 example

$T(n) = 3T(n/4) + n\log n$

$a = 3,\ b = 4,\ f(n) = n\ \log\ n$

$\log_b a = \log_4 3 = 0.793$

$f(n) = n\log n\ = \Omega(n^{\log_b\ a+\epsilon}) = \Omega(n^{0.793+\epsilon})\ \text{for} \epsilon > 0$

if we put $\epsilon = 0.2$, then $(0.793 + 0.2) \approx 1$

$\therefore n\ \log n\ = \Omega(n^1) = \Omega(n)$ – which is lower bound

checking for next condition

$af(n/b) = 3f(n/4) = 3 \cdot (n/4) \cdot \log(n/4) \leq c \cdot f(n)\quad \forall n,\ c < 1$

$(3/4) \cdot \mathrm{n} \cdot \log\ (\mathrm{n}/4) \leq (3/4) \cdot n \cdot \log n \quad \text{for c} = 3/4$

thus $af(n/b) \leq c \cdot f(n)$ is also satisfied.

Hence $T(n) = \Theta(f(n)) = \Theta(n\log n)$

Proof of correctness of algorithms

Methods

1. **Loop invariants**
2. **Proof by counter example**

- In computer science, a **loop invariant** is a **property** of a program loop that is true before (and after) each iteration.
- It is a logical assertion (belief), sometimes checked within the code by an assertion. (wikipedia)
- Loop invariants characterizes the deeper purpose of the loop beyond the details of this implementation.
- They used to provide **correctness** of algorithm using loops.

Proof of correctness of algorithms

```
int SumArray( A, n)
{
   int i=0;sum=0;
   //sum will have addition of A[0,...,0]-initialization
    for i=1to n
           sum=sum + A[i];
      // sum will have addition of A[1,...,i]
     end
    // sum will have addition of A[1,...,n]
return sum;
}
```

Proof of correctness of algorithms- sum of array A

Loop invariant:- At the start of iteration **i th** of the loop, the variable **sum** should contain the sum of the numbers from the subarray A[1: (i-1)].

1. **initialization:**- At the start of the first loop the loop invariant states: 'At the start of the first iteration of the loop, the variable *sum* should contain the **sum of the numbers** from the subarray **A[0:0]**, which is an empty array. The sum of the numbers in an empty array is 0, and this is what sum has been set to.
2. **Maintenance:** Assume that the **loop invariant holds (true)** at the start of *iteration i.* Then it must be that **sum** contains the sum of numbers in subarray ***A[1: (i-1)].*** In the body of the loop we add **A[i]** to **sum**. Thus, at the start of **iteration i+1**, **sum** will contain the sum of numbers in **A[1: i],** which is what we needed to prove.

Proof of correctness of algorithms- sum of array A

3. Termination:

When the **for**-loop terminates when i=n+1. Now the **loop invariant gives:** The variable **sum** contains the sum of all numbers in complete array **A[1:n]**. This is exactly the value that the algorithm should output, and which it then outputs. Therefore the algorithm is **correct**.

Proof of correctness of algorithms

- Proof by counter example

Prove or disprove $\lceil x+y \rceil = \lceil x \rceil + \lceil y \rceil$

counter example : - x $= 1/2$ and y $= 1/2$

$$\lceil 1/2 + 1/2 \rceil = \lceil 1 \rceil = 1$$

$$\text{But } \lceil 1/2 \rceil + \lceil 1/2 \rceil = 1 + 1 = 2$$

Hence algo is not correct since $\lceil x+y \rceil \neq \lceil x \rceil + \lceil y \rceil$

Proof of correctness of algorithms

Any integer is sum of squares of two integers

Counter example :- 3