Sunbeam Institute of Information Technology
Pune and Karad

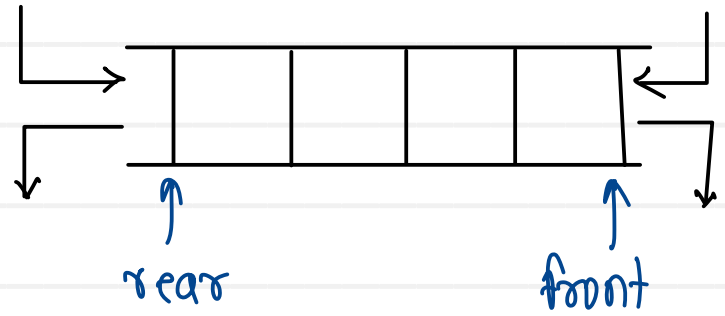
Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Deque : Double Ended Queue

- insertion & deletion is allowed from both the ends



- this queue is implemented by circular array or doubly linked list

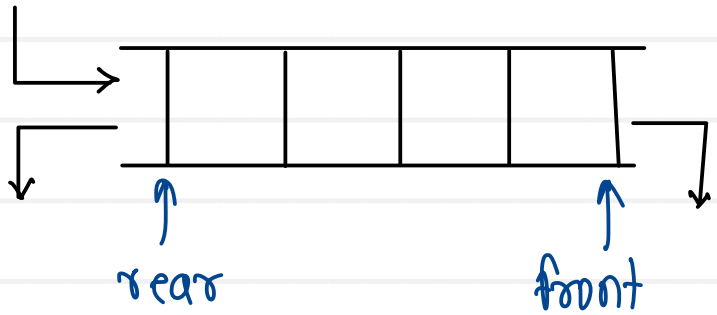
Operations :

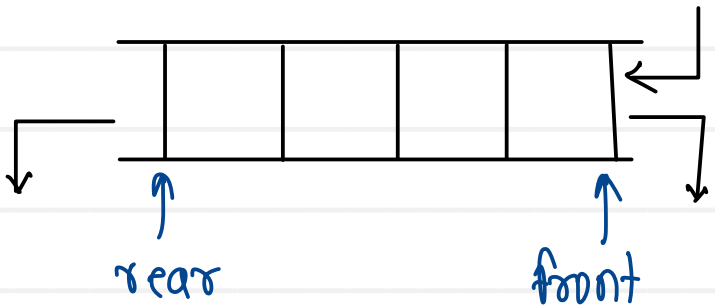
1. pushFront( )
2. pushRear( )
3. popFront( )
4. popRear( )
5. getFront( )
6. getRear( )

isFull( )
isEmpty( )

Input Restricted Deque

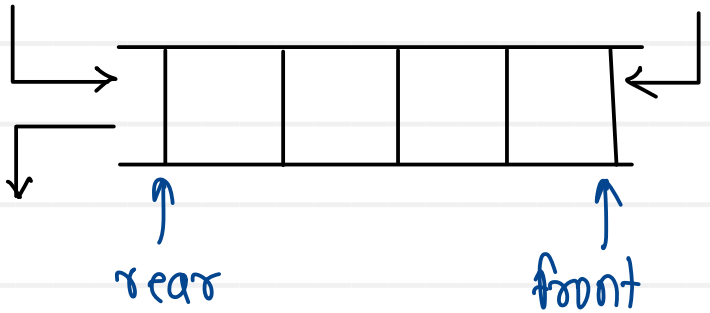
- one input end is closed





Output Restricted Deque

- one output end is closed

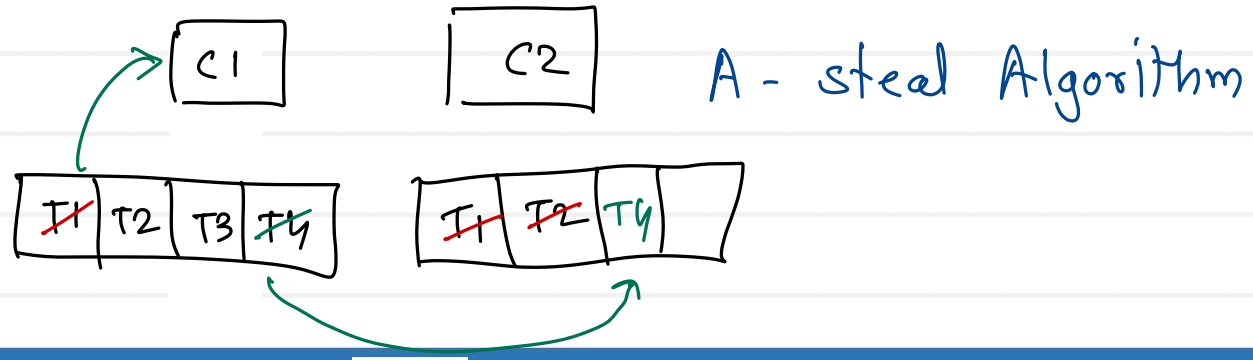


Applications of Queue

- job queue, ready queue or waiting queue in OS
- jobs submitted to printer [spooler directory]
- call to customer care
- token systems
- in networks to modify common files
- advanced data structures for traversal (BFS - queue)

Applications of Deque

- stack & queue can be efficiently implemented together using deque
- can be used to check palindrome
- undo & redo operations in softwares
- Browser history
- per CPU seperate deque is maintained to keep processes



A - steal Algorithm

Stack Applications

- Lexical analysis in compilers
- parenthesis balancing
- function activation records
  - ↳ control the flow program
- traversal in advanced data structures
  - DFS - stack
  - preorder/inorder/postorder - stack
- Expression conversion & evaluation

Expression : combination of operands & operators

Infix	:	op1 opr op2	:	a + b ← human
Prefix	:	opr op1 op2	:	+ a b } ← ALV → CPU → machine
Postfix	:	op1 op2 opr	:	a b + }

compiler → conversion
ALV → evaluation
} stack is used

Applications – Stack and Queue

Stack

- Parenthesis balancing
- Expression conversion and evaluation
- Function calls
- Used in advanced data structures for traversing

- **Expression conversion and evaluation:**
  - Infix to postfix
  - Infix to prefix
  - Postfix evaluation
  - Prefix evaluation

Queue

- Jobs submitted to printer
- In Network setups – file access of file server machine is given to First come First serve basis
- Calls are placed on a queue when all operators are busy
- Used in advanced data structures to give efficiency.
- Process waiting queues in OS

Postfix Evaluation

- Process each element of postfix expression from left to right
- If element is operand
  - Push it on a stack
- If element is operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op2 – first popped element
    - Op1 – second popped element
  - Perform current element (Operator) operation between Op1 and Op2
  - Again push back result onto the stack
- When single value will remain on stack, it is final result
- e.g. 4 5 6 * 3 / + 9 + 7 -

Postfix evaluation

Postfix expression : 4 5 6 * 3 / + 9 + 7 -

left ⟶ right

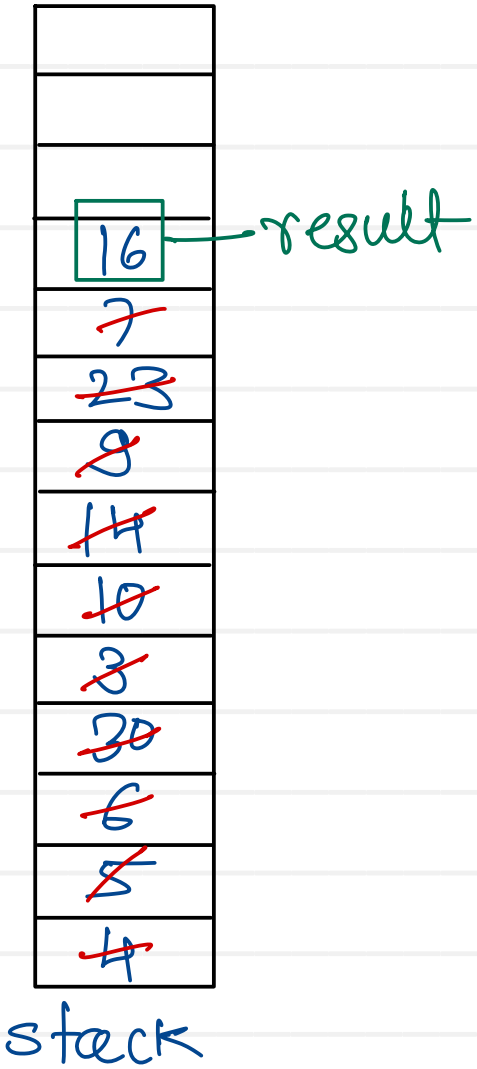
⑤ 23 − 7 = 16

④ 14 + 9 = 23

③ 4 + 10 = 14

② 30 / 3 = 10

① 5 * 6 = 30



Prefix Evaluation

- Process each element of prefix expression from right to left
- If element is operand
  - Push it on a stack
- If element is operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op1 – first popped element
    - Op2 – second popped element
  - Perform current element (Operator) operation between Op1 and Op2
  - Again push back result onto the stack
- When single value will remain on stack, it is final result
- e.g. - + + 4 / * 5 6 3 9 7

Prefix evaluation

Prefix expression : - + + 4 / * 5 6 3 9 7

left ← right

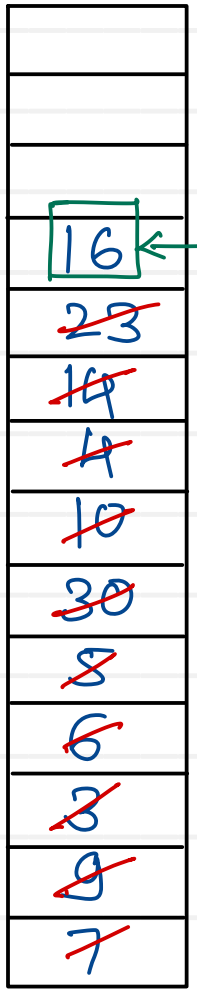
23 - 7 = 16

14 + 9 = 23

4 + 10 = 14

30 / 3 = 10

5 * 6 = 30



infix = 10 + 20

prefix = + 10 20

postfix = 10 20 +

strings arr[] = postfix.split(" ");

arr[] = { "10", "20", "+" };

Infix to Postfix Conversion

- Process each element of infix expression from left to right
- If element is Operand
  - Append it to the postfix expression
- If element is Operator
  - If priority of topmost element (Operator) of stack is greater or equal to current element (Operator), pop topmost element from stack and append it to postfix expression
  - Repeat above step if required
  - Push element on stack
- Pop all remaining elements (Operators) from stack one by one and append them into the postfix expression
- e.g. a * b / c * d + e – f * h + i

Infix to Postfix conversion

Infix expression : 1 $ 9 + 3 * 4 - (6 + 8 / 2) + 7

l ⟶ r

Postfix expression : 19$34*+682/+-7+

Infix to Prefix Conversion

- Process each element of infix expression from right to left
- If element is Operand
  - Append it to the prefix expression
- If element is Operator
  - If priority of topmost element of stack is greater than current element (Operator), pop topmost element from stack and append it to prefix expression
  - Repeat above step if required
  - Push element on stack
- Pop all remaining elements (Operators) from stack one by one and append them into the prefix expression
- Reverse prefix expression
- e.g. a * b / c * d + e – f * h + i

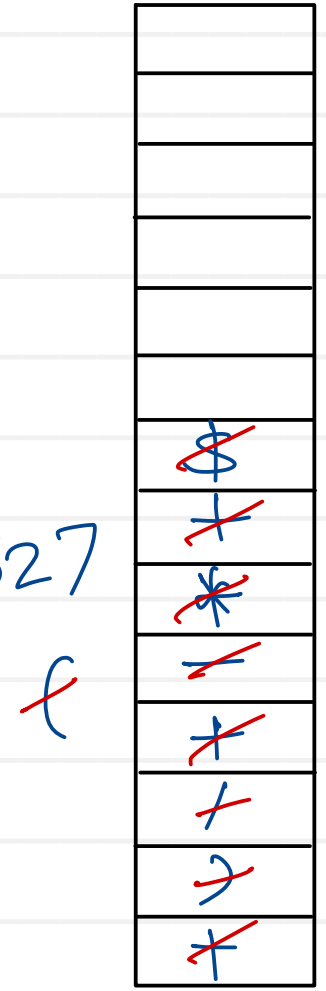
Infix to Prefix conversion

Infix expression : 1 $ 9 + 3 * 4 - (6 + 8 / 2) + 7

l ← r

Expression : 7 2 8 / 6 + 4 3 * 9 1 $ + - +

Prefix expression : + - + $ 1 9 * 3 4 + 6 / 8 2 7



Prefix to Postfix

- Process each element of prefix expression from right to left
- If element is an Operand
  - Push it on to the stack
- If element is an Operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op1 – first popped element
    - Op2 – second popped element
  - Form a string by concatenating Op1, Op2 and Opr (element)
  - String = “Op1+Op2+Opr”, push back on to the stack
- Repeat above two steps until end of prefix expression.
- Last remaining on the stack is postfix expression
- e.g. * + a b – c d

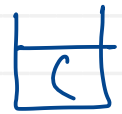
Postfix to Infix

- Process each element of postfix expression from left to right
- If element is an Operand
  - Push it on to the stack
- If element is an Operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op2 – first popped element
    - Op1 – second popped element
  - Form a string by concatenating Op1, Opr (element) and Op2
  - String = “Op1+Opr+Op2”, push back on to the stack
- Repeat above two steps until end of postfix expression.
- Last remaining on the stack is infix expression
- E.g. a b c - + d e – f g – h + / *

Valid Parantheses

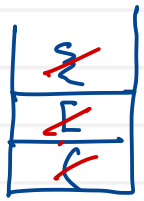
Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
An input string is valid if:
- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.
- Every close bracket has a corresponding open bracket of the same type.

Example 1:
Input: s = "()"
Output: true



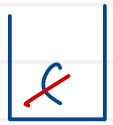
) == (

Example 2:
Input: s = "()[]{}"
Output: true



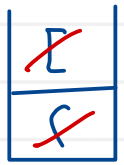
) == (
] == [
} == {

Example 3:
Input: s = "(]"
Output: false



] != (

Example 4:
Input: s = "([])"
Output: true



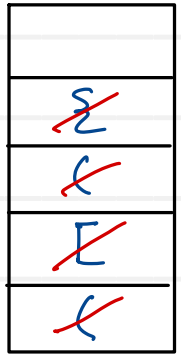
] == [
) == (

1. create stack for parenthesis
2. traverse string from left to right
   2.1 if opening parenthesis
       push it on stack
   2.2 if closing parenthesis
       if stack is not empty,
       pop opening from stack & compare with closing, if they are matching continue, otherwise return false.
3. if stack is not empty, return false
4. if stack is empty, return true

Parenthesis balancing using stack

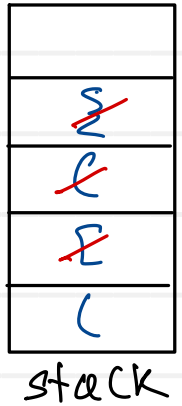
5+([9-4]*(8-{6/2}))

] == [
} == {
) == (
) == (



5+([9-4]*(8-{6/2]))

] == [
} == {
] != (



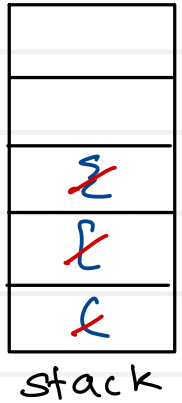
opening ([{
0 1 2

closing)] }
0 1 2

string
↓
indexOf( )
↓
returns index of char
returns -1 if char not found

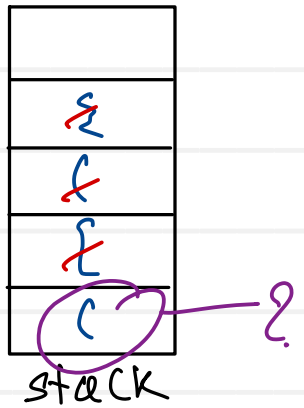
5+([9-4]*8-{6/2}))

] == [
} == {
) == (
) == {



5+([9-4]*(8-{6/2})

] == [
} == {
) == (



Algorithm

Program : set of rules / instructions to processor / CPU
Algorithm : set of instructions to human (programmer)
- step by step solution of given problem

- Algorithms are programming language independent.
- Algorithms can be written in any human understandable language.
- Algorithms can be used as a templates

Algorithm ⟶ Program / function
(Template) (Implementation)

Find sum of array elements
step 1 : create sum & initialize to 0
step 2 : traverse array for 0 to N-1 index
step 3 : Add every element of array in sum
step 4 : return / print sum

e.g. searching, sorting
↓ ↓
linear / binary selection / bubble / quick

Algorithm analysis

- it is done for efficiency measurement and also known as time/space complexity
- It is done to finding time and space requirements of the algorithm
  1. Time - time required to execute the algorithm (μs, ns, ms, s)
  2. Space - space required to execute the algorithm inside memory (byte, kb, mb...)

- finding exact time and space of the algorithm is not possible because it depends on few external factors like
  - time is dependent on type of machine (CPU), number of processes running at that time
  - space is dependent on type of machine (architecture), data types

- Approximate time and space analysis of the algorithm is always done

- Mathematical approach is used to find time and space requirements of the algorithm and it is known as "Asymptotic analysis"

- Asymptotic analysis also tells about behaviour of the algorithm for different input or for change in sequence of input

- This behaviour of the algorithm is observed in three different cases
  1. Best case
  2. Average case
  3. Worst case

Big O (O) (upper bound) Omega (Ω) (lower bound) Theta (Θ) (Avg / tight bound)

Time complexity

- time is directly proportional to number of iterations of the loops used in an algorithm
- To find time complexity/requirement of the algorithm count number of iterations of the loops

1. Print 1D array on console

```
void print1DArray(int arr[], int n) {
    for(i=0; i<n; i++)
        sysout(arr[i]);
}
```

No. of iterations = n

Time ∝ No. of iteration

Time ∝ n

$T(n) = O(n)$

2. Print 2D array on console

```
void print2DArray(int arr[][], int m, int n)
{
    for(i=0; i<m; i++) {
        for(j=0; j<n; j++)
            sysout(arr[i][j]);
    }
}
```

iterations of outer loop = m

iterations of inner loop = n

Total iterations = m * n

Time ∝ m * n

$T(m, n) = O(m * n)$

$m \approx n$

itr = n * n

Time ∝ n^2

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

Time complexity

3. Add two numbers

```
int sum ( int n1 , int n2)
    return n1 + n2;
}
```

- irrespective of input time required will be same all the time.
- constant time requirement

$$T(n) = O(1)$$

4. Print table of given number

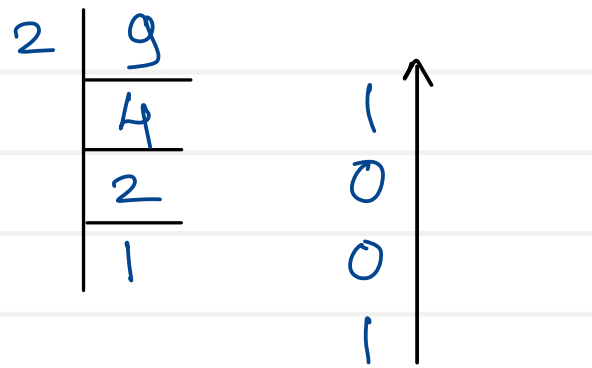
```
void printTable(int num) {
    for (i=1 ; i<=10 ; i++)
        sysout ( i * num);
}
```

- irrespective of number loop will iterate fix number of times
- constant time requirement

$$T(n) = O(1)$$

Time complexity

5. Print binary of decimal number



$(9)_{10} = (1001)_2$

```
void printBinary(int n) {
    while(n > 0) {
        rem = n % 2;
        n = n/2;
    }
}
```

n	n>0	n%2
9	T	1
4	T	0
2	T	0
1	T	1
0	F	

$n = n, n/2, n/4, n/8 \ldots$

$= n/2^0, n/2^1, n/2^2, n/2^3 \ldots\ldots n/2^{itr}$

$n/2^{itr} = 1$

$2^{itr} = n$

$\log 2^{itr} = \log n$

$itr \log 2 = \log n$

$itr = \frac{\log n}{\log 2}$

$Time \propto itr$

$Time \propto \frac{1}{\log 2} \log n$

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

Time complexity

Time complexities : O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3), …., O(2^n), ……

Modification : + or - : time complexity is in terms of n
Modification : * or / : time complexity is in terms of log n

```
for(i=0; i<n; i++) → O(n)
for(i=n; i>0; i--) → O(n)
for(i=0; i<n; i+=20) → O(n)
for(i=n; i>0; i/=2) → O(log n)
for(i=1; i<n; i*=2) → O(log n)
```

n = 9, 4, 2, 1
i = 1, 2, 4, 8

```
for(i=1; i<=10; i++) → O(1)
```

```
for(i=0; i<n; i++) → n
  for(j=0; j<n; j++) → n
                          * → O(n^2)
```

```
for(i=0; i<n; i++); → n
for(j=0; j<n; j++); → n
                        + = 2n → O(n)
```

```
for(i=0; i<n; i++) → n
  for(j=n; j>0; j/=2) → log n
                          * → O(n log n)
```

Time complexity

```
for(i=n/2; i<=n; i++)          -> n
  for(j=1; j+n/2<=n; j++)      -> n
    for(k=2; k<=n; k=k*2)      -> log n
```

Total itr = $n * n * \log n$

$= n^2 \log n$

```
for(i=n/2; i<=n; i++)          -> n
  for(j=1; j<=n; j=2*j)        -> log n
    for(k=1; k<=n, k=k*2)      -> log n
```

total itr = $n * \log n * \log n$

$= n \log^2 n$

Space complexity

- Finding approximate space requirement of the algorithm to execute inside memory

Total space = Input space + Auxiliary space

space of actual input

space of extra variables which are needed to process the input.

```
int findSum(int arr[], int n){
    int sum = 0;
    for(i=0; i<n; i++)
        sum += arr[i];
    return sum;
}
```

input variable = arr
processing variables = n, i, sum
Auxillary space = 3 units

space is constant here

$S(n) = O(1)$

	Stack	Queue
push	O(1)	O(1)
pop	O(1)	O(1)
peek	O(1)	O(1)

	SLL	
	head	head & tail
Add first	O(1)	O(1)
last	O(n)	O(1)
pos	O(n)	O(n)
Delete first	O(1)	O(1)
last	O(n)	O(n)
pos	O(n)	O(n)
Traverse	O(n)	O(n)

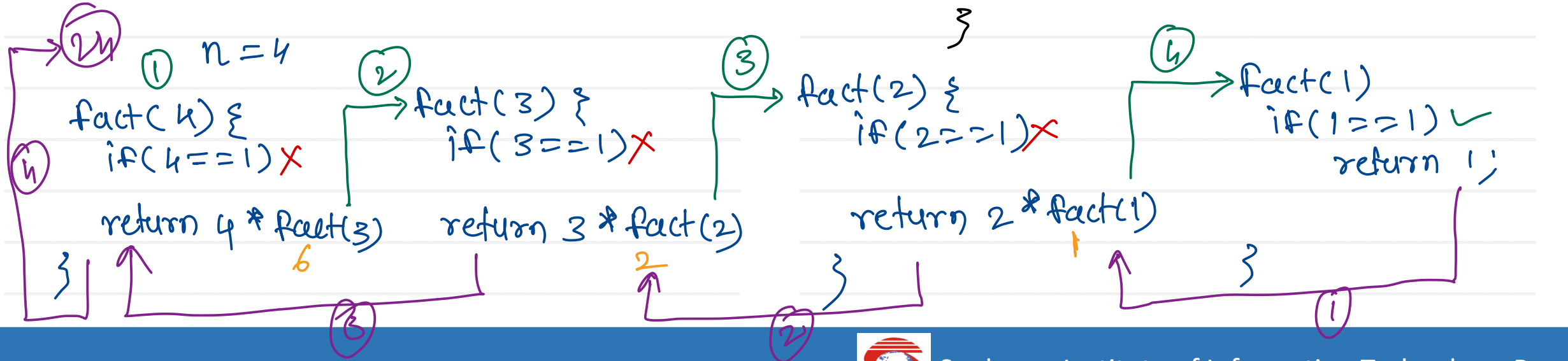
Recursion

- Calling function within itself
- we can use recursion
  - if we know formula/process in terms of itself
  - if we know terminating condition

e.g. $n! = n * (n-1)!$

$0! = 1! = 1$

```
int fact(int n) {
    if(n==1)
        return 1;
    return n * fact(n-1);
}
```



Algorithm analysis

Iterative

- loops are used

```
int fact(int num) {
    int f=1;
    for(int i=1; i<=num; i++)
        f *= i;
    return f;
}
```

n

Time ∝ no. of iterations of the loop

Time ∝ n

$T(n) = O(n)$

$S(n) = O(1)$

Recursive

- recursion is used

```
int rfact(int num) {
  if(num==1)
     return 1;
  return num * rfact(num-1);
}
```

Time ∝ No. of recursive calls

Time ∝ n

$T(n) = O(n)$

$S(n) = O(n)$

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com