1. Binary search (Algorithm, Example with steps)

   Binary Search is an efficient algorithm for finding an element in a **sorted array** by repeatedly dividing the search interval in half.

   It works in **O(log n)** time complexity.

   Algorithm:

   1. Initialize low = 0 and high = n - 1
   2. While low ≤ high:
      a. mid = (low + high) // 2
      b. If arr[mid] == x → return mid
      c. If arr[mid] < x → low = mid + 1
      d. Else → high = mid - 1
   3. Return -1 (element not found)

   Example with Steps

   Given:

- arr = [10, 20, 30, 40, 50, 60, 70]
- x = 50

Step-by-step Execution:

- **Step 1:** low = 0, high = 6
  mid = (0 + 6) // 2 = 3 → arr[3] = 40
  → 50 > 40 → search in right half → low = 4
- **Step 2:** low = 4, high = 6
  mid = (4 + 6) // 2 = 5 → arr[5] = 60
  → 50 < 60 → search in left half → high = 4
- **Step 3:** low = 4, high = 4
  mid = (4 + 4) // 2 = 4 → arr[4] = 50
  → Found the target! Return index 4

2. Shell sort (Explanation, Example with steps)

Shell Sort is an in-place comparison-based sorting algorithm. It is a generalization of **insertion sort** that allows the exchange of items that are far apart. It improves on insertion sort by **comparing elements that are distant** (using a gap), then gradually reducing the gap

Exchange based sorting algorithm

It allows exchange of elements that are gap apart.

Explanation:

- Start with a large **gap** (typically n/2) and reduce the gap to 1.
- For each gap, do a **gapped insertion sort**.

- This allows elements to move faster toward their correct positions.

Gap=3*gap+1

Gap1,k=1

Gap2=>k=k*3+1

3*1+1

3+1

4

Gap3=>k=k*3+1

4*3+1

12+1

13

Example with Steps:

Unsorted Array: [23, 12, 1, 8, 34, 54, 2, 3]

Step 1: gap = 4

Compare and sort elements that are 4 apart:

- Compare arr[0] and arr[4]: 23 vs 34 → OK
- Compare arr[1] and arr[5]: 12 vs 54 → OK
- Compare arr[2] and arr[6]: 1 vs 2 → OK
- Compare arr[3] and arr[7]: 8 vs 3 → **Swap** → [23, 12, 1, 3, 34, 54, 2, 8]

Step 2: gap = 2

- Compare arr[0] and arr[2]: 23 vs 1 → Swap
  → [1, 12, 23, 3, 34, 54, 2, 8]
- Compare arr[1] and arr[3]: 12 vs 3 → Swap
  → [1, 3, 23, 12, 34, 54, 2, 8]
- Compare arr[2] and arr[4]: 23 vs 34 → OK
- Compare arr[3] and arr[5]: 12 vs 54 → OK
- Compare arr[4] and arr[6]: 34 vs 2 → Swap
  → [1, 3, 23, 12, 2, 54, 34, 8]

- Compare arr[5] and arr[7]: 54 vs 8 → Swap
  → [1, 3, 23, 12, 2, 8, 34, 54]

Step 3: gap = 1 (Normal Insertion Sort)

Final sort using normal insertion sort:

→ [1, 2, 3, 8, 12, 23, 34, 54]

3. Indexed sequential search (Explanation with, diagram)

   indexed Sequential Search is a hybrid search algorithm that combines the benefits of **sequential search** and **binary search**. It's mainly used for **large datasets stored on disk**, where direct access is expensive.

   Explanation:

   It works in two steps:

1. **Index Table Search (Fast Access):**
   - An **index table** holds key entries and pointers (or positions) to the actual records.
   - You first search **this index** (usually using **binary search**) to locate the block where the record might be.
2. **Sequential Search in Block:**
   - Once the correct block is identified, a **sequential search** is used within that block to find the exact record.

INDEX TABLE:

Key	Address
10	0
30	2
50	4
70	6

ACTUAL DATA (Sorted):

Index → Value

0 → 10

1 → 20

2 → 30

3 → 40

4 → 50

5 → 60

6 → 70

7 → 80

Example: Search for 40

1. **Search index table:**
   - Keys: [10, 30, 50, 70]
   - 40 lies between 30 and 50 → Go to **address 2**
2. **Sequential search in block starting from address 2 (data[2] to data[3]):**
   - Check data[2] → 30 → not found
   - Check data[3] → 40 → ✅ found!

3. Infix to postfix (Mathematical/using algorithm with stack)

Infix expression: Operators are written between operands (e.g., A + B)
Postfix expression (Reverse Polish Notation): Operators follow their operands (e.g., AB+)

Algorithm: Infix to Postfix using Stack

1. Initialize an empty **stack** and an empty **output string**.
2. Scan the infix expression **left to right**:
   - **Operand:** Add to output.
   - **Left Parenthesis (:** Push to stack.
   - **Right Parenthesis)**: Pop and append until (is found. Discard (.
   - **Operator (+, -, *, /, ^)**:

- While the top of the stack has **higher or equal precedence**, **pop** it to output.
- Push the current operator to the stack.

3. After scanning, **pop remaining operators** from the stack to the output.

Example:

Convert:
A + B * (C - D)

Step-by-step:

Symbol	**Stack**	**Output**
A		A
+	+	A
B	+	AB
*	+ *	AB
(	+ * (	AB
C	+ * (	ABC
-	+ * (-	ABC
D	+ * (-	ABCD
)	+ *	ABCD-
(end)		ABCD-*+

✅ Final Postfix: ABCD-*+

4. Evaluation of postfix expression

Postfix (also called **Reverse Polish Notation**) eliminates the need for parentheses and follows a simple evaluation using a **stack**.

How It Works:

1. **Scan the postfix expression** from left to right.
2. **Operands:** Push them onto the stack.
3. **Operators:** Pop the top two operands from the stack, apply the operator, and **push the result** back onto the stack.
4. At the end, the **stack contains the final result**.

Example:

Postfix Expression:
5 6 2 + * 12 4 / -

✅ This corresponds to:
5 * (6 + 2) - (12 / 4)

Step-by-step Evaluation:

Token	Stack	Action
5	5	Push operand
6	5, 6	Push operand
2	5, 6, 2	Push operand
+	5, 8	6 + 2 = 8
*	40	5 * 8 = 40
12	40, 12	Push operand
4	40, 12, 4	Push operand
/	40, 3	12 / 4 = 3
-	37	40 - 3 = 37

Final Result: **37**

5. Operations of stack (algorithm and code)

Basic Stack Operations

Operation	Description
push(x)	Add element x to the top of the stack
pop()	Remove and return the top element
peek()	Return the top element without removing it
isEmpty()	Check if the stack is empty
isFull()	(In fixed-size stack) Check if it's full

PUSH(x):

```
if top == MAX_SIZE - 1:
    print "Stack Overflow"
else:
    top = top + 1
    stack[top] = x

POP():

if top == -1:
    print "Stack Underflow"
else:
    x = stack[top]
    top = top - 1
    return x

PEEK():

if top == -1:
    print "Stack is empty"
else:
    return stack[top]

isEmpty():
return (top == -1)
```

C Code Implementation

```
#include <stdio.h>
#define MAX 100

int stack[MAX];
int top = -1;

void push(int x) {
    if (top == MAX - 1) {
        printf("Stack Overflow\n");
    } else {
        top++;
        stack[top] = x;
        printf("%d pushed to stack\n", x);
    }
}

int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        return -1;  // Indicate error
    } else {
        int x = stack[top];
        top--;
        return x;
```

```
    }
}

int peek() {
    if (top == -1) {
        printf("Stack is empty\n");
        return -1;  // Indicate error
    } else {
        return stack[top];
    }
}

int isEmpty() {
    return (top == -1);
}

void display() {
    if (top == -1) {
        printf("Stack is empty\n");
        return;
    }
    printf("Stack elements: ");
    for (int i = 0; i <= top; i++) {
        printf("%d ", stack[i]);
    }
```

```
    printf("\n");
}

int main() {
    push(10);
    push(20);
    push(30);
    display();  // Stack elements: 10 20 30

    printf("Top element is %d\n", peek());

    printf("Popped element: %d\n", pop());
    display();  // Stack elements: 10 20

    if (isEmpty()) {
        printf("Stack is empty\n");
    } else {
        printf("Stack is not empty\n");
    }

    return 0;
}
```