Data structures.

Multiplication: we have well-defined rules for transforming an input (two nos) into output (one no).

Input:. Nos. x & y n digit non-ve integers.

```
        5 6 7 8
      x 1 2 3 4
    -------------
        2 2 7 1 2   <- <= 2n operations (Per row)
      1 7 0 3 4 0
    1 1 3 5 6 0 0
    5 6 7 8 0 0 0
  -----------------
  7 0 0 6 6 5 2   <- addition needs <= 2n^2 operations
```

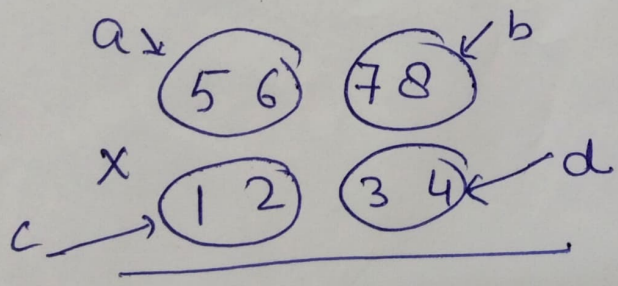
n-rows

Last row's length can be 2n.

one col^m add^n n:

total operations $\leq 4n^2$.

Anatoly Karatsuba (23-old Mathematician 1960)



Step-1. $a * c = 56 \times 12 = 672$] α_1

Step-2. $b * d = 78 \times 34 = 2652$] α_2

Step-3. $(a+b)\cdot(c+d) = 134 \times 46 = 6164$] α_3

Step-4. $\alpha_3 = \alpha_4 - \alpha_2 = 2840$] α_4

$10^4 672 + 10^2 2840 + 2652 = 70066552$

A Recursive Approach :—

$$x = 10^{n/2} a + b$$
$$y = 10^{n/2} c + d$$

$$xy = (10^{n/2} a + b)(10^{n/2} c + d)$$

$$= 10^{n}(ac) + 10^{n/2}(bc + ad) + bd$$

assumption n is power of 2.

if $n = 1$

{
compute x.y
}

else

{
a, b := first & second halves of x
c, d := " " " y

recursively compute
ac, ad, bc, bd

Compute : $10^{n} ac + 10^{n/2}(bc + ad) + bd$

$3n^{\log_2 3}$

Asymptotic Notation -

Asymptotic Analysis: - helps us differentiate b/w better and worse approaches for multiplication.

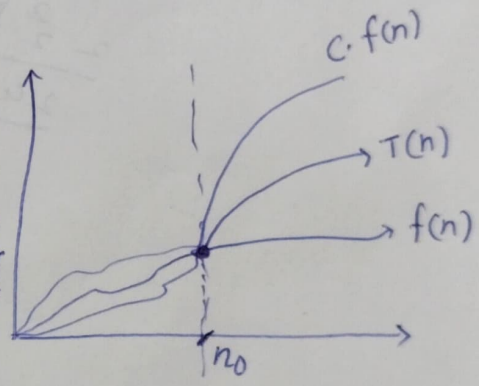
High level Idea :-

Suppress constant factor (too system-dependent) and lower-order terms (irrelevant for large inputs).

Big 'O' notation :

$T(n) = O(f(n))$

$= \{ f(n) : \exists\ c>0, n_0>0,\ s.t.\ T(n) \le c \cdot f(n)\ \forall\ n \ge n_0 \}$



Ex:-

① $2(n^2)$

$O(n^3)$

② $T(n) = 5n+1$,

$T(n) = O(n)$ $\quad c=6,\ n_0=1$

or

$T(n) \le 5n+n$, $n \ge 100$

$T(n) \le 6(n)$, $n \ge n_0 = 100$

at most upperbound

Ex: 3:

$T(n) = 10n^2+5n+6$

$T(n) \le 10n^2+5n^2+6n^2$

$\le 21n^2$

$T(n) \in O(n^2)$

$n \ge 1$

$c = 21$

1-

...would be ignored.

$$T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

$k \geq 0$, a_i are real nos. (.)

↓ non-negative Int.

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

$$T(n) \leq |a_k| n^k + |a_{k-1}| n^{k-1} + \cdots + |a_1| + |a_0|$$

Now, $n \geq 1$

$n^k \geq n^i \quad \forall i = \{0, \cdots k\}$

$$\leq (|a_k| + \cdots |a_0|) n^k$$

$$\leq c n^k$$

$$T(n) = O(n^k), \quad n \geq n_0 = 1$$

- Asymptotic Analysis - classifying the behavior of functions

Classes of functions:

Lect-2

Features:..

1- $5n^2 + 2n + 1$ } should get into same class
$10n^2 + 3$ }

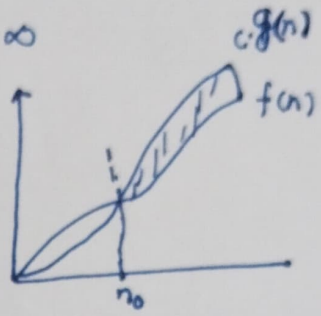
2. constant multiplier should be ignored.

2. Determine behavior when $n \to \infty$

Big-O (upper bound)

$O(g(n))$

$= \{ f(n) \mid$ there exists $c > 0, n_0 > 0,$ such that $f(n) \leq c \cdot g(n) \}$



Ex:- 1

$f(n) = 5n+1$, $g(n) = n$

$f(n) \leq 5n + n$, $n \geq 100$

$f(n) \leq 6n$

$\boxed{f(n) = O(n)}$

Ex:- 2

$f(n) = 10n^2 + 5n + 6$

$f(n) \leq 10n^2 + 5n^2 + 6n^2$ $n \geq 1$

$f(n) \leq 21n^2$

$\boxed{f(n) = O(n^2)}$

Ex:- 3.

$f(n) = 6 2^n + n^2$

$f(n) \leq 6 2^n + 2^n$, $n \geq 5$

$f(n) \leq 7 2^n$

$\boxed{f(n) = O(2^n)}$

Big Ω notation :- (lower bound) (at least)

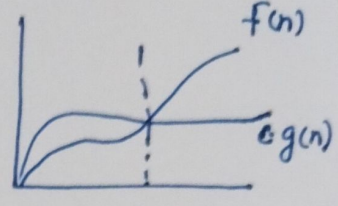
$$\Omega(g(n)) = \{ f(n) \mid \text{there exists } c>0, n_0>0 \text{ such that } f(n) \geq c \cdot g(n) \ \forall\ n \geq n_0 \}$$

Ex:- 1

$f(n) = 10n^2 + 5n + 6$

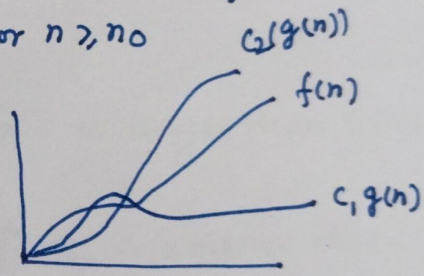
$f(n) \geq n^2$

$\boxed{f(n) = \Omega(n^2)}$



Θ - notation.

$$\Theta(g(n)) = \{ f(n) \mid \text{there exist } c_1, c_2, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for } n \geq n_0 \}$$



Ex:- 1

$f(n) = 10n^3 + 5n^2 + 7$

$10n^3 \leq f(n) \leq (10+5+7)n^3$

$10n^3 \leq f(n) \leq 22n^3$ $\quad n \geq 1$

$\boxed{f(n) = \Theta(n^3)}$

2. $f(n) = 5n^3 + n\log n$

$5n^3 \leq f(n) \leq 6n^3$ $\quad n > \log n$

$\boxed{f(n) = \Theta(n^3)}$, $n_0 = 1$

Ex:- $f(n) = 5n\log n + 10n$

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

$$\boxed{\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))}$$

Ex:- $T(n) = 5n^3 + 4n + 2$

$T(n) = O(n^3)$ $\quad$ $T(n) = O(n^3)$

$T(n) = O(n^4)$ $\quad$ $T(n) = O(n^2)$

$\Rightarrow \boxed{T(n) = \Theta(n^3)}$

Summarization :-

$T(n) \in O(g(n)) \approx$ T is no larger than $g(n)$ $\quad (\leq)$

$\in \Omega(g(n)) \approx T(n)$ – smaller than $\quad (\geq)$

$\in \Theta(g(n)) \approx$ nearly same as $\quad (=)$

Ex:- $T(n) = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$

$T(n) = \sum_{k=1}^{n} k \leq \sum_{k=1}^{n} n$

$= n^2$

$\Rightarrow \boxed{T(n) = O(n^2)}$

$T(n) = \sum_{k=1}^{n} k \geq \sum_{k=\frac{n}{2}+1}^{n} k$

$\geq \sum_{k=n/2+1}^{n} n$

$= \frac{n^2}{4}$

$\boxed{T(n) = \Theta(n^2)}$ $\quad$ $\boxed{T(n) = \Omega(n^2)}$

Results:-

$$T_1(n) * T_2(n) = \Theta(g_1(n) \cdot g_2(n))$$

Lemma-1:- Let $T(n) = \max\{f(n), g(n)\}$, $\forall n \geq 1$,

Then, $T(n) = \Theta(f(n) + g(n))$

Proof:-

for any n:

$$\max(f(n), g(n)) \leq f(n) + g(n) \quad \text{---(1)}$$

$$2 * \max(f(n), g(n)) \geq f(n) + g(n) \quad \text{---(2)}$$

Putting these two inequalities together,

$$\frac{1}{2}(f(n) + g(n)) \leq \max(f(n), g(n)) \leq f(n) + g(n)$$

$$\boxed{\max(f(n), g(n)) = \Theta(f(n) + g(n))}$$

$$\boxed{T_1(n) + T_2(n) = \Theta(f(n) + g(n)) = \max(f(n), g(n))}$$

Analyzing Algorithms :

Framework for analysis:-

Give a problem 'P' and $A_1, \ldots A_n$ algorithm to solve.

Question:- Which is more viable?

RANDOM ACCESS Machine (RAM). Process + Memory

- Instructions are executed one after another
- No concurrent operation
- Model contains – basic arithmetic operations, s.t. add, subtract, multiply, ...
  - data movement (load, store, copy)
  - conditional control
- Data types - Integer & floating point.

Algorithm:- An abstract computational procedure which takes some value(s) as input and produces some value(s) as output.

Program:- Expression of Algorithm (concrete)

Ex:

```
for i = 1 to n
    C[i] = A[i] + B[i]
end
```

```
1. i = 1
2. if i > n goto 9
3. x = A[i]
4. y = B[i]
5. z = x + y
6. C[i] = z
7. i = i + 1
8. goto 2
9. Exit
```

(lines 3–6: body of loop)

Analysis:-

Statement-1 .. executed 1 times

3-6 - n times

7 - n times

8 - n times

Loop test-2 - n+1 times

Total time:-

instructions in body → $Cn + n + n + n + 1 + 1$

$= (C+3)n + 2$

General Rules:-

Rule:- 1. For loops.

RunningTime: running time of statements inside the for loop times number of iterations.

```
A[n], t
for i ← 1 to n
    if A[i] = t.
        return TRUE
end for
return False
```

Searching in one array

```
A[n], B[n], t
for i ← 1 to n
    if A[i] = t
        retur TRUE
    endif
end for
return False
same for B
```

Rule-2 - Nested for loop :- Analyse a inside out

Total running time of a statement inside a group of nested loops is running time of statement multiplied by the product of size of all the for loops :.

Ex :-

```
for i ← 1 to n
    for j ← 1 to n
        k = k+1
    end for
end for
```

$O(n^2)$

Checking for a common element :

```
for i ← 1 to n
    for j = 1 to n
        if A[i] = B[j]
            return True
    end for
end for
Return false
```

$O(n^2)$

Checking for duplicates

```
for i ← 1 to n
    for j ← i+1 to n
        if A[i] = A[j]
            return True
    end
end
return false
```

$O(n^2)$

Rule : 3 . Consecutive statements :

Add them means the maximum

```
if (condition)
    S1
else
    S2
```

never more than test + max(S_1, S_2)

Recursion & Recurrence Relation

```
Fib(n)
1  if n ≤ 1
2     return n
   else
3     return Fib(n-1) + Fib(n-2)
```

- If there are function calls, they must be analyzed first

$T(n) = T(n-1) + T(n-2) + 2$ ↑ (1 unit of ① & 1 unit for addition in ③)

Time complexity -

Ex:- 1

```
temp = 0
for ( i=1; i*i ≤ n; i++)
        temp ← temp+1
```

Length of the loop ⇒ $i^2 \leq n \Rightarrow i \leq \sqrt{n}$

$\boxed{T(n) = O\sqrt{n}}$

Ex:- 2

```
for (i=n/2; i ≤ n; i++)                    → length → n/2
    for (j=1; j+n/2 ≤ n; j=j++)            → length → n/2
        for (k=1; k ≤ n; k=k*2)
```

⇓

Let 'm' th term.

$k=1,\ k=1\times 2,\ k=1\times 2^2,\ \ldots\ m^{th}$

$T_m = ar^{m-1} \leq n$

$k = 1\times 2^{m-1}$

$\frac{k}{2} \times \frac{1}{2} \leq n$

$k \leq 2^{n+1}$

$\boxed{\log_2 k \leq n}$

$\boxed{O(n^2 \log_2 n)}$

Ex:- 3

```
if n ≤ 1
    return n
else
    return f(n-1) + f(n-2)
```

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

- Recurrences / Recurrence Relation -

- An equation / inequality that describes a function in terms of its value.

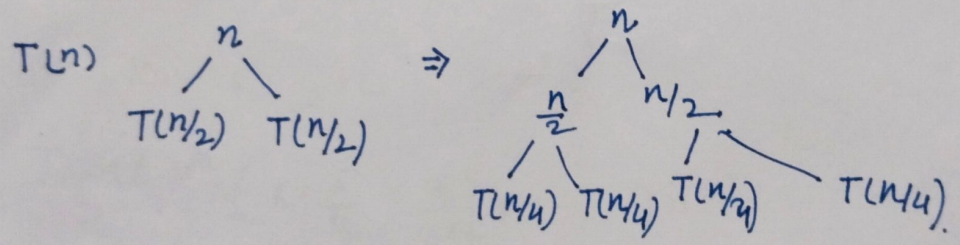
$$T(n) = 2T(n/2) + \Theta(n)$$

- Recursion Tree Method:-

- Node represents the cost of single subproblem
- cost per level = sum the cost of each node
- Total cost = sum the cost of each level

General Method:- Expand.

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



Height of tree ⇒ longest path.

$$n + n/2 + n/4 + \cdots \cdots 1$$

let a level h

$$T_h = n \times \frac{1}{2}^{h-1} = 1$$

$$\boxed{h = \log n}$$

Totalcost

= sum (level-by-level)

= $n * \log n$

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

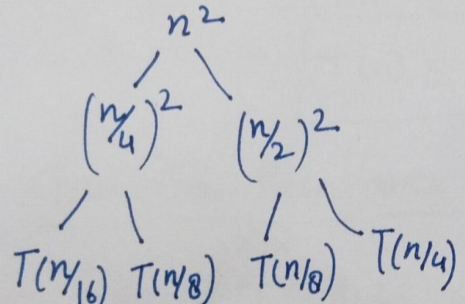
G.P

$S_n = \frac{a(r^n - 1)}{(r-1)}$, $r \neq 1$

$= \frac{a(1-r^n)}{(1-r)}$ $r < 1$

$S_\infty = \frac{a}{1-r}$ $0 < r < 1$

Ex:- $T(n) = T(n/4) + T(n/2) + n^2$



# of leaves $\leq n$

Total = $\left(1 + \frac{5}{16} + \frac{25}{256} + \cdots\right) n^2$

$\leq 2n^2$

$\boxed{T(n) = O(n^2)}$

Ex:-3 . $T(n) = 3T(n/4) + cn^2$

Height = $\log_4 n$

leaves: $\Theta(n^{\log_4 3})$

Karatsuba Algo.

Multiply x & y.

if $n = 1$

compute xy

else

a, b: 1st & 2nd halves of x

c, d: ——— y

compute

ac

ad

bc

bd

compute $10^n ac + 10^{n/2}(ad+bc) + bd$

end.

$$T(n) \leq 4T(n/2) + O(n)$$

Standard Recurrence format:

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + O(n^d)$$

a: # of recursive calls

b: input size shrinkage factor (each call breaks into subproblems of size b)

d: exponent in running time of the combine step

$a \geq 1, b > 1$ | $d \geq 0$ ($O(n^d)$ work out side recursive calls)

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d : \text{Case 1} \\ O(n^d) & \text{if } a < b^d : \text{Case 2} \\ O(n^{\log_b a}) & \text{if } a > b^d : \text{Case 3} \end{cases}$$

...duction

Ex:- $T(n) \leq 4(T(n/2) + O(n)$ (Int. multiplication)

$a = 4,$
$b = 2$
$d = 1$

$\boxed{a > b^d}$

$T(n) = O(n^{\log_2^4})$

$= O(n^2)$

Lo

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

$a = 3$
$b = 4$
$d = 2$

$\boxed{a < b^d}$

$\boxed{T(n) = O(n^2)}$

Ex:- $T(n) = 4T(n/2) + O(n^2)$

$\Rightarrow a = 4, b = 2, d = 2$

$\boxed{a = b^d}$

$\boxed{T(n) = O(n^2 \log n)}$

- Recursion -

- Process of defining defining a problem interms of itself.

Closed relation with reduction -

- Reduction : Algo. design paradigm
  - Reducing a problem X into an other problem (or set of problems) Y means write an algorithm for X, using an algorithm or Y as subroutine.

Loose definition :-

- If the given instance of the problem is small, just solve it.
- Otherwise reduce the problem into one or more simpler instances of the same problem.

Repetitive Algorithm :-

- → use iteration
- → use recursion

Iterative :-

$$Fact(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times n-1 \cdots \times 3 \times 2 \times 1 & \text{if } n>0 \end{cases}$$

Rec.

$$Fact(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times Fact(n-1) & n>0 \end{cases}$$

fact(3) = 3 * fact(2)

↓ ↑

fact(2) = 2 * fact(1)

↓ ↑

fact(1) = 1 * fact(0)

↓ ↑

fact(0) = 1

Design Methodology :-

- Every recursive call must either solve a part of the problem or reduce the size of the problem.

- Solve the Problem :- Basecase

reduce — general case.

1- Determine the base case
2. Determine the general case
3. Combine the base and general case into an algorithm.

$L = \{x_1, \cdots x_n\}$ x_i is an integer

```
Max(L)
{ L = {x1, ... xn}
  if |L| = 1 , return (x1)
  else
  |  L' = L - {x1}
  |  y = max(L')
  |  if (x1 > y) then
  |      return x1
  |        else y
  |_
```

$\{5, 8, 3, 1, 2, 12\}$

$$T(n) = T(n-1) + 1 \quad , n > 1$$
$$= 0 \quad , n = 1$$

$O(n)$

```
if |L| = 1 , return (x1)
  split L into 2 non-empty sets L1 , L2
      y1 = max(L1)
      y2 = max(L2)
    if (y1 ≥ y2) return y1
        else return y2
```

$$T(n) = T(k) + T(n-k) + 1 \quad n > 1$$
$$= 0 \quad n = 1$$

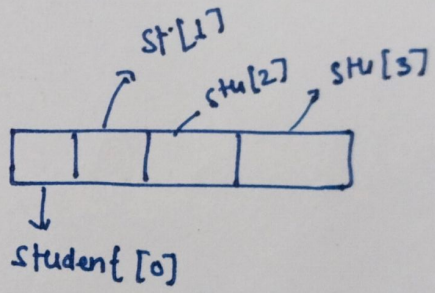
$= O(n-1)$

Array.

- Single name to homogeneous collection of data
- Same type:- predefined equal amount of memory allocated to each one of them.
- Individual element have an associated index that depends on array dimension.
- can be used to process / store a fixed number of data elements that all have the same type.

Memory storage:-

int student[4]



Two-dim. Array.

int A[3][5]

Rows { A[0][0] - - - .. A[0][4]
A[1][0]
A[2][0] ... A[2][4]

Columns.

Row-Major:- consecutive elements of a row reside next to each other

Col^m Major:- _____ a column - - -

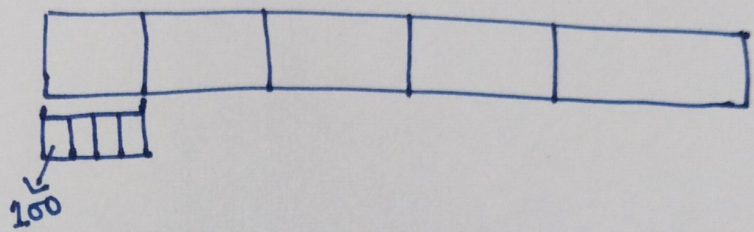
1 Array - Address computation

Address of i^{th} element :

$$Loc(A[i]) = B + W * (i - LB)$$

W: # of bytes to store
- single element

B: - Base address
A[LB]



i: subscript of element whose addr is to be found

LB: lower bound of subscript.

B = 100
W = 4
LB = 0
A[1] = 104

Ex!-2 : LB = 5 , LOC(A[LB]) = 1200 , W = 4,
LOC(A[8]) = ?

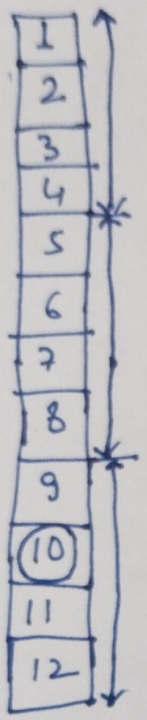
⇩

loc(A[8]) = 1212
W = 4
LB = ?

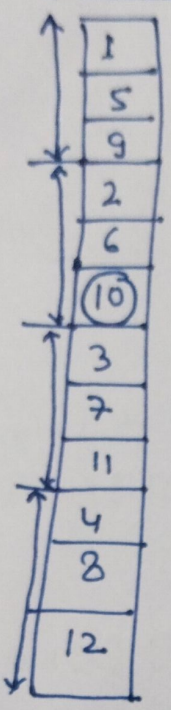
2-D Array.

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

Row-Major



Col^m-Major



Row-Major :-

B : Base address
i : row subscript
J : col^m.
w : # of bytes for single
LBR : LB for row
LBC : LB for C
n : # of row
M : # col^m

$$B + w\left(n * (i - L_1) + (J - L_2)\right)$$

Ex:- A[5][5], Row-Major
w = 4, B = 49

A[4][3] = 141

$$A[2][1] = 120 + 4\left(3(2-0) + (1-0)\right)$$
$$= 120 + 4(3 * 2 + 1)$$

Operation on array

1. Traversal
2. Search - Linear & Binary
3. Insertion
4. Deletion
5. Sorting
6. Merging

(1) Traversal -

```
for i ← 0 to n-1
|   Print A[i]
end
```

O(n)

(2)

```
min = A[0]
for i ← 1 to n-1
|   if min > A[i]
|   |   min ← A[i]
|   end
end
return min
```

(3) Search.

```
for i ← 0 to n-1
  if A[i] == num
      return i
return '-1'
```

- Insertion - Ins(A, n, k, j)

```
for i ← n-1 to j
    A[i+1] ← A[i]        ▷ shift elements to right
end
A[j] ← k                 ▷ Insertion
n ← n+1                  ▷ Increment of size by 1
```

Deletion:

Del (A, n, j)

```
temp ← A[j]
for i ← j to n-2
    A[i] ← A[i+1]        ▷ shifting towards left.
n ← n-1                  ▷ decrement
return temp              ▷ return.
```

- Program Safety -

— A program is unsafe if it executes with any exceptional condition which would cause its execution to abort.

— So for, only division and modulus are potentially unsafe, ~~since~~

— Accessing an element of an array for which no space has been allocated : Error (runtime error)

~~Proving Correctness~~:

= What properties do we want on an algorithm?

— correctness :- It has to solve correctly all instances of the problem.

— Efficiency :- The performance (time and memory) has to be adequate.

Proving Correctness

By — counter example (Indirect proof)
— Induction (Direct proof)
— loop Invariant

* we want to prove that an algorithm will always return the desired output for all legal instances of the problem.

(i) Proof by counter Example:

- Get a counter example for which something is not true

⇒ Getting counter example (sometimes) is not so easy.

⇒ If you don't succeed in finding counter example, it does not mean that "Algorithm is correct".

(ii) Proof by Induction :-

- Mathematical Induction is very useful for proving the correctness of recursive algorithms.
- In case of iterative approach, it will not be easy to find general case.

(iii) Proof by loop Invariant

Loop Invariant :- A condition that is necessarily true immediately before and immediately after each iteration.

Loop Invariant must satisfy following conditions:-

(i) Initialization:- The invariant is true prior to the loop entry

(ii) Maintenance:- Must be # of loop traversal ⇒ [If it is true before an iteration of the loop, it remains the true before the next iteration]

(iii) Termination:- It Must imply desired condition on termination.

Process for loop Invariant

Formal

1- Define loop invariant
2- Initialization
3. Maintenance
4- Termination

Informal

1- find P, a loop invariant
2. show the base case for P
3. use induction to show the rest.

* How To use Induction *

Ex:-1.

```
x = C1 ; y = 0;
While (x > 0)
{
    x ← x-1
    y ← y+1
}
```

Prof:- $y = C_1$, when loop terminates

Strategy:- compute state after 1st iteration, then 2nd, - and after a last you will get $y = C$

INDUCTION:-

Base: Induction hypothesis true at initialization (on loop entry)

Ind.: Assume hold after k iteration, prove it holds for k+1.

- loop entry $x = C, y = 0$
- After 1st iter. $x = C-1, y-1$
- :

} Inductive hypothesis $x+y = C$

Base: $x+y = C$ true

Ind:
- $x = C-k, y = k \Rightarrow x+y = C$
- $x = C-(k+1), y = k+1 \Rightarrow x+y = C$

Correctness -
- Proved that $y = C$ after loop.
- * Proven after finalite.
  * — $x = 0$, $x+y = C$

Hence, $\boxed{y = C}$

Ex-2

4

```
y=0;
for i ← 0 to n+1
    y ← y+2^i
```

Value of y after loop.

$i=0, y_0=0 \quad = 2^0-1$

$i=1, y_1=1 \quad (0+2^0) = 2^1-1$

$i=2, y=3 \quad = 2^2-1$

$\boxed{y_i = 2^i-1}$: loop invariant

→ Base case:

$i=0, y_0 = 2^0-1 = 0$

- Ind.

$y_i = 2^i-1 \quad -①$

$y_{i+1} = y_i + 2^i = 2^i+2^i-1 = 2^{i+1}-1$ □

Ex-3 Binary Search:-

I/P: Sorted array A, n: Array size, Key or b

O/P: $\overset{\rightarrow b}{Key} \in A$

```
(1)  l ← 1, u ← n        ▷ index for lower & upper
(2)  While "no output" do
(3)      if u < l
(4)          output "b not present"
(5)      else
(6)          m ← l + (u-l)/2   ⟶ mid Point i.e., m = (u+l)/2
(7)          if A[m] = b, then output b in position m
(8)          if A[m] > b  then  u ← m-1
(9)          if A[m] < b  Then  l ← m+1
```

5

Correctness -

Step-1:- Check correctness of linear part of algorithm

Step-2: Prove correctness of loop using "loop invariant".

Step3: Ensure the loop exits correctly

step-4: The algorithm terminates and the final output is correct.

Proof:-

loop Invariant:

if $Key \in A$, $A[l] \leq Key \leq A[u]$

Base case:

When algorithm begins, $l = 1$, $u = n$

i.e., l & u enclose all values, then Key must be b/w l & u.

Inductive hypothesis:-

let it is true at k^{th} iteration -

$A[l] \leq Key \leq A[u]$

Inductive step:

Case 1: if $A[m] > Key$, then Key is b/w l & m

- $u \leftarrow m-1$

Case: 2 if $A[m] < Key$, —— m & u

- $l \leftarrow m+1$

In either case, we preserve Inductive hypothesis. for next iteration.

* l alway increases & u always decreases, means these values will converge at single location where $\boxed{l = m}$

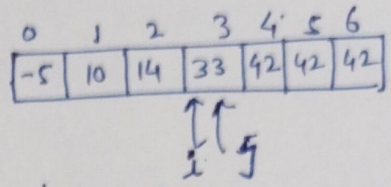
Array Reversal

I/P: A, n

O/P: A'

```
i ← (n-1)/2
J ← n/2
While (i >= 0 && J <= n-1)
    temp ← A[i]
    A[i] ← A[J]
    A[J] ← temp
    i ← i-1
    J ← J+1
```

Exchange



i=3, J=3

temp ← 33

A[3] ← 33

A[3] ← 33

i ← 2

J ← 4

After one iteration

A[i+1, J-1] : reversed

Proof :-

loop invariant -

A[i+1; J-1] is reversed

Base :- upon loop entry (both are at same place)

J-1 < i+1

Inductive case :

At start of K^{th} iteration :

A[i+1, J-1] reversed

Iter. 2 :-

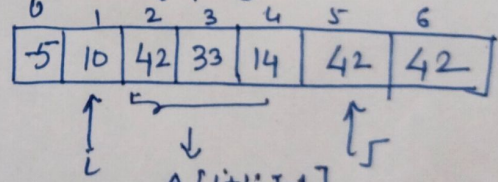
temp ← 14

A[2] ← 42

A[4] ← 14

i ← 1

J ← 5



$(K+1)^{th}$

- loop body swaps A[i] & A[J] and i ↓ and J ↑ } Means - midle part already reversed + extra is reversed now.
- Then at start of K+1, A[i+1; J-1] will be reversed

→ **Correctness :-** - When loop terminates : $\boxed{\begin{matrix} i = -1 \\ J = n \end{matrix}}$ → termination

- Invariant tells : A[i+1; J-1] Reversed
- Therefore, A[0 : n-1] is reversed □

Stack

Atomic data:- - "Data that consist of single piece of information.
- Can't be divided into other meaning pieces of data.

Composite data:-
- broken out into subfields that have meanings.

E.g: Mobile number.

xxxx nnnnnnn
4-digits — six-digits unique to subscriber
Operator's code

Data type:- → Set of data
→ operations that can be performed on data

E.g:-

Type	Data	op.
Integer	$(-\infty, +\infty)$	*, +, −, %, . . .
float	$(-\infty, 0.0 \ldots +\infty)$	+, − . . .

Data structure:-
- An aggregation of atomic and composite data into a set with defined relationship.
- **Structure:-** Set of rules that holds the data together.

Abstract Data Type:-

Abstract → to pull out

Abstraction:- The principle of ignoring details of an object that is not relevent to current purpose.

Concept of Abstraction :-

- We know what data type can do
- How it is done is hidden.

ADTs :-

Defn:
- collection of data
- Operations on the data
- A set of Axioms (pre condition & post conditions) that define the semantic of operation.

Examples: Dynamic sets (DSs)
- operations are provided that change the set.

*Operations on DS: S: set

- New() : ADT
- Insert(S,V) : ADT
- Delete(S,V)
- IsIn(S,V) : boolean

IsIn: Access Method

* Axioms:

- IsIn(New(),V) : false (Is 'v' in new set that we have created (empty))
- ~~IsIn(Insert(S,V),V)~~
- IsIn(Insert(S,v),v) = true (Inserted v. then Access v)
- IsIn(Insert(S,u),v) = IsIn(S,v) , if v ≠ u

Other ADTs Example :-

STACK
QUEUE
LIST

STACK.

- Collection of elements arranged in linear order
- Add and remove an element from top.
-

OPerations supported by Stack ADT

* New() : Creates a new stack
* Push() or Push(S,O) : push element 'o' onto top of stack 's'.
* Pop(S,O) : remove ——
* Top (S) : Return top of stack without removing it.

Some supporting operations call also be defined

isFull ( ) : Check overflow
isEmpty() : underflow
size() : number of objects in stack.

Axioms:

pop (push(S,V)) = S : push of v followed by pop leaves 's' unchanged
Top (push(S,V)) = V : push followed by top gives the v
push(S,V) = S+1 : push increase the size by 1.
pop (S,V) = S-1 : —. decrease ——→

LIFO: Model

Stack Implementation

Array:

- Worst case for insertion & deletion from an array when insert and delete from from the begining: shift elements to the left.
- Best case: for insert and delete is at the end of the array: no need to shift any element.
- Implement push() and pop() by inserting and deleting at the end of an array.

Algorithm - Push

~~S size~~ : S[size]: 1-D array holds stack elements.
Top: pointer that points the topmost element
v : element (data) item) to be pushed

```
If  Top = size-1
        Display "overflow condition"
    else
        Top ← Top+1                    O(1)
        S[Top] ← v
        return v
```

Pop:-

```
If  Top = -1
        Display "underflow condition"
    else
        v ← S[Top]                     O(1)
        Top ← Top-1
        return v
```

Applications of stack:

- Reverse a word
- Undo mechanism in text editor
- Parsing
- Recursive function calls
- Expression Evaluation
- Expression conversion

Use of stack

Infix:

A+B : Infix
+AB : Prefix
AB+ : Postfix

Five Binary Operators:

- Addition
- Subtraction
- Multiplication
- Division
- Exponentiation (↑)

$$\uparrow > * > / > + > -$$

- For exponentiation, the right-to-left rule applies –

  A↑B↑C means A↑(B↑C)

- For other operators of same precedence, the left-to-right rule applies -

  A+B+C ⇒ (A+B)+C

Infix to Postfix

- (A+B)*(C-D) ⇒ AB+CD-*
- A↑B*C-D+E/F ⇒ AB↑C*D-EF/+

INFIX TO POSTFIX (USING STACK)

1- If it is an operand, Output it

2. If it is an opening paranthesis, push it on stack.

3. If it is an operator, then

   i) if stack is empty, push operator on stack

   ii) If top of the stack is opening paranthesis. push operator on stack

   (iii) If it has higher priority than the top of stack., push operator on TOS.

   (IV). Else pop the operator from stack and Output it, repeat step 4

4. if it is a closing paranthesis

   - Pop operators from stack and output them until an opening paranthesis encountered
   - Remove the opening paranthesis and don't add it to output.

I/N	STACK	OIP
~~A+(B*(C-D)/E)~~	—	2
2	*	2
*	*	23
3	/	23*
/	/(	23*
(	/(	23*2
2	/(-	23*2
-	/(-	23*21
1	/	23*21
)	+	23*21-1
+	+	23*21-15
5	+*	23*21-153
*	+*	23*21-153
3	—	23*21-153*+

2*3/(2-1)+5*3 =

Infix to Prefix Conversion

- Reverse the given expression
- Apply Algo. for infix to postfix
- Reverse the resultant expression

Algorithm for Evaluating Postfix notation

- Each operator in a postfix string refers to previous two operands on the string.
- Each time we read an operand, we push it into a stack. When we reach an operator, its operands will then be top two elements on the stack.
- We can ~~perform~~ pop these two elements, perform the indicated operation on them, and push the result on stack.

Algo:-

- Initialize an empty stack
- While token remain in the input stream
  - Read next token
  - If token is number, push it into stack.
  - Else, if token is an operator, pop two tokens off the stack, apply the operator, and push the ~~answer~~ result back into the stack.
- Pop the result off the stack

Ex:-

234+*6-

Postfix to Infix

scanning ⇒ "left to right"

- While token remain in the input stream
  - If it is operand, push it
  - it is an operator, pop, opr1 & opr2, Cancatenate them in proper order (opr1 op opr2)
  - Push result on stack.
- Pop the result off the stack.

Ex:-

ABC-+DE-FG-H+/* ⇒ ((A+(B-C))*((D-E)/((F+H)+H))

Prefix to INfix

Same as above, EXcept either scanning from right to left
or Reverse the expression and do left to right scanning.

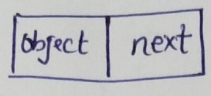
*+A-BC/-DE+-FGH

⇒ Same as above

Linked List

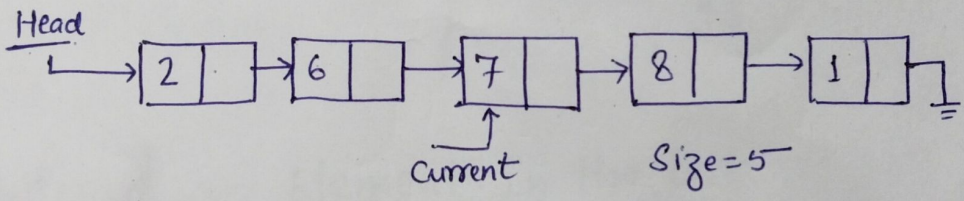
- various cell of memory are not allocated consecutively in memory.

* Create a structure called "Node"



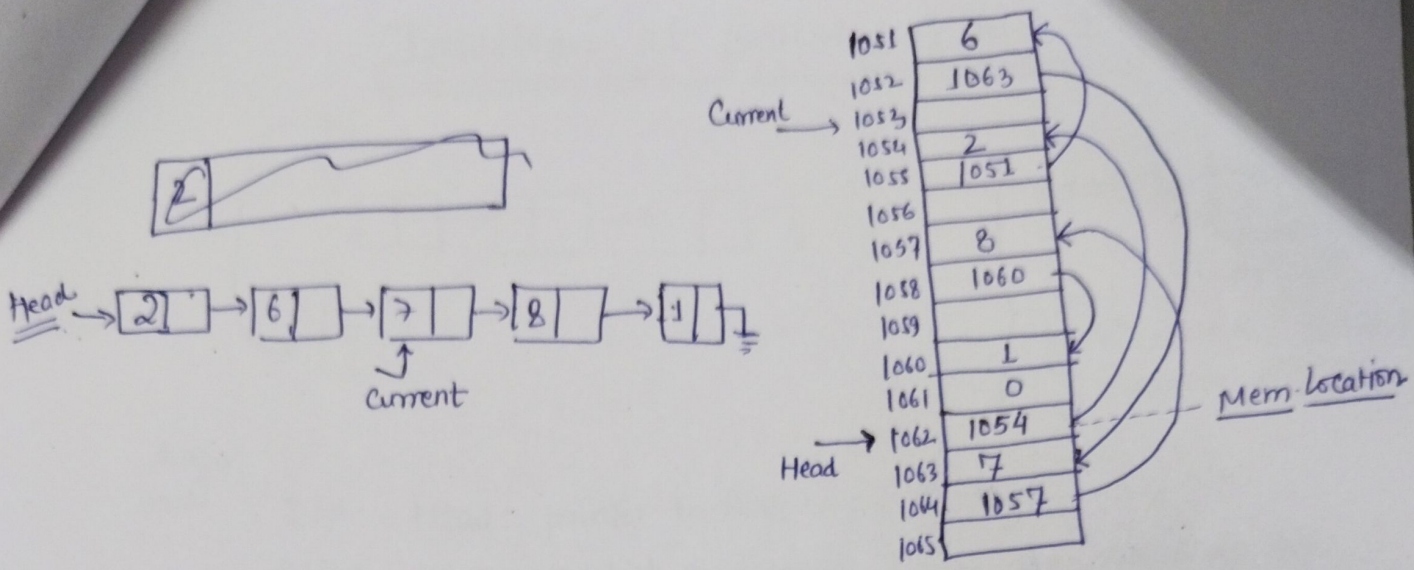
The object field will hold the actual list element
The next field will hold the starting location of the next node.
"Chain" the nodes together to form a linked list.

Ex:- list (2, 6, 7, 8, 1)



→ Feature of the list :-

- Need a "head" to point the first ~~element~~ node of the list. Otherwise, we wouldn't know where is the start of the list is.
- The current here is a pointer not index.
- The ~~last~~ next field in the last node points to nothing. We will place the memory address NULL which is gaurantee to be inaccessible.



Operations:-

- Traversal :- searching
- Insertion
- Deletion

Searching an Element In the list

Algo: I/P : Head, Key
O/P : Appropriate message
- → Element found
- → element not found
- → list is empty

```
if (head = Null)
    Print " list is empty"
```

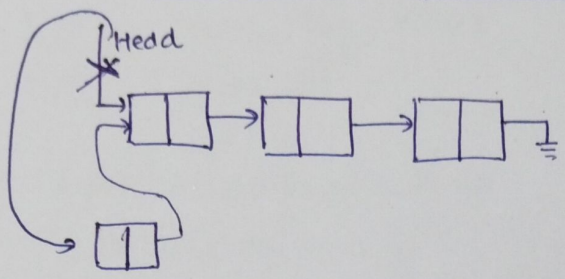
Initialize a node pointer temp with head
While (temp != Null && temp[data] != Key)

temp ← temp [next] ▷ temp takes new pointer

Outside while:
```
if ( temp = NULL)
    ele. not found
else
    ele. found.
```

O(n)

Insertion at Begining of the list



add (9): create a new node in memory to hold 9.

Node* newNode = new Node(9)

newNode → [9 | ·]

Algo:

I/P: Head: pointer to first node & key

O/P: Augmented list containing a new node which has key

1: Create ~~new~~ a node pointer temp

2: temp [data] ← Key

3: temp [next] ← Head

4: Head ← temp

O(1)

INsertion at the end

Algo:

I/P: Head, Key

O/P: New list

1: Create node pointer "temp"

2: temp [data] ← Key

3: temp [next] ← NULL

4: If (Head = NULL)

Head ← temp

else

temp1 ← Head ▷ Initialize new pointer temp1

While (temp1 [next] != NULL)

temp1 ← temp [next]

temp1 [next] ← temp

O(n)

- Insertion After Specific Value -

I/P: Head, Key, value
O/P: New list

1- create a new pointer temp
2. temp [data] ← Key
3. temp [next] ← NULL
4. temp1 with head (initialize temp1 with head: give the address of Head to temp1)
5. While (temp1 != NULL && temp1 [data] != value)

   temp1 ← temp [next]

6. If (temp1 = Null)

   node with "value" not present

   else if (temp1[next] = NULL) ▷ last node with value

   temp1 [next] = temp

   else

   temp [next] ← temp1 [next]

   temp1 [next] ← temp

O(n)

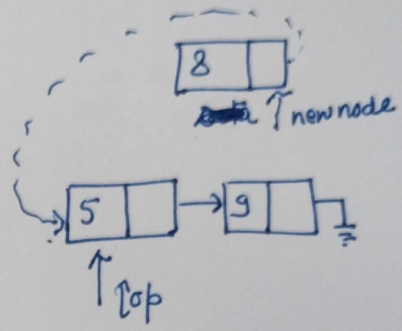
Linked List Implementation of Stack

Top: pointer that points the top most element of the stack.

Data: data item to be pushed

Procedure: (Push Operation)

* Create a new node
* put the ~~dat~~ add^s of Head in the next part of new node
* update the top pointer.



1: Create a node pointer (newNode)
2: newNode[data] ← Data
3: newNode[next] ← Top
4: Top ← newNode

Pop Procedure:-

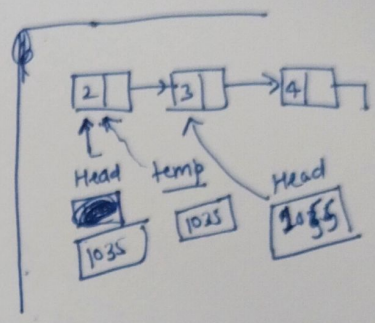
- Create a temp pointer
- update temp for pointing first node
- store the value of first node somewhere
- shift the top pointer towards next node
- Delete temporary node.

1. if (Top = Null)
2. Print "underflow condition"
3. else
4. initialize temp pointer with Top
5. Top = Top[next]
6. release the memory location pointed by temp.

Deletion In the list

From beginning:-

```
1   If (head = NULL)
2       Print " list is empty"
3.  else
4.
5.      temp ← Head
        Head ← Head [next]
6.  release the memory pointed by temp
```



From End:-

```
If (Head = NULL)                          ] List is empty
|   Print " List is empty".               ]
else
|   temp ← Head
    While ( temp [next] != NULL)
    |
    |       temp ← temp [next]
    |
    if (temp = Head)                      } only one node
    |       Head ← NULL                   }
    else
        Prev [next] ← ~~Head~~ NULL
    Release the memory location of temp.
```