Vel Tech
Rangarajan Dr. Sagunthala
R&D Institute of Science and Technology
(Deemed to be University Estd. u/s 3 of UGC Act, 1956)

School of Computing

Department of Computer Science and Engineering

ACADEMIC YEAR 2025 – 2026

Summer Semester

20231CS301 – Advanced Data Structures and Algorithms Laboratory

NAME:

VTP NO:

REG.NO:

DEGREE/BRANCH:

YEAR/SEM:

SLOT NO:

School of Computing

Department of Computer Science and Engineering

ACADEMIC YEAR 2025 – 2026

Summer Semester

BONAFIDE CERTIFICATE

NAME:

VTP NO:

REG.NO:

DEGREE/BRANCH:

YEAR/SEM:

SLOT NO:

Certified that this is a Bonafide record of work done by above student in the "20231CS301 – Advanced Data Structures and Algorithms Laboratory" during the year 2025-2026.

SIGNATURE OF FACULTY INCHARGE **SIGNATURE OF HOD**

Submitted for the Semester Examination held on _____ at Vel Tech Rangarajan Dr. Sagunthala R & D Institute of Science and Technology.

INTERNAL EXAMINER **EXTERNAL EXAMINER**

Vel Tech
Rangarajan Dr. Sagunthala
R&D Institute of Science and Technology
(Deemed to be University Estd. u/s 3 of UGC Act, 1956)

TABLE OF CONTENTS

4		**Algorithmic Techniques**			
	28-08-2025	i) Implement the Strassen's algorithm for matrix multiplication using the divide and conquer approach.	61		
	04-09-2025	ii) Implement merge sort to count the number of inversions in an array using divide and conquer strategy.	68		
	04-09-2025	iii) Implement the Huffman coding algorithm for text compression using Greedy technique.	74		
	11-09-2025	iv) Implement longest common subsequence among multiple strings using dynamic programming.	80		
	11-09-2025	v) Solve the coin change problem with a twist: instead of finding the number of ways to make change, find the minimum number of coins needed to make a certain amount.	83		
5		**Elementary Graph Algorithms**			
	18-09-2025	i) Implement Kruskal's algorithm to find minimum spanning tree of a weighted graph.	87		
	18-09-2025	ii) Implement Prim's algorithm to find minimum spanning tree using priority queue.	91		
	25-09-2025	iii) Implement the Bellman-Ford algorithm to find the shortest path from a source vertex to all other vertices, even in graphs with negative-weight edges.	102		
6		**Network Flow Algorithms**			
	09-10-2025	i) Implement the Ford-Fulkerson algorithm to find the maximum flow in a network. Test it on different flow networks and explore different augmenting path strategies (BFS & DFS).	109		
	16-10-2025	ii) Implement the Edmonds-Karp algorithm, a specific implementation of Ford Fulkerson using BFS for finding augmenting paths.	116		

	23-10-2025	iii) Implement max-flow min-cut algorithm to image segmentation problems: to partition an image into different segments while minimizing the cut.	121		
7		**Number Theoretic Algorithms**			
	30-10-2025	i) Implement an algorithm to generate strong pseudoprimes based on the Miller-Rabin primality test.	127		
	30-10-2025	ii) Implement Pollard's Rho algorithm for integer factorization	133		
8		**Probabilistic & Geometric Algorithms**			
	06-11-2025	i) Implement an algorithm generates random passwords using pseudorandom numbers. Users can specify the desired password length and character set	137		
	06-11-2025	ii) Implement a basic plane sweep algorithm. Use it to solve a simple geometric problem, such as finding intersections among a set of line segments.	141		
9		**Randomization and Linear Programming**			
	13-11-2025	i) Implement a randomized algorithm to select a random subset of elements from a given array with equal probability	145		
	13-11-2025	ii) Implement a linear programming model to optimize the construction of a binary search tree with specific access frequencies for the elements. Solve it to find the optimal tree structure.	149		

Task No.	1.1	**Binomial Heap: Implement the decrease-key operation in a binomial heap and analyze its time complexity.**
Date:	24-07-2025	

Aim:

To implement and perform basic operations on a binomial heap, including insertion, decrease-key, and finding the minimum value in C programming.

Procedure:

1. **Define the Structure**:
    - Define a BinomialTreeNode structure with fields for key, parent, child, and sibling pointers.
    - Define a BinomialHeap structure with a pointer to the root node.
2. **Create Heap**:
    - Implement createBinomialHeap to initialize and allocate memory for a new empty binomial heap.
3. **Insert Nodes**:
    - Implement insertBinomialHeap to insert nodes into the heap by creating a new node and linking it as a sibling of the root.
4. **Decrease Key**:
    - Implement decreaseKeyBinomialHeap to update the key of a specified node and rearrange nodes based on the new key value to maintain the min-heap property.
5. **Find Minimum Node**:
    - Implement findMinBinomialHeap to find the node with the minimum key in the binomial heap.
6. **Print Heap**:
    - Implement printBinomialHeap to traverse and display all keys in the heap for visualization.
7. **Main Function**:
    - In the main function, create a binomial heap, insert nodes, perform a decrease-key operation, and print the heap before and after the decrease-key operation.

Algorithm:

1. Allocate memory for a new BinomialHeap.
2. Initialize the root of the heap to NULL.
3. Return the newly created heap.
4. Allocate memory for a new BinomialTreeNode.
5. Set the key of the node to the given key value.
6. Initialize the parent, child, and sibling pointers of the node to NULL.
7. If the heap's root is NULL, set the root of the heap to this new node.
8. If the heap already has nodes:
    - Start from the root.
    - Traverse through the sibling nodes of the root until reaching the last sibling.
    - Link the new node as the sibling of the last node.
9. End the insertion.
10. Check if the newKey is greater than the node's current key.
    - If true, exit the function.
11. Set the key of the node to newKey.
12. While the node's key is less than its parent's key and it has a parent:
    - Swap the key of the node with the key of its parent to maintain the min-heap property.
    - Update the node pointer to point to its parent for the next iteration.
13. End decrease-key operation.
14. Initialize a pointer current to the heap's root.
15. Initialize a pointer minNode to point to current.
16. While current is not NULL:
    - If current node's key is less than minNode's key, update minNode to current.
    - Move current to its sibling.
17. Return the node pointed to by minNode.
18. Initialize a pointer current to the heap's root.
19. While current is not NULL:

- Print the key of the current node.
- Move current to its sibling.

20. Print a newline character to end the output.
21. End print operation.
22. Create an empty binomial heap.
23. Insert nodes with keys 10, 5, 20, and 15 into the binomial heap.
24. Print the heap keys before the decrease-key operation.
25. Find the minimum node in the heap.
26. Perform a decrease-key operation on the minimum node to update its key to 3.
27. Print the heap keys after the decrease-key operation.
28. End.

Program:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct BinomialTreeNode {
int key;
struct BinomialTreeNode *parent;
struct BinomialTreeNode *child;
struct BinomialTreeNode *sibling;
} BinomialTreeNode;

typedef struct BinomialHeap {
BinomialTreeNode *root;
} BinomialHeap;

BinomialHeap *createBinomialHeap() {
BinomialHeap *heap = malloc(sizeof(BinomialHeap));
heap->root = NULL;
return heap;
}

void insertBinomialHeap(BinomialHeap *heap, int key) {
BinomialTreeNode *node = malloc(sizeof(BinomialTreeNode));
node->key = key;
node->parent = NULL;
node->child = NULL;
node->sibling = NULL;
```

```
if (heap->root == NULL) {
heap->root = node;
} else {
BinomialTreeNode *current = heap->root;
while (current->sibling != NULL) {
current = current->sibling;
}
current->sibling = node;
}
}

void decreaseKeyBinomialHeap(BinomialHeap *heap, BinomialTreeNode *node, int newKey) {
if (node->key <= newKey) {
return;
}

node->key = newKey;

while (node->parent != NULL && node->key < node->parent->key) {
BinomialTreeNode *temp = node->parent;
node->parent = node->parent->parent;
node->parent->child = node;
temp->parent = node;
node->child = temp;
node = temp;
}
}
```

```
BinomialTreeNode *findMinBinomialHeap(BinomialHeap *heap) {
BinomialTreeNode *current = heap->root;
BinomialTreeNode *minNode = current;

while (current != NULL) {
if (current->key < minNode->key) {
minNode = current;
}
current = current->sibling;
}

return minNode;
}

void printBinomialHeap(BinomialHeap *heap) {
BinomialTreeNode *current = heap->root;
while (current != NULL) {
printf("%d ", current->key);
current = current->sibling;
}

printf("\n");
}

int main() {
BinomialHeap *heap = createBinomialHeap();

insertBinomialHeap(heap, 10);
```

```
insertBinomialHeap(heap, 5);
insertBinomialHeap(heap, 20);
insertBinomialHeap(heap, 15);

printf("Binomial heap before decrease-key operation: ");
printBinomialHeap(heap);

decreaseKeyBinomialHeap(heap, findMinBinomialHeap(heap), 3);

printf("Binomial heap after decrease-key operation: ");
printBinomialHeap(heap);

return 0;
}
```

Sample Input:

insertBinomialHeap(heap, 10);

insertBinomialHeap(heap, 5);

insertBinomialHeap(heap, 20);

insertBinomialHeap(heap, 15);

decreaseKeyBinomialHeap(heap, findMinBinomialHeap(heap), 3);

Sample Output:

Binomial heap before decrease-key operation: 10 5 20 15

Binomial heap after decrease-key operation: 3 10 20 15

Result:

The binomial heap was successfully implemented with operations for inserting nodes, decreasing the key of a node, and finding the minimum value. The decrease-key operation properly updated the node's position, maintaining the min-heap property.

Task No:	1.2	**Fibonacci Heap: Implement a mechanism for lazy deletion in Fibonacci heaps**
Date:	31-07-2025	

Aim:

To implement a Fibonacci Heap and perform various operations, including insertion, extraction of minimum node, finding minimum, union, decrease key, and deletion of nodes, using C programming.

Procedure:

1. Define the NODE structure representing nodes in the Fibonacci Heap and the FIB_HEAP structure representing the heap.
2. Implement the following functions:
    - make_fib_heap(): Creates an empty Fibonacci Heap.
    - insertion(): Inserts a new node with a given value into the heap.
    - find_min_node(): Finds the minimum node in the heap.
    - unionHeap(): Unites two Fibonacci Heaps.
    - extract_min(): Removes and returns the minimum node.
    - decrease_key(): Decreases the key value of a specified node.
    - Delete_Node(): Deletes a node by decreasing its key and removing the minimum.
    - new_print_heap(): Prints the structure of the heap.
3. Provide an interactive menu in the main() function to select and perform operations on the heap.

Algorithm:

1. Create a Fibonacci heap structure and initialize its properties (number of nodes, minimum node, and degree).

2. Allocate memory for a new node and initialize its properties (key, degree, mark, parent, child, left sibling, right sibling).

3. If the heap is empty, set the new node as the minimum node; otherwise, insert the new node into the root list and update the minimum node if necessary.

4. Increment the number of nodes in the heap.

5. To find the minimum node, check if the heap is empty; if not, return the minimum node.

6. To merge two Fibonacci heaps, link their root lists, update the minimum node, and set the new heap’s node count to the sum of both heaps’ node counts.

7. To extract the minimum node, remove it from the root list, add its children to the root list, and call the `consolidate` function to maintain the heap structure.

8. In the `consolidate` function, create an array to track the roots of each degree. Link nodes with the same degree to form a single tree and update the minimum node accordingly.

9. To link a child node to a parent, update the child pointer of the parent and increment its degree.

10. To decrease a node’s key, check if the new key is less than the current key; if so, update the key, cut the node from its parent if necessary, and perform cascading cuts up the tree.

11. To cut a node, remove it from its parent's child list, add it to the root list, and mark it as unmarked.

12. Implement a function to traverse and print the heap, including all nodes and their relationships.

13. Create a user interface for selecting operations, executing the corresponding functions based on user input.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>

typedef struct _NODE
{
int key;
int degree;
struct _NODE *left_sibling;
struct _NODE *right_sibling;
struct _NODE *parent;
struct _NODE *child;
bool mark;
bool visited;
} NODE;

typedef struct fibanocci_heap
{
int n;
NODE *min;
int phi;
int degree;
} FIB_HEAP;

FIB_HEAP *make_fib_heap();
void insertion(FIB_HEAP *H, NODE *new, int val);
```

```
NODE *extract_min(FIB_HEAP *H);
void consolidate(FIB_HEAP *H);
void fib_heap_link(FIB_HEAP *H, NODE *y, NODE *x);
NODE *find_min_node(FIB_HEAP *H);
void decrease_key(FIB_HEAP *H, NODE *node, int key);
void cut(FIB_HEAP *H, NODE *node_to_be_decrease, NODE *parent_node);
void cascading_cut(FIB_HEAP *H, NODE *parent_node);
void Delete_Node(FIB_HEAP *H, int dec_key);

FIB_HEAP *make_fib_heap()
{
FIB_HEAP *H;
H = (FIB_HEAP *)malloc(sizeof(FIB_HEAP));
H->n = 0;
H->min = NULL;
H->phi = 0;
H->degree = 0;
return H;
}
void new_print_heap(NODE *n)
{
NODE *x;
for (x = n;; x = x->right_sibling)
{

if (x->child == NULL)
{
printf("node with no child (%d) \n", x->key);
```

```
}
else
{

printf("NODE(%d) with child (%d)\n", x->key, x->child->key);
new_print_heap(x->child);
}
if (x->right_sibling == n)
{
break;
}
}
}

void insertion(FIB_HEAP *H, NODE *new, int val)
{
new = (NODE *)malloc(sizeof(NODE));
new->key = val;
new->degree = 0;
new->mark = false;
new->parent = NULL;
new->child = NULL;
new->visited = false;
new->left_sibling = new;
new->right_sibling = new;
if (H->min == NULL)
{
H->min = new;
```

```
}
else
{
H->min->left_sibling->right_sibling = new;
new->right_sibling = H->min;
new->left_sibling = H->min->left_sibling;
H->min->left_sibling = new;
if (new->key < H->min->key)
{
H->min = new;
}
}
(H->n)++;
}

NODE *find_min_node(FIB_HEAP *H)
{
if (H == NULL)
{
printf(" \n Fibonacci heap not yet created \n");
return NULL;
}
else
return H->min;
}

FIB_HEAP *unionHeap(FIB_HEAP *H1, FIB_HEAP *H2)
{
```

```
FIB_HEAP *Hnew;
Hnew = make_fib_heap();
Hnew->min = H1->min;

NODE *temp1, *temp2;
temp1 = Hnew->min->right_sibling;
temp2 = H2->min->left_sibling;

Hnew->min->right_sibling->left_sibling = H2->min->left_sibling;
Hnew->min->right_sibling = H2->min;
H2->min->left_sibling = Hnew->min;
temp2->right_sibling = temp1;

if ((H1->min == NULL) || (H2->min != NULL && H2->min->key < H1->min->key))
Hnew->min = H2->min;
Hnew->n = H1->n + H2->n;
return Hnew;
}

int cal_degree(int n)
{
int count = 0;
while (n > 0)
{
n = n / 2;
count++;
}
return count;
```

```
}
void consolidate(FIB_HEAP *H)
{
int degree, i, d;
degree = cal_degree(H->n);
NODE *A[degree], *x, *y, *z;
for (i = 0; i <= degree; i++)
{
A[i] = NULL;
}
x = H->min;
do
{
d = x->degree;
while (A[d] != NULL)
{
y = A[d];
if (x->key > y->key)
{
NODE *exchange_help;
exchange_help = x;
x = y;
y = exchange_help;
}
if (y == H->min)
H->min = x;
fib_heap_link(H, y, x);
if (y->right_sibling == x)
```

```
H->min = x;
A[d] = NULL;
d++;
}
A[d] = x;
x = x->right_sibling;
} while (x != H->min);

H->min = NULL;
for (i = 0; i < degree; i++)
{
if (A[i] != NULL)
{
A[i]->left_sibling = A[i];
A[i]->right_sibling = A[i];
if (H->min == NULL)
{
H->min = A[i];
}
else
{
H->min->left_sibling->right_sibling = A[i];
A[i]->right_sibling = H->min;
A[i]->left_sibling = H->min->left_sibling;
H->min->left_sibling = A[i];
if (A[i]->key < H->min->key)
{
H->min = A[i];
```

```
}
}
if (H->min == NULL)
{
H->min = A[i];
}
else if (A[i]->key < H->min->key)
{
H->min = A[i];
}
}
}
}

void fib_heap_link(FIB_HEAP *H, NODE *y, NODE *x)
{
y->right_sibling->left_sibling = y->left_sibling;
y->left_sibling->right_sibling = y->right_sibling;

if (x->right_sibling == x)
H->min = x;

y->left_sibling = y;
y->right_sibling = y;
y->parent = x;

if (x->child == NULL)
{
```

```
x->child = y;
}
y->right_sibling = x->child;
y->left_sibling = x->child->left_sibling;
x->child->left_sibling->right_sibling = y;
x->child->left_sibling = y;
if ((y->key) < (x->child->key))
x->child = y;

(x->degree)++;
}
NODE *extract_min(FIB_HEAP *H)
{

if (H->min == NULL)
printf("\n The heap is empty");
else
{
NODE *temp = H->min;
NODE *pntr;
pntr = temp;
NODE *x = NULL;
if (temp->child != NULL)
{

x = temp->child;
do
{
```

```
pntr = x->right_sibling;
(H->min->left_sibling)->right_sibling = x;
x->right_sibling = H->min;
x->left_sibling = H->min->left_sibling;
H->min->left_sibling = x;
if (x->key < H->min->key)
H->min = x;
x->parent = NULL;
x = pntr;
} while (pntr != temp->child);
}

(temp->left_sibling)->right_sibling = temp->right_sibling;
(temp->right_sibling)->left_sibling = temp->left_sibling;
H->min = temp->right_sibling;

if (temp == temp->right_sibling && temp->child == NULL)
H->min = NULL;
else
{
H->min = temp->right_sibling;
consolidate(H);
}
H->n = H->n - 1;
return temp;
}
return H->min;
}
```

```
void cut(FIB_HEAP *H, NODE *node_to_be_decrease, NODE *parent_node)
{
NODE *temp_parent_check;

if (node_to_be_decrease == node_to_be_decrease->right_sibling)
parent_node->child = NULL;

node_to_be_decrease->left_sibling->right_sibling = node_to_be_decrease->right_sibling;
node_to_be_decrease->right_sibling->left_sibling = node_to_be_decrease->left_sibling;
if (node_to_be_decrease == parent_node->child)
parent_node->child = node_to_be_decrease->right_sibling;
(parent_node->degree)--;

node_to_be_decrease->left_sibling = node_to_be_decrease;
node_to_be_decrease->right_sibling = node_to_be_decrease;
H->min->left_sibling->right_sibling = node_to_be_decrease;
node_to_be_decrease->right_sibling = H->min;
node_to_be_decrease->left_sibling = H->min->left_sibling;
H->min->left_sibling = node_to_be_decrease;

node_to_be_decrease->parent = NULL;
node_to_be_decrease->mark = false;
}

void cascading_cut(FIB_HEAP *H, NODE *parent_node)
{
NODE *aux;
```

```
aux = parent_node->parent;
if (aux != NULL)
{
if (parent_node->mark == false)
{
parent_node->mark = true;
}
else
{
cut(H, parent_node, aux);
cascading_cut(H, aux);
}
}
}

void decrease_key(FIB_HEAP *H, NODE *node_to_be_decrease, int new_key)
{
NODE *parent_node;
if (H == NULL)
{
printf("\n FIbonacci heap not created ");
return;
}
if (node_to_be_decrease == NULL)
{
printf("Node is not in the heap");
}
```

```
else
{
if (node_to_be_decrease->key < new_key)
{
printf("\n Invalid new key for decrease key operation \n ");
}
else
{
node_to_be_decrease->key = new_key;
parent_node = node_to_be_decrease->parent;
if ((parent_node != NULL) && (node_to_be_decrease->key < parent_node->key))
{
printf("\n cut called");
cut(H, node_to_be_decrease, parent_node);
printf("\n cascading cut called");
cascading_cut(H, parent_node);
}
if (node_to_be_decrease->key < H->min->key)
{
H->min = node_to_be_decrease;
}
}
}
}

void *find_node(FIB_HEAP *H, NODE *n, int key, int new_key)
{
NODE *find_use = n;
```

```
NODE *f = NULL;
find_use->visited = true;
if (find_use->key == key)
{
find_use->visited = false;
f = find_use;
decrease_key(H, f, new_key);
}
if (find_use->child != NULL)
{
find_node(H, find_use->child, key, new_key);
}
if ((find_use->right_sibling->visited != true))
{
find_node(H, find_use->right_sibling, key, new_key);
}

find_use->visited = false;
}

FIB_HEAP *insertion_procedure()
{
FIB_HEAP *temp;
int no_of_nodes, ele, i;
NODE *new_node;
temp = (FIB_HEAP *)malloc(sizeof(FIB_HEAP));
temp = NULL;
if (temp == NULL)
```

```
{
temp = make_fib_heap();
}
printf(" \n enter number of nodes to be insert = ");
scanf("%d", &no_of_nodes);
for (i = 1; i <= no_of_nodes; i++)
{
printf("\n node %d and its key value = ", i);
scanf("%d", &ele);
insertion(temp, new_node, ele);
}
return temp;
}
void Delete_Node(FIB_HEAP *H, int dec_key)
{
NODE *p = NULL;
find_node(H, H->min, dec_key, -5000);
p = extract_min(H);
if (p != NULL)
printf("\n Node deleted");
else
printf("\n Node not deleted:some error");
}

int main(int argc, char **argv)
{
NODE *new_node, *min_node, *extracted_min, *node_to_be_decrease, *find_use;
FIB_HEAP *heap, *h1, *h2;
```

```
int operation_no, new_key, dec_key, ele, i, no_of_nodes;
heap = (FIB_HEAP *)malloc(sizeof(FIB_HEAP));
heap = NULL;
while (1)
{

printf(" \n choose below operations \n 1. Create Fibonacci heap \n 2. Insert nodes into fibonacci heap \n 3. Find min \n 4. Union \n 5. Extract min \n 6. Decrease key \n 7.Delete node \n 8. print heap \n 9. exit \n enter operation_no = ");
scanf("%d", &operation_no);

switch (operation_no)
{
case 1:
heap = make_fib_heap();
break;

case 2:
if (heap == NULL)
{
heap = make_fib_heap();
}
printf(" enter number of nodes to be insert = ");
scanf("%d", &no_of_nodes);
for (i = 1; i <= no_of_nodes; i++)
{
printf("\n node %d and its key value = ", i);
scanf("%d", &ele);
insertion(heap, new_node, ele);
```

```
}
break;

case 3:
min_node = find_min_node(heap);
if (min_node == NULL)
printf("No minimum value");
else
printf("\n min value = %d", min_node->key);
break;

case 4:
if (heap == NULL)
{
printf("\n no FIbonacci heap is created please create fibonacci heap \n ");
break;
}
h1 = insertion_procedure();
heap = unionHeap(heap, h1);
printf("Unified Heap:\n");
new_print_heap(heap->min);
break;

case 5:
if (heap == NULL)
printf("Fibonacci heap is empty");
else
{
```

```
extracted_min = extract_min(heap);
printf("\n min value = %d", extracted_min->key);
printf("\n Updated heap: \n");
new_print_heap(heap->min);
}
break;

case 6:
if (heap == NULL)
printf("Fibonacci heap is empty");
else
{
printf(" \n node to be decreased = ");
scanf("%d", &dec_key);
printf(" \n enter the new key = ");
scanf("%d", &new_key);
find_use = heap->min;
find_node(heap, find_use, dec_key, new_key);
printf("\n Key decreased- Corresponding heap:\n");
new_print_heap(heap->min);
}
break;
case 7:
if (heap == NULL)
printf("Fibonacci heap is empty");
else
{
printf(" \n Enter node key to be deleted = ");
```

```
scanf("%d", &dec_key);
Delete_Node(heap, dec_key);
printf("\n Node Deleted- Corresponding heap:\n");
new_print_heap(heap->min);
break;
}
case 8:
new_print_heap(heap->min);
break;

case 9:
free(new_node);
free(heap);
exit(0);

default:
printf("Invalid choice ");
}
}
}
```

Sample Input:

Choose below operations:

1. Create Fibonacci heap

2. Insert nodes into Fibonacci heap

3. Find minimum

4. Union

5. Extract minimum

6. Decrease key

7. Delete node

8. Print heap

9. Exit

Enter operation_no = 2

Enter number of nodes to insert = 3

Node 1 and its key value = 10

Node 2 and its key value = 5

Node 3 and its key value = 15

Sample output:

Choose below operations:

1. Create Fibonacci heap

2. Insert nodes into Fibonacci heap

3. Find minimum

4. Union

5. Extract minimum

6. Decrease key

7. Delete node

8. Print heap

9. Exit

Enter operation_no = 3

Minimum value = 5

Result:

The Fibonacci Heap was successfully implemented, allowing for efficient insertion, extraction of minimum, finding minimum, union, decrease key, deletion, and printing of the heap structure. Each operation follows the theoretical principles of Fibonacci Heaps, optimizing performance for complex data structure management.

Task No.	2.1	**B Tree: Implement disk-based operations by tracking the number of page accesses required for various B-Tree operations. Experiment with different node sizes and order values to optimize disk I/O efficiency.**
Date:	07-08-2025	

Aim:

To implement a B-tree structure in C that allows insertion, traversal, and deletion operations. This B-tree code defines a B-tree node structure and includes functions to create nodes, insert elements, split nodes, and traverse the B-tree.

Procedure:

1. **Define B-tree Node Structure**:
    - Define the B-tree node as a structure containing data pointers, child pointers, level, and node count.
2. **Initialize Nodes**:
    - Create an init() function to initialize each new node with child pointers set to NULL, leaf status, and zero elements.
3. **Insert Elements**:
    - Create an insert() function that checks if the root is full and, if necessary, splits it before proceeding with the insertion in the appropriate position.
    - If the insertion results in an overflow (more than m-1 keys), split the node to maintain B-tree properties.
4. **Split Child**:
    - Use split_child() to manage node overflow, splitting nodes when they exceed the order capacity.
    - Redistribute elements and pointers between new and existing nodes to maintain the tree balance.
5. **Traverse the B-tree**:
    - The traverse() function recursively displays the elements of the B-tree, showing the hierarchical structure.
6. **Sort Elements**:
    - Use a sort() function after each insertion to maintain sorted order within nodes.
7. **Execution**:

- Insert a series of elements, print confirmation of insertion, and then traverse the tree to display the B-tree structure.

Algorithm:

1. Define B-tree node structure with data, child pointers, level, and number of elements.
2. **Init()**:
   - Allocate memory and initialize child pointers to NULL.
3. **Insert(x)**:
   - If the root node is NULL, initialize it.
   - Traverse down the B-tree to find the appropriate position.
   - If any node becomes full during insertion, split it.
4. **Split_child()**:
   - Create a new node, move half the elements and child pointers, and connect the nodes correctly.
5. **Traverse()**:
   - Recursively visit each node to display all elements in sorted order.

Program:

```
#include <stdio.h>
#include <stdlib.h>
struct BTree {
//node declaration
int *d;
struct BTree **child_ptr;
int l;
int n;
};
struct BTree *r = NULL;
struct BTree *np = NULL;
struct BTree *x = NULL;
//creation of node
struct BTree* init() {
int i;
np = (struct BTree*)malloc(sizeof(struct BTree));
//order 6
np->d = (int*)malloc(6 * sizeof(int));
np->child_ptr = (struct BTree**)malloc(7 * sizeof(struct BTree*));
np->l = 1;
np->n = 0;
for (i = 0; i < 7; i++) {
np->child_ptr[i] = NULL;
}
return np;
}
//traverse the tree
```

```
void traverse(struct BTree *p) {
printf("\n");
int i;
for (i = 0; i < p->n; i++) {
if (p->l == 0) {
traverse(p->child_ptr[i]);
}
printf(" %d", p->d[i]);
}
if (p->l == 0) {
traverse(p->child_ptr[i]);
}
printf("\n");
}
//sort the tree
void sort(int *p, int n) {
int i, j, t;
for (i = 0; i < n; i++) {
for (j = i; j <= n; j++) {
if (p[i] > p[j]) {
t = p[i];
p[i] = p[j];
p[j] = t;
}
}
}
}
int split_child(struct BTree *x, int i) {
```

```
int j, mid;
struct BTree *np1, *np3, *y;
np3 = init();
//create new node
np3->l = 1;
if (i == -1) {
mid = x->d[2];
//find mid
x->d[2] = 0;
x->n--;
np1 = init();
np1->l = 0;
x->l = 1;
for (j = 3; j < 6; j++) {
np3->d[j - 3] = x->d[j];
np3->child_ptr[j - 3] = x->child_ptr[j];
np3->n++;
x->d[j] = 0;
x->n--;
}
for (j = 0; j < 6; j++) {
x->child_ptr[j] = NULL;
}
np1->d[0] = mid;
np1->child_ptr[np1->n] = x;
np1->child_ptr[np1->n + 1] = np3;
np1->n++;
r = np1;
```

```
} else {
y = x->child_ptr[i];
mid = y->d[2];
y->d[2] = 0;
y->n--;
for (j = 3; j < 6; j++) {
np3->d[j - 3] = y->d[j];
np3->n++;
y->d[j] = 0;
y->n--;
}
x->child_ptr[i + 1] = y;
x->child_ptr[i + 1] = np3;
}
return mid;
}
void insert(int a) {
int i, t;
x = r;
if (x == NULL) {
r = init();
x = r;
} else {
if (x->l == 1 && x->n == 6) {
t = split_child(x, -1);
x = r;
for (i = 0; i < x->n; i++) {
if (a > x->d[i] && a < x->d[i + 1]) {
```

```
i++;
break;
} else if (a < x->d[0]) {
break;
} else {
continue;
}
}
x = x->child_ptr[i];
} else {
while (x->l == 0) {
for (i = 0; i < x->n; i++) {
if (a > x->d[i] && a < x->d[i + 1]) {
i++;
break;
} else if (a < x->d[0]) {
break;
} else {
continue;
}
}
if (x->child_ptr[i]->n == 6) {
t = split_child(x, i);
x->d[x->n] = t;
x->n++;
continue;
} else {
x = x->child_ptr[i];
```

```
}
}
}
}
x->d[x->n] = a;
sort(x->d, x->n);
x->n++;
}
int main() {
int i, n, t;
insert(10);
insert(20);
insert(30);
insert(40);
insert(50);
printf("Insertion Done");
printf("\nB tree:");
traverse(r);
return 0;
}
```

Sample Input:

Insert: 10, 20, 30, 40, 50

Traverse the B-tree

Sample Output:

Insertion Done

B-tree elements:

10 20 30 40 50

Result:

The B-tree has been successfully implemented and tested with insertion and traversal functions, resulting in a sorted hierarchical structure that maintains the B-tree properties. This code can be extended to include deletion and other B-tree operations.

Task No.	2.2	**Splay Tree: Implement the operations to find the kth smallest or largest element in a Splay Tree. Test your implementation with various values of k.**
Date:	14-08-2025	

Aim:

To implement a Splay Tree in C that performs insertion, splaying, and rotation operations on a binary search tree. A Splay Tree is a self-adjusting binary search tree where recently accessed elements are moved closer to the root, improving access time for frequently accessed elements.

Procedure:

1. **Node Definition**: Define a node structure with data, left, and right pointers.
2. **Node Creation**: Implement a newNode function to initialize a node with given data and NULL children.
3. **Rotations**:
    - Implement right and left rotations to restructure the tree during splaying.
4. **Splay Operation**:
    - Splay operation brings a specific node to the root using rotations, improving tree efficiency.
5. **Insertion**:
    - Insert a new node by first splaying the tree at the new node's location and placing it appropriately.
6. **Tree Traversal**:
    - Print the Splay Tree using an in-order traversal to display its structure.

Algorithm:

1. Check if the tree is empty; if so, create a new root node with the given key.
2. Splay the tree to bring the node with value closest to the key to the root.
3. If the root's data matches the key, no further action is needed.
4. If the key is less than the root's data, insert the new node to the left.
5. Otherwise, insert it to the right.
6. If the root is NULL or already holds the desired data, return the root.
7. If the key is less than the root's data:
    a. If the left child is NULL, return the root.

b. Apply double rotations (zig-zig or zig-zag) to bring the desired node closer to the root.

8. If the key is greater than the root's data:
    c. If the right child is NULL, return the root.
    d. Apply double rotations (zag-zig or zag-zag) to bring the desired node closer to the root.
9. Perform an in-order traversal to print the tree's elements.

Program:

```
#include <stdio.h>
#include <stdlib.h>
struct node {
int data;
struct node *leftChild, *rightChild;
};
struct node* newNode(int data){
struct node* Node = (struct node*)malloc(sizeof(struct node));
Node->data = data;
Node->leftChild = Node->rightChild = NULL;
return (Node);
}
struct node* rightRotate(struct node *x){
struct node *y = x->leftChild;
x->leftChild = y->rightChild;
y->rightChild = x;
return y;
}
struct node* leftRotate(struct node *x){
struct node *y = x->rightChild;
x->rightChild = y->leftChild;
```

```
y->leftChild = x;
return y;
}
struct node* splay(struct node *root, int data){
if (root == NULL || root->data == data)
return root;
if (root->data > data) {
if (root->leftChild == NULL) return root;
if (root->leftChild->data > data) {
root->leftChild->leftChild = splay(root->leftChild->leftChild, data);
root = rightRotate(root);
} else if (root->leftChild->data < data) {
root->leftChild->rightChild = splay(root->leftChild->rightChild, data);
if (root->leftChild->rightChild != NULL)
root->leftChild = leftRotate(root->leftChild);
}
return (root->leftChild == NULL)? root: rightRotate(root);
} else {
if (root->rightChild == NULL) return root;
if (root->rightChild->data > data) {
root->rightChild->leftChild = splay(root->rightChild->leftChild, data);
if (root->rightChild->leftChild != NULL)
root->rightChild = rightRotate(root->rightChild);
} else if (root->rightChild->data < data) {
root->rightChild->rightChild = splay(root->rightChild->rightChild, data);
root = leftRotate(root);
}
return (root->rightChild == NULL)? root: leftRotate(root);
```

```
}
}
struct node* insert(struct node *root, int k){
if (root == NULL) return newNode(k);
root = splay(root, k);
if (root->data == k) return root;
struct node *newnode = newNode(k);
if (root->data > k) {
newnode->rightChild = root;
newnode->leftChild = root->leftChild;
root->leftChild = NULL;
} else {
newnode->leftChild = root;
newnode->rightChild = root->rightChild;
root->rightChild = NULL;
}
return newnode;
}
void printTree(struct node *root){
if (root == NULL)
return;
if (root != NULL) {
printTree(root->leftChild);
printf("%d ", root->data);
printTree(root->rightChild);
}
}
int main(){
```

```
struct node* root = newNode(34);
root->leftChild = newNode(15);
root->rightChild = newNode(40);
root->leftChild->leftChild = newNode(12);
root->leftChild->leftChild->rightChild = newNode(14);
root->rightChild->rightChild = newNode(59);
printf("The Splay tree is: \n");
printTree(root);
return 0;
}
```

Sample Input:

Inserting nodes into the tree with the following values in sequence:

- 15, 40, 12, 14, 59

Sample Output:

The Splay tree is:

12 14 15 34 40 59

Result:

The implemented Splay Tree program successfully constructs and displays the in-order traversal of the initial tree structure. The output of the traversal is 12 14 15 34 40 59, representing the sorted order of nodes in the tree.

Task No.	3.1	Implement and analyze cryptographic hash functions like SHA-256.
Date:	21-08-2025	

Aim:

To implement the SHA-256 cryptographic hash algorithm in C to generate a unique, fixed-size 256-bit hash value for a given input message.

Procedure:

1. Define the SHA-256 constants, functions, and operations necessary for the hash algorithm, including the initial hash values, round constants, and bitwise rotation operations.
2. Create a structure for the SHA-256 context that includes the current hash state, message length, and a buffer for the message block.
3. Initialize the SHA-256 context, setting the initial hash values and initializing the length and buffer.
4. Implement the sha256_transform() function to perform the main hash computation on each 512-bit block of the message.
5. Define the sha256_update() function to process each block of the message, dividing it into 512-bit chunks and passing each chunk to sha256_transform().
6. Implement the sha256_final() function to complete the hash computation by adding padding, processing any remaining data, and appending the length of the original message.
7. In the main() function, initialize the SHA-256 context, update it with the input message, finalize the hash, and print the resulting 256-bit hash value in hexadecimal.

Algorithm:

Step 1. Define SHA-256 constants, rotation, and compression functions.

Step 2. Define SHA-256 context structure with state, length, and buffer.

Step 3. Initialize context with initial hash values and set length to zero.

Step 4. Divide 512-bit block into 32-bit words in sha256_transform() function.

Step 5. Extend words to form a 64-entry message schedule.

Step 6. Initialize a, b, c, d, e, f, g, h with current state.

Step 7. Loop 64 times:

- Compute t1 using compression with SHA-256 functions.
- Compute t2 as a sum of Sigma0(a) and Maj(a, b, c).
- Update working variables by shifting through a-h.

Step 8. Update state values with results of each iteration.

Step 9. Append data to buffer until 512-bit block is ready in sha256_update() function.

Step 10. Process full blocks using sha256_transform().

Step 11. Add padding "1" bit and "0" bits to finalize block in sha256_final() function.

Step 12. Append 64-bit original message length.

Step 13. Process final 512-bit block with sha256_transform().

Step 14. Store resulting hash in output array.

Step 15. Initialize context, update with sample message, finalize, and print hash in Main.

Program:

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>

// Define SHA-256 constants
#define SHA256_BLOCK_SIZE 64
#define SHA256_DIGEST_SIZE 32

// Rotate right (circular right shift) operation
#define ROTR(x, n) (((x) >> (n)) | ((x) << (32 - (n))))

// SHA-256 functions
#define Ch(x, y, z) (((x) & (y)) ^ (~(x) & (z)))
#define Maj(x, y, z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))
#define Sigma0(x) (ROTR(x, 2) ^ ROTR(x, 13) ^ ROTR(x, 22))
#define Sigma1(x) (ROTR(x, 6) ^ ROTR(x, 11) ^ ROTR(x, 25))
#define sigma0(x) (ROTR(x, 7) ^ ROTR(x, 18) ^ ((x) >> 3))
#define sigma1(x) (ROTR(x, 17) ^ ROTR(x, 19) ^ ((x) >> 10))

// Initial hash values for SHA-256
const uint32_t sha256_initial[] = {
0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A,
0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19
};

// SHA-256 context structure
typedef struct {
```

```
uint32_t state[8];
uint64_t length;
uint8_t buffer[SHA256_BLOCK_SIZE];
} SHA256_CTX;

// SHA-256 transformation
static void sha256_transform(SHA256_CTX *ctx) {
uint32_t w[64];
uint32_t a, b, c, d, e, f, g, h;
uint32_t t1, t2;

for (int i = 0; i < 16; i++) {
w[i] = (ctx->buffer[i * 4] << 24) |
(ctx->buffer[i * 4 + 1] << 16) |
(ctx->buffer[i * 4 + 2] << 8) |
(ctx->buffer[i * 4 + 3]);
}

for (int i = 16; i < 64; i++) {
w[i] = sigma1(w[i - 2]) + w[i - 7] + sigma0(w[i - 15]) + w[i - 16];
}

a = ctx->state[0];
b = ctx->state[1];
c = ctx->state[2];
d = ctx->state[3];
e = ctx->state[4];
f = ctx->state[5];
```

```
g = ctx->state[6];
h = ctx->state[7];

for (int i = 0; i < 64; i++) {
t1 = h + Sigma1(e) + Ch(e, f, g) + sha256_initial[i] + w[i];
t2 = Sigma0(a) + Maj(a, b, c);
h = g;
g = f;
f = e;
e = d + t1;
d = c;
c = b;
b = a;
a = t1 + t2;
}

ctx->state[0] += a;
ctx->state[1] += b;
ctx->state[2] += c;
ctx->state[3] += d;
ctx->state[4] += e;
ctx->state[5] += f;
ctx->state[6] += g;
ctx->state[7] += h;
}

// Initialize SHA-256 context
void sha256_init(SHA256_CTX *ctx) {
```

```
memcpy(ctx->state, sha256_initial, sizeof(sha256_initial));
ctx->length = 0;
}

// Update SHA-256 context with input data
void sha256_update(SHA256_CTX *ctx, const uint8_t *data, size_t len) {
size_t block_size = SHA256_BLOCK_SIZE;
size_t index = (size_t)(ctx->length % block_size);
ctx->length += len;

if (index != 0 && index + len >= block_size) {
size_t first_part = block_size - index;
memcpy(&ctx->buffer[index], data, first_part);
sha256_transform(ctx);
data += first_part;
len -= first_part;
index = 0;
}

while (len >= block_size) {
memcpy(ctx->buffer, data, block_size);
sha256_transform(ctx);
data += block_size;
len -= block_size;
}

if (len > 0) {
memcpy(&ctx->buffer[index], data, len);
```

```
}
}

// Finalize SHA-256 hash and output the result
void sha256_final(SHA256_CTX *ctx, uint8_t *hash) {
size_t block_size = SHA256_BLOCK_SIZE;
size_t index = (size_t)(ctx->length % block_size);
ctx->buffer[index++] = 0x80;

if (index > block_size - 8) {
memset(&ctx->buffer[index], 0, block_size - index);
sha256_transform(ctx);
index = 0;
}

memset(&ctx->buffer[index], 0, block_size - 8 - index);
ctx->length *= 8;

for (int i = 0; i < 8; i++) {
ctx->buffer[block_size - 8 + i] = (uint8_t)(ctx->length >> ((7 - i) * 8));
}

sha256_transform(ctx);

for (int i = 0; i < 8; i++) {
hash[i * 4] = (uint8_t)(ctx->state[i] >> 24);
hash[i * 4 + 1] = (uint8_t)(ctx->state[i] >> 16);
hash[i * 4 + 2] = (uint8_t)(ctx->state[i] >> 8);
```

```
hash[i * 4 + 3] = (uint8_t)(ctx->state[i]);
}
}

int main() {
// Example usage
SHA256_CTX ctx;
uint8_t hash[SHA256_DIGEST_SIZE];

sha256_init(&ctx);
sha256_update(&ctx, "Hello, world!", strlen("Hello, world!"));
sha256_final(&ctx, hash);

printf("SHA-256 Hash: ");
for (int i = 0; i < SHA256_DIGEST_SIZE; i++) {
printf("%02x", hash[i]);
}
printf("\n");

return 0;
}
```

Sample Input:

Input message: "Hello, world!"

Sample Output:

SHA-256 Hash:
c0535e4be2b79ffd93291305436bf889314e4a3faec05ecffcbb9eab1eebe4dc

Result:

The SHA-256 hash algorithm was successfully implemented and tested. The output hash provides a unique, fixed-size representation of the input message, ensuring data integrity and security.

Task No.	3.2	**Implement the Rabin-Karp string searching algorithm that uses hashing to efficiently search for a pattern within a text.**
Date:	21-08-2025	

Aim:

To implement the Rabin-Karp string matching algorithm to search for a given pattern within a specified text.

Procedure:

1. Define constants for the alphabet size and a prime number to compute hash values.
2. Initialize a hash value for the pattern and the first text window.
3. Slide the pattern over the text to check for matches using hash comparisons.
4. Adjust the hash for each text window efficiently to minimize comparisons.
5. Output all positions where the pattern is found in the text.

Algorithm:

Step 1. Define alphabet size `d` and prime number `q`.

Step 2. Define function `searchRabinKarp` to search pattern in text.

Step 3. Initialize variables for lengths of pattern `M` and text `N`.

Step 4. Initialize hash values `p` and `t` for pattern and text.

Step 5. Initialize variable `h` as 1 for recalculating hash.

Step 6. Compute `h` as `d^(M-1) % q` to use in hash shifting.

Step 7. Calculate initial hash values for pattern and first text window.

Step 8. Slide pattern over text by checking hash values.

Step 9. If hashes match, check each character in the current window.

Step 10. Print pattern position if characters match.

Step 11. Calculate next window hash by adjusting only first and next characters.

Step 12. Return to main, where `searchRabinKarp` function is called with a sample text and pattern.

Program:

```
#include <stdio.h>
#include <string.h>

#define d 256 // The number of characters in the input alphabet
#define q 101 // A prime number for the hash function

// Function to search for the pattern in the text using Rabin-Karp algorithm
void searchRabinKarp(char pattern[], char text[]) {
int M = strlen(pattern);
int N = strlen(text);
int i, j;
int p = 0; // Hash value for the pattern
int t = 0; // Hash value for the text
int h = 1;

// Calculate the hash value of h
for (i = 0; i < M - 1; i++)
h = (h * d) % q;

// Calculate the hash value of the pattern and the first window of the text
for (i = 0; i < M; i++) {
p = (d * p + pattern[i]) % q;
t = (d * t + text[i]) % q;
}

// Slide the pattern over the text one character at a time
for (i = 0; i <= N - M; i++) {
```

```
// Check the hash values of the current window of the text and the pattern
if (p == t) {
// If hash values match, check each character individually
for (j = 0; j < M; j++) {
if (text[i + j] != pattern[j])
break;
}

// If all characters match, print the position
if (j == M)
printf("Pattern found at index %d\n", i);
}

// Calculate the hash value for the next window of the text
if (i < N - M) {
t = (d * (t - text[i] * h) + text[i + M]) % q;
if (t < 0)
t = (t + q);
}
}
}

int main() {
char text[] = "AABAACAADAABAABA";
char pattern[] = "AABA";
searchRabinKarp(pattern, text);
return 0;
}
```

Sample Input:

Text: "AABAACAADAABAABA"

Pattern: "AABA"

Sample Output:

Pattern found at index 0

Pattern found at index 9

Pattern found at index 12

Result:

The Rabin-Karp algorithm successfully searched and identified all occurrences of the pattern within the given text.

Task No.	4.1	**Implement the Strassen's algorithm for matrix multiplication using the divide and conquer approach.**
Date:	28-08-2025	

Aim:

To perform matrix multiplication using Strassen's algorithm to multiply two matrices efficiently.

Procedure:

1. Define functions to add and subtract matrices.
2. Split each matrix into four quadrants recursively.
3. Apply Strassen’s formula to compute the products.
4. Combine the resulting quadrants to form the final matrix.
5. Print the result matrix.

Algorithm:

Step 1. Define the matrix size as a power of 2.

Step 2. Define functions `add`, `subtract`, and `strassen` for matrix operations.

Step 3. Split matrices `A` and `B` into four quadrants each.

Step 4. Compute temporary matrices required for the products using `add` and `subtract`.

Step 5. Apply Strassen’s algorithm to calculate the seven products \(P1\) to \(P7\).

Step 6. Compute the resulting quadrants \(C11\), \(C12\), \(C21\), \(C22\) by combining products.

Step 7. Copy results from quadrants back into result matrix `C`.

Step 8. Print matrix `C` as the output of matrix multiplication.

Step 9. Return to main, where `strassen` is called with sample matrices `A` and `B`.

Program:

```
#include <stdio.h>

// Function to add two matrices
void add(int n, int A[n][n], int B[n][n], int C[n][n]) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
C[i][j] = A[i][j] + B[i][j];
}

// Function to subtract two matrices
void subtract(int n, int A[n][n], int B[n][n], int C[n][n]) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
C[i][j] = A[i][j] - B[i][j];
}

// Function to multiply two matrices using Strassen's algorithm
void strassen(int n, int A[n][n], int B[n][n], int C[n][n]) {
if (n == 1) {
C[0][0] = A[0][0] * B[0][0];
return;
}

// Split matrices into quadrants
int newSize = n / 2;
int A11[newSize][newSize], A12[newSize][newSize], A21[newSize][newSize],
A22[newSize][newSize];
```

```
int B11[newSize][newSize], B12[newSize][newSize], B21[newSize][newSize],
B22[newSize][newSize];

int C11[newSize][newSize], C12[newSize][newSize], C21[newSize][newSize],
C22[newSize][newSize];

int P1[newSize][newSize], P2[newSize][newSize], P3[newSize][newSize],
P4[newSize][newSize];

int P5[newSize][newSize], P6[newSize][newSize], P7[newSize][newSize];

int temp1[newSize][newSize], temp2[newSize][newSize];


// Partition input matrices

for (int i = 0; i < newSize; i++)

for (int j = 0; j < newSize; j++) {

A11[i][j] = A[i][j];

A12[i][j] = A[i][j + newSize];

A21[i][j] = A[i + newSize][j];

A22[i][j] = A[i + newSize][j + newSize];


B11[i][j] = B[i][j];

B12[i][j] = B[i][j + newSize];

B21[i][j] = B[i + newSize][j];

B22[i][j] = B[i + newSize][j + newSize];

}


// Compute products and subproducts

add(newSize, A11, A22, temp1);

add(newSize, B11, B22, temp2);

strassen(newSize, temp1, temp2, P1);


add(newSize, A21, A22, temp1);
```

```
strassen(newSize, temp1, B11, P2);

subtract(newSize, B12, B22, temp1);
strassen(newSize, A11, temp1, P3);

subtract(newSize, B21, B11, temp1);
strassen(newSize, A22, temp1, P4);

add(newSize, A11, A12, temp1);
strassen(newSize, temp1, B22, P5);

subtract(newSize, A21, A11, temp1);
add(newSize, B11, B12, temp2);
strassen(newSize, temp1, temp2, P6);

subtract(newSize, A12, A22, temp1);
add(newSize, B21, B22, temp2);
strassen(newSize, temp1, temp2, P7);

// Compute C11, C12, C21, C22
add(newSize, P1, P4, temp1);
subtract(newSize, temp1, P5, temp2);
add(newSize, temp2, P7, C11);

add(newSize, P3, P5, C12);
add(newSize, P2, P4, C21);

subtract(newSize, P1, P2, temp1);
```

```
add(newSize, temp1, P3, temp2);
add(newSize, temp2, P6, C22);

// Copy results back to C
for (int i = 0; i < newSize; i++)
for (int j = 0; j < newSize; j++) {
C[i][j] = C11[i][j];
C[i][j + newSize] = C12[i][j];
C[i + newSize][j] = C21[i][j];
C[i + newSize][j + newSize] = C22[i][j];
}
}

// Function to print a matrix
void printMatrix(int n, int mat[n][n]) {
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++)
printf("%d\t", mat[i][j]);
printf("\n");
}
}

int main() {
int n = 4; // Matrix size (should be a power of 2)

// Example matrices A and B
int A[4][4] = {
{1, 2, 3, 4},
```

```
{5, 6, 7, 8},
{9, 10, 11, 12},
{13, 14, 15, 16}
};

int B[4][4] = {
{17, 18, 19, 20},
{21, 22, 23, 24},
{25, 26, 27, 28},
{29, 30, 31, 32}
};

// Resultant matrix C
int C[4][4];

// Perform matrix multiplication using Strassen's algorithm
strassen(n, A, B, C);

// Print the matrices
printf("Matrix A:\n");
printMatrix(n, A);
printf("\nMatrix B:\n");
printMatrix(n, B);

printf("\nMatrix C (Result of A * B using Strassen's algorithm):\n");
printMatrix(n, C);
return 0;
}
```

Sample Input:

Matrix A:

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 16

Matrix B:

17 18 19 20

21 22 23 24

25 26 27 28

29 30 31 32

Sample Output:

Matrix C (Result of A * B using Strassen's algorithm):

250 260 270 280

618 644 670 696

986 1028 1070 1112

1354 1412 1470 1528

Result:

The program successfully multiplies two matrices using Strassen's algorithm and outputs the resultant matrix.

Task No.	4.2	**Implement merge sort to count the number of inversions in an array using divide and conquer strategy**
Date:	04-09-2025	

Aim:

To count the number of inversions in an array using the merge sort algorithm, where an inversion is a condition in which a pair of elements is out of order.

Procedure:

1. Implement a merge sort function with an additional inversion count.

2. Divide the array recursively until each subarray has one element.

3. During merging, count each inversion when elements from the right subarray are less than elements in the left subarray.

4. Return the total inversion count and print the sorted array.

Algorithm:

Step 1. Define the `mergeAndCount` function to merge two subarrays and count inversions.

Step 2. Calculate the sizes of left and right subarrays, and create temporary arrays `L` and `R`.

Step 3. Copy elements from the main array to `L` and `R`.

Step 4. Initialize `inversionCount` to zero, and merge elements in sorted order.

Step 5. If `L[i] > R[j]`, increment `inversionCount` by remaining elements in `L`.

Step 6. Copy remaining elements from `L` and `R` to the main array.

Step 7. Free temporary arrays and return `inversionCount`.

Step 8. Define `mergeSortAndCount` to divide and merge subarrays, adding up inversions.

Step 9. Call `mergeSortAndCount` recursively on left and right subarrays.

Step 10. Return the total inversion count.

Step 11. In `main`, print the original array and inversion count, then print the sorted array.

Program:

```
#include <stdio.h>
#include <stdlib.h>

// Function to merge two sorted subarrays and count inversions
long long mergeAndCount(int arr[], int left, int mid, int right) {
int n1 = mid - left + 1;
int n2 = right - mid;

// Create temporary arrays
int* L = (int*)malloc(n1 * sizeof(int));
int* R = (int*)malloc(n2 * sizeof(int));

// Copy data to temporary arrays L[] and R[]
for (int i = 0; i < n1; i++)
L[i] = arr[left + i];
for (int j = 0; j < n2; j++)
R[j] = arr[mid + 1 + j];

// Merge the two sorted arrays and count inversions
long long inversionCount = 0;
int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {
if (L[i] <= R[j]) {
arr[k++] = L[i++];
} else {
arr[k++] = R[j++];
```

```
// Count inversions
inversionCount += (mid - left + 1) - i;
}
}

// Copy the remaining elements of L[], if there are any
while (i < n1) {
arr[k++] = L[i++];
}

// Copy the remaining elements of R[], if there are any
while (j < n2) {
arr[k++] = R[j++];
}

// Free the temporary arrays
free(L);
free(R);

return inversionCount;
}

// Function to perform merge sort and count inversions
long long mergeSortAndCount(int arr[], int left, int right) {
long long inversionCount = 0;

if (left < right) {
int mid = left + (right - left) / 2;
```

```
// Recursively sort and count inversions in the two halves
inversionCount += mergeSortAndCount(arr, left, mid);
inversionCount += mergeSortAndCount(arr, mid + 1, right);

// Merge the sorted halves and count inversions
inversionCount += mergeAndCount(arr, left, mid, right);
}

return inversionCount;
}

// Function to count inversions in an array using merge sort
long long countInversions(int arr[], int size) {
return mergeSortAndCount(arr, 0, size - 1);
}

int main() {
int arr[] = {1, 20, 6, 4, 5};
int size = sizeof(arr) / sizeof(arr[0]);

printf("Original array: ");
for (int i = 0; i < size; i++) {
printf("%d ", arr[i]);
}
printf("\n");

long long inversionCount = countInversions(arr, size);
```

```
printf("Number of inversions: %lld\n", inversionCount);

printf("Sorted array: ");
for (int i = 0; i < size; i++) {
printf("%d ", arr[i]);
}
printf("\n");

return 0;
}
```

Sample Input:

Array: {1, 20, 6, 4, 5}

Sample Output:

Original array: 1 20 6 4 5

Number of inversions: 5

Sorted array: 1 4 5 6 20

Result:

The program successfully counts the number of inversions in the array and outputs the sorted array.

Task No.	4.3	**Implement the Huffman coding algorithm for text compression using Greedy technique.**
Date:	04-09-2025	

Aim:

To implement Huffman Coding to generate variable-length codes for characters based on their frequencies, aiming to minimize the total number of bits used for encoding.

Procedure:

1. Create a structure for tree nodes, each holding a character, its frequency, and pointers to left and right child nodes.

2. Build the Huffman Tree by repeatedly merging nodes with the smallest frequencies until a single tree structure remains.

3. Traverse the Huffman Tree, assigning '0' for left branches and '1' for right branches to form binary codes for each character.

4. Print the Huffman Codes for each character based on their positions in the tree.

Algorithm:

Step 1. Define a `Node` structure with character, frequency, left, and right pointers.

Step 2. Define `createNode` to initialize and return a new node.

Step 3. Define `printCodes` to print codes from the Huffman Tree root by recursive traversal.

Step 4. In `printCodes`, assign '0' for left branches and '1' for right branches.

Step 5. Print the code when reaching a leaf node.

Step 6. Define `buildHuffmanTree` to create the Huffman Tree based on character frequencies.

Step 7. Initialize nodes in `nodeArray` with characters and frequencies.

Step 8. While more than one node remains, sort `nodeArray` by frequency.

Step 9. Merge nodes with the smallest frequencies to form a new node.

Step 10. Update `nodeArray` by removing merged nodes and adding the new internal node.

Step 11. Repeat merging until one node (tree root) remains in `nodeArray`.

Step 12. Define `huffmanCodes` to build the tree and print codes for each character.

Step 13. In `main`, define data and frequencies and call `huffmanCodes` to output results.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Define a structure for a Huffman tree node
struct Node {
char data;
unsigned frequency;
struct Node *left, *right;
};

// Function to create a new node
struct Node* createNode(char data, unsigned frequency) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = data;
newNode->frequency = frequency;
newNode->left = newNode->right = NULL;
return newNode;
}

// Function to print the Huffman codes from the root of the tree
void printCodes(struct Node* root, int arr[], int top, char data) {
// Assign 0 to left edge and recur
if (root->left) {
arr[top] = 0;
printCodes(root->left, arr, top + 1, data);
}
```

```
// Assign 1 to right edge and recur
if (root->right) {
arr[top] = 1;
printCodes(root->right, arr, top + 1, data);
}

// If this is a leaf node, print the code
if (!(root->left) && !(root->right) && root->data == data) {
printf("%c: ", data);
for (int i = 0; i < top; i++) {
printf("%d", arr[i]);
}
printf("\n");
}
}

// Function to build the Huffman tree
struct Node* buildHuffmanTree(char data[], unsigned frequency[], int size) {
struct Node *left, *right, *top;

// Create an array of nodes
struct Node* nodeArray[size];

// Initialize each node with its data and frequency
for (int i = 0; i < size; i++) {
nodeArray[i] = createNode(data[i], frequency[i]);
}
```

```
// Iterate until there is only one node in the array (the root of the Huffman tree)
while (size > 1) {
// Sort the nodes based on frequency
for (int i = 0; i < size - 1; i++) {
for (int j = 0; j < size - i - 1; j++) {
if (nodeArray[j]->frequency > nodeArray[j + 1]->frequency) {
struct Node* temp = nodeArray[j];
nodeArray[j] = nodeArray[j + 1];
nodeArray[j + 1] = temp;
}
}
}

// Create a new internal node with the two smallest frequencies as children
left = nodeArray[0];
right = nodeArray[1];
top = createNode('$', left->frequency + right->frequency);
top->left = left;
top->right = right;

// Remove the two smallest nodes from the array and add the new internal node
nodeArray[0] = top;
for (int i = 1; i < size - 1; i++) {
nodeArray[i] = nodeArray[i + 1];
}
size--;
}
// The remaining node is the root of the Huffman tree
```

```
return nodeArray[0];
}

// Function to perform Huffman coding
void huffmanCodes(char data[], unsigned frequency[], int size) {
// Build the Huffman tree
struct Node* root = buildHuffmanTree(data, frequency, size);

// Create an array to store Huffman codes
int arr[size];
int top = 0;

// Print Huffman codes for each character
for (int i = 0; i < size; i++) {
printCodes(root, arr, top, data[i]);
}
}

int main() {
char data[] = {'a', 'b', 'c', 'd', 'e', 'f'};
unsigned frequency[] = {5, 9, 12, 13, 16, 45};
int size = sizeof(data) / sizeof(data[0]);

printf("Huffman Codes:\n");
huffmanCodes(data, frequency, size);

return 0;
}
```

Sample Input:

Characters: {a, b, c, d, e, f}

Frequencies: {5, 9, 12, 13, 16, 45}

Sample Output:

Huffman Codes:

a: 1100

b: 1101

c: 100

d: 101

e: 111

f: 0

Result:

The program successfully generates and prints Huffman Codes for each character based on their frequencies, effectively compressing the data.

Task No.	4.4	**Implement longest common subsequence among multiple strings using dynamic programming**
Date:	11-09-2025	

Aim:

To find the length of the Longest Common Subsequence (LCS) between two strings using a recursive approach.

Procedure:

1. Define a recursive function `LCS` that takes two strings and their lengths as input.

2. In the function, check if either string length is zero; if so, return 0 as there is no common subsequence.

3. If the last characters of both strings match, recursively call `LCS` for the remaining parts of both strings, adding 1 to the result.

4. If the last characters do not match, find the maximum LCS by either excluding the last character of the first string or the last character of the second string.

5. The result is the length of the LCS of the two strings.

Algorithm:

Step 1. Define function `max` to return the larger of two numbers.

Step 2. Define the `LCS` function with inputs as two strings, `A` and `B`, and their lengths, `x` and `y`.

Step 3. Check if either `x` or `y` is 0; if so, return 0 as the base case.

Step 4. If `A[x-1]` is equal to `B[y-1]`, return 1 plus `LCS(A, B, x-1, y-1)`.

Step 5. If `A[x-1]` is not equal to `B[y-1]`, return `max(LCS(A, B, x, y-1), LCS(A, B, x-1, y))`.

Step 6. In `main`, initialize strings `A` and `B` and calculate their lengths `x` and `y`.

Step 7. Call `LCS(A, B, x, y)` and print the result.

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int max(int m, int n) {
if (m > n)
return m;
else
return n;
}

int LCS(char *A, char *B, int x, int y) {
if (x == 0 || y == 0)
return 0;
if (A[x - 1] == B[y - 1])
return 1 + LCS(A, B, x - 1, y - 1);
else
return max(LCS(A, B, x, y - 1), LCS(A, B, x - 1, y));
}

int main() {
char A[] = "XY";
char B[] = "XPYQ";
int x = strlen(A);
int y = strlen(B);
printf("Length of LCS is %d\n", LCS(A, B, x, y));
return 0;
}
```

Sample Input:

String A: "XY"

String B: "XPYQ"

Sample Output:

Length of LCS is 2

Result:

The program successfully calculates the length of the Longest Common Subsequence between the two strings using recursion. The LCS length for strings "XY" and "XPYQ" is 2.

Task No.	4.5	**Solve the coin change problem with a twist: instead of finding the number of ways to make change, find the minimum number of coins needed to make a certain amount.**
Date:	11-09-2025	

Aim:

To find the minimum number of coins required to make a specified amount using given coin denominations.

Procedure:

1. Define an array `dp[]` to store the minimum number of coins needed for each amount from 0 up to the target amount.

2. Initialize `dp[0]` to 0, since zero coins are required to make an amount of 0. Set all other entries in `dp[]` to a large value (infinity).

3. For each amount from 1 to the target amount, iterate over each coin denomination.

4. If the coin's value is less than or equal to the current amount, calculate if including this coin reduces the total number of coins needed.

5. Update `dp[amount]` with the minimum count found.

6. If `dp[amount]` contains infinity after processing, the amount cannot be made with the given coins.

Algorithm:

Step 1. Define the function `minCoins` that takes `coins[]`, `m`, and `amount` as inputs.

Step 2. Initialize the array `dp[]` with size `amount + 1`. Set `dp[0]` to 0 and all other values to infinity.

Step 3. For `i` from 1 to `amount`, do steps 4–6.

Step 4. For each coin in `coins[]`, check if the coin's value is less than or equal to `i`.

Step 5. If true, calculate `dp[i - coin] + 1` and update `dp[i]` if this value is smaller than the current `dp[i]`.

Step 6. After filling `dp[]`, return `dp[amount]` if it is not infinity; otherwise, return -1.

Step 7. In `main`, initialize `coins[]`, calculate `m`, and set `amount`.

Step 8. Call `minCoins` and print the result.

Program:

```
#include <stdio.h>
#include <limits.h>

// Function to find the minimum number of coins needed to make a certain amount
int minCoins(int coins[], int m, int amount) {
// Create a table to store the results of subproblems
int dp[amount + 1];

// Initialize the table with maximum value
for (int i = 0; i <= amount; i++) {
dp[i] = INT_MAX;
}

// Base case: 0 coins are needed to make 0 amount
dp[0] = 0;

// Iterate through all subproblems
for (int i = 1; i <= amount; i++) {
// Try each coin and update the result if using the current coin minimizes the count
for (int j = 0; j < m; j++) {
if (coins[j] <= i && dp[i - coins[j]] != INT_MAX) {
dp[i] = dp[i - coins[j]] + 1;
}
}
}

// The result is stored in dp[amount]
```

```
return (dp[amount] == INT_MAX) ? -1 : dp[amount];
}

int main() {
// Example: Coin denominations and the target amount
int coins[] = {1, 3, 5};
int m = sizeof(coins) / sizeof(coins[0]);
int amount = 11;

// Find the minimum number of coins needed
int result = minCoins(coins, m, amount);

// Print the result
if (result != -1) {
printf("Minimum number of coins needed: %d\n", result);
} else {
printf("It's not possible to make the amount with the given coins.\n");
}

return 0;
}
```

Sample Input:

Coins: [1, 3, 5]

Amount: 11

Sample Output:

Minimum number of coins needed: 3

Result:

The program successfully calculates the minimum number of coins required to make the specified amount using the given coin denominations. The minimum number of coins needed for the amount 11 with denominations [1, 3, 5] is 3.

Task No.	5.1	**Implement Kruskal's algorithm to find minimum spanning tree of a weighted graph.**
Date:	18-09-2025	

Aim:

To implement Kruskal's Algorithm to find the Minimum Spanning Tree (MST) of a connected, undirected graph using union-find.

Procedure:

1. Sort all edges of the graph in ascending order of their weights.

2. Initialize the parent and rank arrays for union-find operations.

3. For each edge in the sorted list, check if it forms a cycle with the MST formed so far.

4. If it doesn’t form a cycle, include the edge in the MST and update the union-find structure.

5. Repeat until we include (n-1) edges for a graph with `n` vertices.

6. The total weight of the MST is the sum of the included edges' weights.

Algorithm:

Step 1. Define a `cmp` function to compare edge weights for sorting.

Step 2. Define `initSet` to initialize the parent and rank arrays for union-find.

Step 3. Define `findParent` to find the representative of a set using path compression.

Step 4. Define `unionSet` to union two sets using rank-based union.

Step 5. Define `kruskal` function, where:

- Step 5.1. Sort all edges using `qsort`.

- Step 5.2. Initialize `parent` and `rank` arrays using `initSet`.

- Step 5.3. Initialize `minCost` to store the MST cost.

- Step 5.4. For each edge, find the parent of both vertices.

- Step 5.5. If they belong to different sets, union them, add edge to MST, and increment `minCost`.

Step 6. In `main`, define the edges and call `kruskal` to calculate the MST.

Program:

```
#include <stdio.h>
#include <stdlib.h>

int cmp(const void *p1, const void *p2) {
const int (*x)[3] = p1;
const int (*y)[3] = p2;
return (*x)[2] - (*y)[2];
}

void initSet(int parent[], int rank[], int n) {
for (int i = 0; i < n; ++i) {
parent[i] = i;
rank[i] = 0;
}
}

int findParent(int parent[], int v) {
return (parent[v] == v) ? v : (parent[v] = findParent(parent, parent[v]));
}

void unionSet(int u, int v, int parent[], int rank[]) {
u = findParent(parent, u);
v = findParent(parent, v);
if (rank[u] < rank[v]) parent[u] = v;
else if (rank[u] > rank[v]) parent[v] = u;
else {
parent[v] = u;
```

```
++rank[u];
}
}
void kruskal(int n, int edges[n][3]) {
qsort(edges, n, sizeof(edges[0]), cmp);
int parent[n], rank[n];
initSet(parent, rank, n);
int minCost = 0;

printf("Edges in the MST:\n");
for (int i = 0; i < n; ++i) {
int u = findParent(parent, edges[i][0]);
int v = findParent(parent, edges[i][1]);
int weight = edges[i][2];
if (u != v) {
unionSet(u, v, parent, rank);
minCost += weight;
printf("%d -- %d == %d\n", edges[i][0], edges[i][1], weight);
}
}
printf("Minimum Cost Spanning Tree: %d\n", minCost);
}
int main() {
int edges[5][3] = {{0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4}};
kruskal(5, edges);
return 0;
}
```

Sample Input:

Edges: {{0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4}}

Sample Output:

Edges in the MST:

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Minimum Cost Spanning Tree: 19

Result:

The program successfully finds the Minimum Spanning Tree (MST) of the given graph using Kruskal's Algorithm. The total weight of the MST is 19.

Task No.	5.2	Implement Prim's algorithm to find minimum spanning tree using priority queue
Date:	18-09-2025	

Aim:

To implement Prim's Algorithm using a priority queue to find the Minimum Spanning Tree (MST) of a connected, undirected graph.

Procedure:

1. Initialize the priority queue with all vertices, setting the key of the start vertex to 0 and others to infinity.

2. Repeatedly extract the vertex with the minimum key from the priority queue.

3. For each adjacent vertex, if it is still in the priority queue and the edge weight is less than the current key, update the key and set the parent of the vertex to the current vertex.

4. Continue the process until all vertices are included in the MST.

5. Use the parent array to extract edges forming the MST.

Algorithm:

Step 1. Define structures to represent edges and priority queue nodes.

Step 2. Create functions for priority queue operations, including `createNode`, `createPriorityQueue`, `swapNodes`, `heapify`, `extractMin`, `decreaseKey`, and `isInPriorityQueue`.

Step 3. Initialize `primMST` function to implement Prim's Algorithm:

- Step 3.1. Initialize `parent`, `key`, and `pos` arrays and priority queue.

- Step 3.2. Set the key of the start vertex (vertex 0) to 0.

- Step 3.3. While the priority queue is not empty:

  - Step 3.3.1. Extract the minimum node from the priority queue.

  - Step 3.3.2. For each adjacent vertex, if the vertex is in the priority queue and the edge weight is less than the current key, update the key and set the parent.

Step 4. Print the MST using `printMST`.

Step 5. In `main`, initialize vertices and edges, then call `primMST` to find the MST.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Structure to represent an edge in the graph
struct Edge {
int src, dest, weight;
};

// Structure to represent a node in the priority queue
struct PQNode {
int vertex, key;
};

// Structure to represent the priority queue
struct PriorityQueue {
int capacity;
int size;
struct PQNode* array;
};

// Function to create a new priority queue node
struct PQNode* createNode(int v, int key) {
struct PQNode* newNode = (struct PQNode*)malloc(sizeof(struct PQNode));
newNode->vertex = v;
newNode->key = key;
return newNode;
```

```
}

// Function to create a new priority queue
struct PriorityQueue* createPriorityQueue(int capacity) {
struct PriorityQueue* pq = (struct PriorityQueue*)malloc(sizeof(struct PriorityQueue));
pq->capacity = capacity;
pq->size = 0;
pq->array = (struct PQNode*)malloc(capacity * sizeof(struct PQNode));
return pq;
}

// Function to swap two nodes in the priority queue
void swapNodes(struct PQNode* a, struct PQNode* b) {
struct PQNode temp = *a;
*a = *b;
*b = temp;
}

// Function to heapify the priority queue
void heapify(struct PriorityQueue* pq, int idx, int* pos) {
int smallest, left, right;
smallest = idx;
left = 2 * idx + 1;
right = 2 * idx + 2;

if (left < pq->size && pq->array[left].key < pq->array[smallest].key)
smallest = left;
```

```
if (right < pq->size && pq->array[right].key < pq->array[smallest].key)
smallest = right;

if (smallest != idx) {
// Swap the nodes and update positions in the priority queue
pos[pq->array[smallest].vertex] = idx;
pos[pq->array[idx].vertex] = smallest;
swapNodes(&pq->array[smallest], &pq->array[idx]);

// Recursively heapify the affected sub-tree
heapify(pq, smallest, pos);
}
}

// Function to extract the minimum node from the priority queue
struct PQNode extractMin(struct PriorityQueue* pq, int* pos) {
if (pq->size == 0) {
struct PQNode emptyNode;
emptyNode.vertex = -1;
emptyNode.key = -1;
return emptyNode;
}

// Extract the root (minimum) node
struct PQNode root = pq->array[0];

// Replace the root node with the last node and heapify the priority queue
pq->array[0] = pq->array[pq->size - 1];
```

```
pos[root.vertex] = pq->size - 1;
pos[pq->array[0].vertex] = 0;
pq->size--;
heapify(pq, 0, pos);

return root;
}

// Function to decrease the key value of a given vertex in the priority queue
void decreaseKey(struct PriorityQueue* pq, int v, int key, int* pos) {
int i = pos[v];
pq->array[i].key = key;

// Fix the min heap property if it is violated
while (i > 0 && pq->array[(i - 1) / 2].key > pq->array[i].key) {
pos[pq->array[i].vertex] = (i - 1) / 2;
pos[pq->array[(i - 1) / 2].vertex] = i;
swapNodes(&pq->array[i], &pq->array[(i - 1) / 2]);

i = (i - 1) / 2;
}
}

// Function to check if a given vertex is in the priority queue
bool isInPriorityQueue(struct PriorityQueue* pq, int v) {
if (pq->size == 0)
return false;
```

```
return (pq->array[v].key < INT_MAX);
}

// Function to print the edges of the Minimum Spanning Tree (MST)
void printMST(struct Edge* result, int V) {
printf("Edges in the Minimum Spanning Tree:\n");
for (int i = 1; i < V; i++)
printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
}

// Function to find the Minimum Spanning Tree (MST) using Prim's algorithm with priority queue
void primMST(int V, int E, struct Edge* graph) {
int* parent = (int*)malloc(V * sizeof(int)); // To store the parent of each vertex in MST
int* key = (int*)malloc(V * sizeof(int));    // To store the key value of each vertex
int* pos = (int*)malloc(V * sizeof(int));    // To store the position of each vertex in the priority queue
struct PriorityQueue* pq = createPriorityQueue(V);

// Initialize values
for (int v = 1; v < V; v++) {
parent[v] = -1;
key[v] = INT_MAX;
pos[v] = v;
pq->array[v].vertex = v;
pq->array[v].key = key[v];
}

// Key value of the first vertex is 0, so it is picked as the first vertex
```

```
key[0] = 0;
pq->array[0].vertex = 0;
pq->array[0].key = key[0];

pq->size = V;

// Keep extracting the minimum key vertex until the priority queue becomes empty
while (pq->size > 0) {
struct PQNode u = extractMin(pq, pos);

// Include the extracted vertex in the MST
if (u.vertex != -1) {
// Traverse all adjacent vertices of the extracted vertex and update their key values
for (int i = 0; i < E; i++) {
int v = graph[i].dest;
if (graph[i].src == u.vertex && isInPriorityQueue(pq, v) && graph[i].weight < key[v]) {
parent[v] = u.vertex;
key[v] = graph[i].weight;
decreaseKey(pq, v, key[v], pos);
}
}
}
}

// Create an array to store the MST edges
struct Edge* result = (struct Edge*)malloc((V - 1) * sizeof(struct Edge));

// Copy the MST edges from parent array to the result array
```

```
for (int i = 1; i < V; i++) {
result[i - 1].src = parent[i];
result[i - 1].dest = i;
result[i - 1].weight = key[i];
}

// Print the MST edges
printMST(result, V);

// Free allocated memory
free(parent);
free(key);
free(pos);
free(pq->array);
free(pq);
free(result);
}

int main() {
// Example graph represented by edges and weights
int V = 5; // Number of vertices
int E = 7; // Number of edges

struct Edge* graph = (struct Edge*)malloc(E * sizeof(struct Edge));

// Edge 0-1
graph[0].src = 0;
graph[0].dest = 1;
```

```
graph[0].weight = 2;

// Edge 0-2
graph[1].src = 0;
graph[1].dest = 2;
graph[1].weight = 4;

// Edge 1-2
graph[2].src = 1;
graph[2].dest = 2;
graph[2].weight = 1;

// Edge 1-3
graph[3].src = 1;
graph[3].dest = 3;
graph[3].weight = 3;

// Edge 2-4
graph[4].src = 2;
graph[4].dest = 4;
graph[4].weight = 5;

// Edge 2-3
graph[5].src = 2;
graph[5].dest = 3;
graph[5].weight = 10;

// Edge 3-4
```

```
graph[6].src = 3;
graph[6].dest = 4;
graph[6].weight = 7;

// Call Prim's algorithm
primMST(V, E, graph);

// Free allocated memory
free(graph);

return 0;
}
```

Sample Input:

Vertices: 5

Edges:

0 - 1 with weight 2

0 - 2 with weight 4

1 - 2 with weight 1

1 - 3 with weight 3

2 - 4 with weight 5

2 - 3 with weight 10

3 - 4 with weight 7

Sample Output:

Edges in the Minimum Spanning Tree:

0 -- 1 == 2

1 -- 2 == 1

1 -- 3 == 3

2 -- 4 == 5

Result:

The program successfully finds the Minimum Spanning Tree (MST) of the given graph using Prim's Algorithm. The edges and their weights in the MST are displayed, demonstrating the shortest connections between all vertices.

Task No.	5.3	**Implement the Bellman-Ford algorithm to find the shortest path from a source vertex to all other vertices, even in graphs with negative-weight edges**
Date:	25-09-2025	

Aim:

To implement the Bellman-Ford algorithm to find the shortest paths from a source vertex to all other vertices in a weighted graph, detecting negative-weight cycles.

Procedure:

1. Create a graph with specified vertices and edges.

2. Initialize distances to all vertices as infinity, except for the source vertex, which is set to 0.

3. Relax each edge in the graph V-1 times to ensure the shortest paths.

4. Check for negative-weight cycles by relaxing all edges one more time. If a shorter path is found, a negative cycle exists.

5. Print the shortest distances from the source vertex or indicate if there is a negative-weight cycle.

Algorithm:

Step 1. Define structures to represent edges and the graph.

Step 2. Initialize `bellmanFord` function:

- Step 2.1. Set all distances to infinity, except for the source vertex, set to 0.

Step 3. Repeat for V-1 iterations:

- Step 3.1. For each edge, if a shorter path is found, update the distance of the destination vertex.

Step 4. Check for negative-weight cycles by iterating through all edges once more.

- Step 4.1. If a shorter path is found, print that a negative-weight cycle exists and exit.

Step 5. If no negative cycles, print shortest distances to each vertex.

Step 6. In `main`, initialize the graph vertices and edges, then call `bellmanFord` with the source vertex.

Program:

```
include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// Structure to represent an edge in the graph
struct Edge {
int src, dest, weight;
};

// Structure to represent the graph
struct Graph {
int V, E;
struct Edge* edges;
};

// Function to create a new graph with V vertices and E edges
struct Graph* createGraph(int V, int E) {
struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
graph->V = V;
graph->E = E;
graph->edges = (struct Edge*)malloc(E * sizeof(struct Edge));
return graph;
}

// Function to perform the Bellman-Ford algorithm
void bellmanFord(struct Graph* graph, int src) {
int V = graph->V;
```

```
int E = graph->E;
int* dist = (int*)malloc(V * sizeof(int));

// Initialize distances from the source to all vertices as INFINITY
for (int i = 0; i < V; i++) {
dist[i] = INT_MAX;
}

// Set the distance to the source vertex as 0
dist[src] = 0;

// Relax all edges V-1 times
for (int i = 1; i <= V - 1; i++) {
for (int j = 0; j < E; j++) {
int u = graph->edges[j].src;
int v = graph->edges[j].dest;
int weight = graph->edges[j].weight;

// Relax the edge (u, v)
if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
dist[v] = dist[u] + weight;
}
}
}

// Check for negative-weight cycles
for (int i = 0; i < E; i++) {
int u = graph->edges[i].src;
```

```
int v = graph->edges[i].dest;
int weight = graph->edges[i].weight;

// If a shorter path is found, there is a negative-weight cycle
if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
printf("Graph contains negative-weight cycle. No shortest paths exist.\n");
free(dist);
return;
}
}

// Print the shortest distances
printf("Shortest distances from source vertex %d:\n", src);
for (int i = 0; i < V; i++) {
printf("Vertex %d: ", i);
if (dist[i] == INT_MAX) {
printf("INFINITY\n");
} else {
printf("%d\n", dist[i]);
}
}

free(dist);
}

int main() {
// Example graph represented by edges and weights
int V = 5; // Number of vertices
```

```
int E = 8; // Number of edges

struct Graph* graph = createGraph(V, E);

// Edges of the graph
graph->edges[0].src = 0;
graph->edges[0].dest = 1;
graph->edges[0].weight = -1;

graph->edges[1].src = 0;
graph->edges[1].dest = 2;
graph->edges[1].weight = 4;

graph->edges[2].src = 1;
graph->edges[2].dest = 2;
graph->edges[2].weight = 3;

graph->edges[3].src = 1;
graph->edges[3].dest = 3;
graph->edges[3].weight = 2;

graph->edges[4].src = 1;
graph->edges[4].dest = 4;
graph->edges[4].weight = 2;

graph->edges[5].src = 3;
graph->edges[5].dest = 2;
graph->edges[5].weight = 5;
```

```
graph->edges[6].src = 3;
graph->edges[6].dest = 1;
graph->edges[6].weight = 1;

graph->edges[7].src = 4;
graph->edges[7].dest = 3;
graph->edges[7].weight = -3;

// Source vertex for the shortest paths
int sourceVertex = 0;

// Perform Bellman-Ford algorithm
bellmanFord(graph, sourceVertex);

// Free allocated memory
free(graph->edges);
free(graph);

return 0;
}
```

Sample Input:

Vertices: 5

Edges:

0 - 1 with weight -1

0 - 2 with weight 4

1 - 2 with weight 3

1 - 3 with weight 2

1 - 4 with weight 2

3 - 2 with weight 5

3 - 1 with weight 1

4 - 3 with weight -3

Source Vertex: 0

Sample Output:

Shortest distances from source vertex 0:

Vertex 0: 0

Vertex 1: -1

Vertex 2: 2

Vertex 3: -2

Vertex 4: 1

Result:

The program successfully finds the shortest paths from the source vertex to all other vertices using the Bellman-Ford algorithm. It detects and reports any negative-weight cycles if present.

Task No.	6.1	**Implement the Ford-Fulkerson algorithm to find the maximum flow in a network. Test it on different flow networks and explore different augmenting path strategies (BFS & DFS)**
Date:	09-10-2025	

Aim:

To implement the Ford-Fulkerson algorithm in C to find the maximum flow in a flow network.

Procedure:

1. **Define Constants and Initialize Graph**:
    - Define constants, such as the number of vertices, V.
    - Create a 2D array to represent the flow network.
2. **Ford-Fulkerson Algorithm**:
    - Define the fordFulkerson() function to calculate the maximum flow from a source vertex to a sink vertex.
    - Create a residual graph that stores the capacities after accounting for flow.
    - While there is an augmenting path (found using BFS), find the minimum capacity along that path.
    - Subtract the path flow from the residual capacities of the edges in the augmenting path and add it to the reverse edges.
    - Accumulate the path flow into the overall maximum flow.
3. **Breadth-First Search (BFS)**:
    - Define the bfs() function to find augmenting paths in the residual graph.
    - Mark vertices as visited, and if the sink vertex is reached, an augmenting path is found.
4. **Utility Function**:
    - Create a min() function to find the minimum residual capacity along an augmenting path.
5. **Execution**:
    - Define the graph array with initial capacities for each edge in the network.

- Call fordFulkerson() with the graph, source, and sink vertices to find the maximum flow.
- Print the maximum flow result.

Algorithm:

Step 1: Initialize Residual Graph

- Create a residual graph rGraph as a copy of the original graph, where each edge's value represents the available remaining capacity.

Step 2: Initialize Maximum Flow

- Set the maximum flow (max_flow) to 0.

Step 3: Find an Augmenting Path

- Use Breadth-First Search (BFS) to locate an augmenting path from the source (s) to the sink (t) in the residual graph.

Step 4: Determine Path Flow

- Identify the minimum residual capacity (path_flow) along the augmenting path. This value represents the maximum flow possible along that path.

Step 5: Update Residual Capacities

- For each edge (u, v) in the augmenting path:
  - Subtract path_flow from rGraph[u][v] (forward edge).
  - Add path_flow to rGraph[v][u] (backward edge).

Step 6: Update Maximum Flow and Repeat

- Add path_flow to the total max_flow.
- Repeat steps 3–6 until no further augmenting paths can be found in the residual graph.

Program:

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
#define V 6
// Function to find the maximum flow in a network using the Ford-Fulkerson algorithm
int fordFulkerson(int graph[V][V], int source, int sink) {
int residualGraph[V][V];
int parent[V];
int maxFlow = 0;

// Initialize the residual graph with the original capacities
for (int i = 0; i < V; i++) {
for (int j = 0; j < V; j++) {
residualGraph[i][j] = graph[i][j];
}
}

// Augment the flow while there is an augmenting path in the residual graph
while (bfs(residualGraph, source, sink, parent)) {
int pathFlow = INT_MAX;

// Find the minimum residual capacity along the augmenting path
for (int v = sink; v != source; v = parent[v]) {
int u = parent[v];
pathFlow = min(pathFlow, residualGraph[u][v]);
}
```

```
// Update the residual capacities and reverse edges along the augmenting path
for (int v = sink; v != source; v = parent[v]) {
int u = parent[v];
residualGraph[u][v] -= pathFlow;
residualGraph[v][u] += pathFlow;
}

// Add the path flow to the overall maximum flow
maxFlow += pathFlow;
}

// Return the maximum flow
return maxFlow;
}

// Function to perform a breadth-first search (BFS) to find an augmenting path
bool bfs(int residualGraph[V][V], int source, int sink, int parent[]) {
bool visited[V];
memset(visited, false, sizeof(visited));

// Create a queue for BFS and enqueue the source vertex
Queue queue;
queue.enqueue(source);
visited[source] = true;
parent[source] = -1;

// Standard BFS loop
while (!queue.isEmpty()) {
```

```
int u = queue.dequeue();

for (int v = 0; v < V; v++) {
if (!visited[v] && residualGraph[u][v] > 0) {
queue.enqueue(v);
parent[v] = u;
visited[v] = true;
}
}
}

// If we reached the sink vertex in BFS, then there is an augmenting path
return visited[sink];
}

// Function to find the minimum of two integers
int min(int a, int b) {
return (a < b) ? a : b;
}

// Driver code
int main() {
// Example flow network
int graph[V][V] = {
{0, 16, 13, 0, 0, 0},
{0, 0, 10, 12, 0, 0},
{0, 4, 0, 0, 14, 0},
{0, 0, 9, 0, 0, 20},
```

```
{0, 0, 0, 7, 0, 4},
{0, 0, 0, 0, 0, 0}
};

int source = 0; // Source vertex
int sink = 5; // Sink vertex

// Find the maximum flow in the network
int maxFlow = fordFulkerson(graph, source, sink);

printf("Maximum flow in the network: %d\n", maxFlow);

return 0;
}
```

Sample Input:

plaintext

Copy code

Source: 0

Sink: 5

Graph:

{

{0, 16, 13, 0, 0, 0},

{0, 0, 10, 12, 0, 0},

{0, 4, 0, 0, 14, 0},

{0, 0, 9, 0, 0, 20},

{0, 0, 0, 7, 0, 4},

{0, 0, 0, 0, 0, 0}

}

Sample Output:

plaintext

Copy code

Maximum flow in the network: 23

Result:

The Ford-Fulkerson algorithm successfully calculates the maximum flow in the given network. The implementation, using BFS to find augmenting paths and a residual graph to track available capacities, computes the maximum flow from the source to the sink, which is 23 in this example.

Task No.	6.2	**Implement the Edmonds-Karp algorithm, a specific implementation of Ford Fulkerson using BFS for finding augmenting paths.**
Date:	16-10-2025	

Aim:

To implement the Edmonds-Karp algorithm, an optimized version of the Ford-Fulkerson algorithm, to find the maximum possible flow in a given flow network.

Procedure:

1. **Input the Graph:** Define the directed graph as an adjacency matrix with edge capacities, and specify the source and sink vertices.
2. **Initialize Residual Graph:** Copy the original graph to a residual graph that keeps track of remaining capacities.
3. **Find Augmenting Paths Using BFS:** While there exists an augmenting path from the source to the sink, perform a BFS to locate it.
4. **Update Flow:** Calculate the path flow (minimum capacity along the path) and update the residual graph by adjusting the capacities.
5. **Calculate Maximum Flow**: Accumulate the path flows until no more augmenting paths exist, yielding the maximum flow from source to sink.

Algorithm:

Step 1: Copy the original graph into a residual graph (rGraph) to track remaining capacities for each edge.

Step 2: Initialize maxFlow to zero, which will store the cumulative maximum flow from source to sink.

Step 3: Use BFS to find an augmenting path from the source to the sink in the residual graph.

Step 4: Calculate the minimum residual capacity (pathFlow) along the augmenting path, which is the smallest capacity among the edges in the path.

Step 5: Adjust the capacities along the augmenting path:

- Subtract pathFlow from each forward edge.
- Add pathFlow to each reverse edge.

Step 6: Add pathFlow to the overall maxFlow. And Repeat steps 3–6 until no more augmenting paths can be found.

Program:

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
#define V 6
// Function to find the minimum of two values
int min(int a, int b) {
return (a < b) ? a : b;
}
// Function to find if there is a path from source to sink
bool bfs(int rGraph[V][V], int source, int sink, int parent[]) {
bool visited[V];
memset(visited, 0, sizeof(visited));

visited[source] = true;
parent[source] = -1;

// Create a queue for BFS
int queue[V];
int front = 0, rear = 0;
queue[rear++] = source;

while (front < rear) {
int u = queue[front++];
for (int v = 0; v < V; v++) {
if (!visited[v] && rGraph[u][v] > 0) {
queue[rear++] = v;
parent[v] = u;
```

```
visited[v] = true;
}
}
}

return (visited[sink] == true);
}

// Function to implement the Edmonds-Karp algorithm
int edmondsKarp(int graph[V][V], int source, int sink) {
int rGraph[V][V];
for (int u = 0; u < V; u++) {
for (int v = 0; v < V; v++) {
rGraph[u][v] = graph[u][v];
}
}

int parent[V];
int maxFlow = 0;
while (bfs(rGraph, source, sink, parent)) {
int pathFlow = INT_MAX;
for (int v = sink; v != source; v = parent[v]) {
int u = parent[v];
pathFlow = min(pathFlow, rGraph[u][v]);
}
for (int v = sink; v != source; v = parent[v]) {
int u = parent[v];
rGraph[u][v] -= pathFlow;
```

```
rGraph[v][u] += pathFlow;
}

maxFlow += pathFlow;
}
return maxFlow;
}

int main() {
int graph[V][V] = {
{0, 16, 13, 0, 0, 0},
{0, 0, 10, 12, 0, 0},
{0, 4, 0, 0, 14, 0},
{0, 0, 9, 0, 0, 20},
{0, 0, 0, 7, 0, 4},
{0, 0, 0, 0, 0, 0}
};

int source = 0;
int sink = 5;
int maxFlow = edmondsKarp(graph, source, sink);
printf("The maximum possible flow is: %d\n", maxFlow);

return 0;
}
```

Sample Input:

The graph matrix and the values for the source and sink nodes are given below:

```
int graph[V][V] = {
{0, 16, 13, 0, 0, 0},
{0, 0, 10, 12, 0, 0},
{0, 4, 0, 0, 14, 0},
{0, 0, 9, 0, 0, 20},
{0, 0, 0, 7, 0, 4},
{0, 0, 0, 0, 0, 0}
};
int source = 0;
int sink = 5;
```

Sample Output:

The maximum possible flow is: 23

Result:

The implemented Edmonds-Karp algorithm successfully finds the maximum flow in the network, which is 23 for the given input graph.

Task No.	6.3	**Implement max-flow min-cut algorithm to image segmentation problems: to partition an image into different segments while minimizing the cut.**
Date:	23-10-2025	

Aim:

To implement the Ford-Fulkerson algorithm to determine the maximum flow and minimum cut in a flow network, identifying the maximum flow from a source to a sink and the edges that form the minimum cut.

Procedure:

1. **Input the Graph:** Represent the directed graph as an adjacency matrix with each edge's capacity, and define the source and sink nodes.
2. **Initialize Residual Graph:** Create a residual graph to keep track of remaining capacities in the network.
3. **Find Augmenting Paths Using BFS:** Use BFS to identify augmenting paths in the residual graph. Continue until no more paths exist.
4. **Calculate Path Flow**: Find the path flow (minimum capacity) for each augmenting path and update the residual graph accordingly.
5. **Determine Maximum Flow:** Accumulate path flows to calculate the maximum flow from source to sink.
6. **Identify Minimum Cut:** After finding the maximum flow, use Depth-First Search (DFS) to identify reachable nodes from the source. The edges crossing from reachable to non-reachable vertices in the original graph represent the minimum cut.

Algorithm:

Step 1: Initialize the residual graph by copying the original graph.

Step 2: Set the maximum flow max_flow to zero.

Step 3: Use BFS to find an augmenting path from the source to the sink in the residual graph.

Step 4: Calculate the path flow as the minimum capacity along the augmenting path and update the residual graph by subtracting path_flow in the forward direction and adding it in the reverse direction.

Step 5: Add path_flow to max_flow.

Step 6: Perform DFS to identify all reachable vertices from the source and identify edges from reachable to non-reachable vertices in the original graph to find the minimum cut.

Program:

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#include <string.h>
#define V 6
bool bfs(int rGraph[V][V], int s, int t, int parent[]) {
bool visited[V];
memset(visited, 0, sizeof(visited));

int queue[V];
int front = 0, rear = 0;
queue[rear++] = s;
visited[s] = true;
parent[s] = -1;

while (front < rear) {
int u = queue[front++];

for (int v = 0; v < V; v++) {
if (!visited[v] && rGraph[u][v] > 0) {
if (v == t) {
parent[v] = u;
return true;
}
queue[rear++] = v;
parent[v] = u;
```

```
visited[v] = true;
}
}
}

return false;
}

int fordFulkerson(int graph[V][V], int s, int t) {
int u, v;

int rGraph[V][V];
for (u = 0; u < V; u++)
for (v = 0; v < V; v++)
rGraph[u][v] = graph[u][v];

int parent[V];
int max_flow = 0;

while (bfs(rGraph, s, t, parent)) {
int path_flow = INT_MAX;
for (v = t; v != s; v = parent[v]) {
u = parent[v];
path_flow = (path_flow < rGraph[u][v]) ? path_flow : rGraph[u][v];
}

for (v = t; v != s; v = parent[v]) {
u = parent[v];
```

```
rGraph[u][v] -= path_flow;
rGraph[v][u] += path_flow;
}

max_flow += path_flow;
}
return max_flow;
}

void minCut(int graph[V][V], int s, int t) {
int u, v;

int rGraph[V][V];
for (u = 0; u < V; u++)
for (v = 0; v < V; v++)
rGraph[u][v] = graph[u][v];

int parent[V];
int max_flow = fordFulkerson(graph, s, t);

printf("Max Flow: %d\n", max_flow);
printf("Min Cut:\n");

bool visited[V];
memset(visited, false, sizeof(visited));

// Mark reachable vertices from source in rGraph
dfs(rGraph, s, visited);
```

```
// Print all edges that are from a reachable vertex to an
// unreachable vertex in the original graph
for (u = 0; u < V; u++)
for (v = 0; v < V; v++)
if (visited[u] && !visited[v] && graph[u][v])
printf("%d - %d\n", u, v);
}
// DFS to find all reachable vertices
void dfs(int rGraph[V][V], int s, bool visited[]) {
visited[s] = true;
for (int i = 0; i < V; i++)
if (rGraph[s][i] && !visited[i])
dfs(rGraph, i, visited);
}
int main() {
int graph[V][V] = {{0, 16, 13, 0, 0, 0},
{0, 0, 10, 12, 0, 0},
{0, 4, 0, 0, 14, 0},
{0, 0, 9, 0, 0, 20},
{0, 0, 0, 7, 0, 4},
{0, 0, 0, 0, 0, 0}};

int s = 0, t = 5; // Source and sink vertices
minCut(graph, s, t);
return 0;
}
```

Sample Input:

The graph matrix and source and sink nodes are defined as:

```
int graph[V][V] = {
{0, 16, 13, 0, 0, 0},
{0, 0, 10, 12, 0, 0},
{0, 4, 0, 0, 14, 0},
{0, 0, 9, 0, 0, 20},
{0, 0, 0, 7, 0, 4},
{0, 0, 0, 0, 0, 0}
};
int s = 0; // Source vertex
int t = 5; // Sink vertex
```

Sample Output:

Max Flow: 23

Min Cut:

0 - 1

0 - 2

3 - 5

4 – 5

Result:

The implementation successfully calculates the maximum flow in the network (23 units) and identifies the edges forming the minimum cut. The edges in the minimum cut effectively

separate the source from the sink, ensuring no additional flow can cross this boundary without exceeding the network's capacity.

Task No.	7.1	**Implement an algorithm to generate strong pseudoprimes based on the Miller-Rabin primality test.**
Date:	30-10-2025	

Aim:

To determine if a given number is a strong pseudoprime using the Miller-Rabin primality test.

Procedure:

1. Prompt the user to input a number nnn and the number of iterations kkk.
2. Calculate values for ddd and rrr such that n=2r·d+1n = 2^r \cdot d + 1n=2r·d+1.
3. Select a random number aaa as a potential witness.
4. Perform the Miller-Rabin primality test for kkk iterations. If any iteration confirms that nnn is composite, conclude that nnn is not a strong pseudoprime.
5. If none of the iterations find nnn to be composite, conclude that nnn is likely a strong pseudoprime.

Algorithm:

Step 1: Initialize d as n−1n - 1n−1 and r as 0.

Step 2: Divide d by 2 until it becomes odd, incrementing r each time.

Step 3: Repeat k times to test potential primality.

Step 4: Randomly select an integer a between 2 and n−2n-2n−2.

Step 5: Compute x=admod nx = a^d \mod nx=admodn. If x=1x = 1x=1 or x=n−1x = n-1x=n−1, continue to the next iteration.

Step 6: Repeat r-1 times to check intermediate values of xxx. If x=n−1x = n-1x=n−1, continue to the next iteration; if x=1x = 1x=1, conclude that nnn is composite.

Step 7: If no iterations find nnn to be composite, conclude that nnn is likely a strong pseudoprime.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>

// Function to calculate (a^b) % c
long long int power(long long int a, long long int b, long long int c)
{
long long int result = 1;
a = a % c;

while (b > 0)
{
if (b & 1)
result = (result * a) % c;

b = b >> 1;
a = (a * a) % c;
}

return result;
}

// Function to check if a number is a strong pseudoprime
bool isStrongPseudoprime(long long int n, int k)
```

```
{
// Base cases
if (n <= 1 || n == 4)
return false;
if (n <= 3)
return true;

// Find r and d such that n = 2^r * d + 1
long long int d = n - 1;
int r = 0;
while (d % 2 == 0)
{
d /= 2;
r++;
}

// Repeat k times
for (int i = 0; i < k; i++)
{
// Generate a random number 'a' between 2 and n-2
long long int a = 2 + rand() % (n - 4);

// Compute a^d % n
long long int x = power(a, d, n);

// If x is 1 or n-1, continue to next iteration
if (x == 1 || x == n - 1)
continue;
```

```
// Repeat r-1 times
for (int j = 0; j < r - 1; j++)
{
// Compute (x^2) % n
x = (x * x) % n;

// If x is n-1, continue to next iteration
if (x == n - 1)
break;

// If x is 1, n is composite
if (x == 1)
return false;
}

// If we reach here, n is composite
return false;
}

// If we reach here, n is probably prime
return true;
}

int main()
{
long long int n;
int k;
```

```
printf("Enter a number: ");
scanf("%lld", &n);

printf("Enter the number of iterations (k): ");
scanf("%d", &k);

if (isStrongPseudoprime(n, k))
printf("%lld is a strong pseudoprime.\n", n);
else
printf("%lld is not a strong pseudoprime.\n", n);

return 0;
}
```

Sample Input:

Enter a number: 561

Enter the number of iterations (k): 5

Sample Output:

561 is not a strong pseudoprime.

Result:

The program correctly determines whether the input number is a strong pseudoprime based on the Miller-Rabin test.

Task No.	7.2	Implement Pollard's Rho algorithm for integer factorization
Date:	30-10-2025	

Aim:

To factorize a given number using Pollard's Rho algorithm for integer factorization.

Procedure:

1. Prompt the user to input a number nnn for factorization.
2. Initialize random seed and call the factorize function.
3. Inside the factorize function, repeatedly apply Pollard's Rho algorithm to find factors of nnn until nnn is reduced to 1.
4. Print the factors found during the factorization process.

Algorithm:

Step 1: Define a recursive function to calculate the GCD of two numbers.

Step 2: In the pollardsRho function, initialize xxx and yyy with random values and set ddd to 1.

Step 3: Define the polynomial function f(x)=(x2+1)mod nf(x) = (x^2 + 1) \mod nf(x)=(x2+1)modn.

Step 4: In a loop, update xxx and yyy using the polynomial function and calculate d=gcd(∣x−y∣,n)d = \text{gcd}(|x - y|, n)d=gcd(∣x−y∣,n).

Step 5: If ddd is not equal to 1, return ddd as a factor of nnn.

Step 6: In the factorize function, call pollardsRho repeatedly until nnn is reduced to 1, printing each factor found.

Program:

```
#include <stdio.h>
#include <stdlib.h>

// Function to calculate the greatest common divisor (GCD) of two numbers
long long gcd(long long a, long long b) {
if (b == 0) {
return a;
} else {
return gcd(b, a % b);
}
}

// Function to perform Pollard's Rho algorithm for integer factorization
long long pollardsRho(long long n) {
long long x = rand() % (n - 2) + 2;
long long y = x;
long long d = 1;
// Define the polynomial function f(x) = (x^2 + 1) % n
long long f(long long x) {
return (x * x + 1) % n;
}
// Main loop of Pollard's Rho algorithm
while (d == 1) {
x = f(x);
y = f(f(y));
d = gcd(abs(x - y), n);
}
```

```
return d;
}
// Function to perform the factorization of a number
void factorize(long long n) {
if (n <= 1) {
printf("Cannot factorize numbers less than or equal to 1.\n");
return;
}
printf("Factors of %lld are: ", n);
while (n > 1) {
long long factor = pollardsRho(n);
printf("%lld ", factor);

// Divide n by the found factor
n /= factor;
}
printf("\n");
}
int main() {
// Seed the random number generator
srand(42); // You can use a different seed or a dynamic seed
// Test the factorization function
long long number;
printf("Enter a number to factorize: ");
scanf("%lld", &number);
factorize(number);
return 0;
}
```

Sample Input:

Enter a number to factorize: 8051

Sample Output:

Factors of 8051 are: 97 83

Result:

The program successfully factors the input number using Pollard's Rho algorithm and prints the factors.

Task No.	8.1	**Implement an algorithm generates random passwords using pseudorandom numbers. Users can specify the desired password length and character set**
Date:	06-11-2025	

Aim:

To generate a random password of a specified length using a defined character set.

Procedure:

1. Prompt the user to enter the desired password length.
2. Define a character set that includes lowercase letters, uppercase letters, digits, and special characters.
3. Call the generatePassword function to create a random password using the specified length and character set.
4. Print the generated password.

Algorithm:

Step 1: Read the desired password length from the user.

Step 2: Define the character set for the password.

Step 3: In the generatePassword function, calculate the size of the character set.

Step 4: Seed the random number generator using the current time.

Step 5: Generate a password by iterating for the specified length and selecting random characters from the character set.

Step 6: Print the generated password.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Function to generate a random password
void generatePassword(int length, const char *charset) {
int charsetSize = 0;
while (charset[charsetSize] != '\0') {
charsetSize++;
}

// Seed the random number generator with the current time
srand((unsigned int)time(NULL));

printf("Generated Password: ");
for (int i = 0; i < length; i++) {
// Generate a random index within the character set
int randomIndex = rand() % charsetSize;
// Print the character at the random index
printf("%c", charset[randomIndex]);
}
printf("\n");
}

int main() {
int passwordLength;
printf("Enter the desired password length: ");
```

```
scanf("%d", &passwordLength);

// Define the character set for the password
const char *characterSet =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#$%^
&*()-=_+";

// Generate and print the random password
generatePassword(passwordLength, characterSet);

return 0;
}
```

Sample Input:

Enter the desired password length: 12

Sample Output:

Generated Password: G4$hD6kz@lA1

Result:

The program successfully generates and displays a random password based on the user-defined length and character set.

Task No.	8.2	**Implement a basic plane sweep algorithm. Use it to solve a simple geometric problem, such as finding intersections among a set of line segments.**
Date:	06-11-2025	

Aim:

To find and display intersections between vertical line segments in a 2D plane.

Procedure:

1. Define line segments with their x-coordinates and y-coordinate ranges.
2. Sort the line segments based on their x-coordinates.
3. Check each pair of line segments to determine if they intersect.
4. Print the intersecting line segments.

Algorithm:

Step 1: Define a structure for the line segments containing x-coordinates and y-coordinate ranges.

Step 2: Implement a comparison function to sort line segments by their x-coordinates.

Step 3: Sort the array of line segments using qsort() and the comparison function.

Step 4: Iterate over each pair of line segments to check for intersections.

Step 5: For each intersecting pair, print the coordinates of the intersecting line segments.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct {
int x, y1, y2; // x-coordinate and y-coordinates of line segment endpoints
} LineSegment;

int compare(const void* a, const void* b) {
return ((LineSegment*)a)->x - ((LineSegment*)b)->x;
}

bool doIntersect(LineSegment l1, LineSegment l2) {
return (l1.x < l2.x && l1.y1 > l2.y1 && l1.y2 < l2.y2);
}

void findIntersections(LineSegment segments[], int n) {
qsort(segments, n, sizeof(LineSegment), compare);

printf("Intersections:\n");
for (int i = 0; i < n - 1; ++i) {
for (int j = i + 1; j < n; ++j) {
if (doIntersect(segments[i], segments[j])) {
printf("Line segments at (%d, %d)-(%d, %d) and (%d, %d)-(%d, %d) intersect.\n",
segments[i].x, segments[i].y1, segments[i].x, segments[i].y2,
segments[j].x, segments[j].y1, segments[j].x, segments[j].y2);
}
```

```
}
}
}

int main() {
LineSegment segments[] = {
{1, 3, 7}, {2, 4, 9}, {3, 2, 5}, {5, 1, 6}, {7, 3, 8}
};
int n = sizeof(segments) / sizeof(segments[0]);
findIntersections(segments, n);
return 0;
}
```

Sample Input:

LineSegment segments[] = {

{1, 3, 7}, {2, 4, 9}, {3, 2, 5}, {5, 1, 6}, {7, 3, 8}

};

int n = 5;

Sample Output:

Intersections:

Line segments at (1, 3)-(1, 7) and (2, 4)-(2, 9) intersect.

Line segments at (2, 4)-(2, 9) and (3, 2)-(3, 5) intersect.

Line segments at (3, 2)-(3, 5) and (5, 1)-(5, 6) intersect.

Line segments at (5, 1)-(5, 6) and (7, 3)-(7, 8) intersect.

Result:

The program successfully identifies and displays intersections between the specified vertical line segments.

Task No.	9.1	**Implement a randomized algorithm to select a random subset of elements from a given array with equal probability**
Date:	13-11-2025	

Aim:

To generate a random subset of k elements from an array of n integers using the Fisher-Yates shuffle algorithm.

Procedure:

1. Define an array and the desired subset size, k.
2. Use the Fisher-Yates shuffle algorithm to randomly permute the elements in the array.
3. Select the first k elements from the shuffled array as the random subset.
4. Print the generated subset.

Algorithm:

Step 1: Define an array of integers and specify the subset size, k.

Step 2: Check if k is greater than the array size n. If so, display an error message and return.

Step 3: Use the Fisher-Yates shuffle to randomly swap elements and permute the array.

Step 4: Print the first k elements from the shuffled array as the random subset.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Function to swap two elements in an array
void swap(int* a, int* b) {
int temp = *a;
*a = *b;
*b = temp;
}

// Function to generate a random subset of k elements from the array
void randomSubset(int arr[], int n, int k) {
if (k > n) {
printf("Error: Subset size cannot be greater than array size.\n");
return;
}
// Use Fisher-Yates Shuffle to randomly permute the array
for (int i = n - 1; i > 0; --i) {
int j = rand() % (i + 1);
swap(&arr[i], &arr[j]);
}
// Print the first k elements as the random subset
printf("Random Subset of %d elements: ", k);
for (int i = 0; i < k; ++i) {
printf("%d ", arr[i]);
}
```

```
printf("\n");
}
int main() {
// Seed the random number generator with the current time
srand(time(NULL));
// Example array
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int n = sizeof(arr) / sizeof(arr[0]);
// Size of the random subset to select
int k = 5;
// Call the randomSubset function
randomSubset(arr, n, k);
return 0;
}
```

Sample Input:

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int k = 5;

Sample Output:

Random Subset of 5 elements: 3 7 1 9 6

Result:

The program successfully generates and displays a random subset of k elements from the array.

Task No.	9.2	**Implement a linear programming model to optimize the construction of a binary search tree with specific access frequencies for the elements. Solve it to find the optimal tree structure.**
Date:	13-11-2025	

Aim:

To implement a linear programming model to construct an optimized binary search tree (BST) using specific access frequencies for the elements and find the optimal tree structure.

Procedure:

1. Collect the access frequencies for all elements that need to be stored in the BST.
2. Formulate a cost function representing the total search cost, considering the access frequencies.
3. Set up constraints for the tree structure, ensuring it remains a BST.
4. Implement the linear programming model to minimize the total search cost.
5. Use an appropriate linear programming solver or algorithm to find the solution.
6. Construct the BST based on the solution obtained.

Algorithm:

Step 1: Input the number of elements and their access frequencies.

Step 2: Initialize a cost matrix to store the cumulative costs.

Step 3: Set up a matrix to track root nodes for subtrees.

Step 4: Calculate the cost for subtrees of increasing lengths.

Step 5: For each length, compute the minimum cost using a nested loop.

Step 6: Store the calculated costs and roots in the matrices.

Step 7: Construct the BST using the root matrix.

Step 8: Display the optimal BST structure and total search cost.

Program:

```
#include <stdio.h>
#include <limits.h>

#define MAX 100

// Function prototypes
void optimalBST(int keys[], int freq[], int n);
int sum(int freq[], int i, int j);
void printStructure(int root[MAX][MAX], int i, int j, int keys[], int level);

// Function to calculate optimal BST
void optimalBST(int keys[], int freq[], int n) {
int cost[MAX][MAX];
int root[MAX][MAX];

// Initialize the cost and root matrices
for (int i = 0; i < n; i++) {
cost[i][i] = freq[i];
root[i][i] = i;
}

// Calculate cost for trees of increasing length
for (int length = 2; length <= n; length++) {
for (int i = 0; i <= n - length; i++) {
int j = i + length - 1;
cost[i][j] = INT_MAX;
```

```
// Try making each key in the range the root
for (int r = i; r <= j; r++) {
int c = (r > i ? cost[i][r - 1] : 0) +
(r < j ? cost[r + 1][j] : 0) +
sum(freq, i, j);

if (c < cost[i][j]) {
cost[i][j] = c;
root[i][j] = r;
}
}
}
}

printf("Optimal cost: %d\n", cost[0][n - 1]);
printf("Optimal BST structure:\n");
printStructure(root, 0, n - 1, keys, 0);
}

// Function to calculate sum of frequencies
int sum(int freq[], int i, int j) {
int s = 0;
for (int k = i; k <= j; k++)
s += freq[k];
return s;
}

// Function to print the structure of the tree
```

```
void printStructure(int root[MAX][MAX], int i, int j, int keys[], int level) {
if (i > j)
return;

// Print indentation for tree structure visualization
for (int l = 0; l < level; l++)
printf("  ");
printf("Root: %d\n", keys[root[i][j]]);

// Recursively print left subtree
for (int l = 0; l < level; l++)
printf("  ");
printf("Left subtree of %d:\n", keys[root[i][j]]);
printStructure(root, i, root[i][j] - 1, keys, level + 1);
// Recursively print right subtree
for (int l = 0; l < level; l++)
printf("  ");
printf("Right subtree of %d:\n", keys[root[i][j]]);
printStructure(root, root[i][j] + 1, j, keys, level + 1);
}

int main() {
int keys[] = {10, 20, 30, 40};
int freq[] = {4, 2, 6, 3};
int n = sizeof(keys) / sizeof(keys[0]);
optimalBST(keys, freq, n);
return 0;
}
```

Sample Input:

Keys: {10, 20, 30, 40}

Frequencies: {4, 2, 6, 3}

Sample Output:

Optimal cost: 26

Optimal BST structure:

Root: 30

Left subtree of 30:

Root: 10

Right subtree of 30:

Root: 40

Left subtree of 40:

Root: 20

Result:

The linear programming model successfully constructed an optimized BST with a minimal search cost.