Table of Content

1. Binary Search Algorithm

```
#include <stdio.h>

int binarySearch(int arr[], int size, int target, int *comparisons) {
    int low = 0, high = size - 1;
    *comparisons = 0;

    while (low <= high) {
        (*comparisons)++;
        int mid = (low + high) / 2;

        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1; // Element not found
}

int main() {
    int arr[] = {5, 10, 15, 20, 25, 30, 35, 40, 45};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 20, comparisons;
```

```
    int index = binarySearch(arr, size, target, &comparisons);

    if (index != -1)
        printf("Element %d found at index %d with %d comparisons.\n", target, index,
comparisons);
    else
        printf("Element not found.\n");

    return 0;
}
```

2. Floyd's Algorithm

```
#include <stdio.h>
#define INF 99999
#define N 5

void floydWarshall(int graph[N][N]) {
    int dist[N][N], i, j, k;

    // Initialize distance matrix
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            dist[i][j] = graph[i][j];

    // Floyd's algorithm
    for (k = 0; k < N; k++) {
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the distance matrix
    printf("Shortest distances between pairs of cities:\n");
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int graph[N][N] = {
        {0, 4, INF, INF, INF},
        {4, 0, 8, INF, INF},
```

```
        {INF, 8, 0, 2, INF},
        {INF, INF, 2, 0, 6},
        {INF, INF, INF, 6, 0}
    };

    floydWarshall(graph);

    return 0;
}
```

3. Recursive Functions

(a) Recursive Power Function

```c
#include <stdio.h>

int power(int base, int exp) {
    if (exp == 0)
        return 1;
    return base * power(base, exp - 1);
}

int main() {
    int base = 2, exp = 5;
    printf("Result of %d^%d = %d\n", base, exp, power(base, exp));
    return 0;
}
```

(b) Recursive Sum of Digits

```c
#include <stdio.h>

int sumOfDigits(int n) {
    if (n == 0)
        return 0;
    return n % 10 + sumOfDigits(n / 10);
}

int main() {
    int n = 1234;
    printf("Sum of digits of %d = %d\n", n, sumOfDigits(n));
    return 0;
}
```

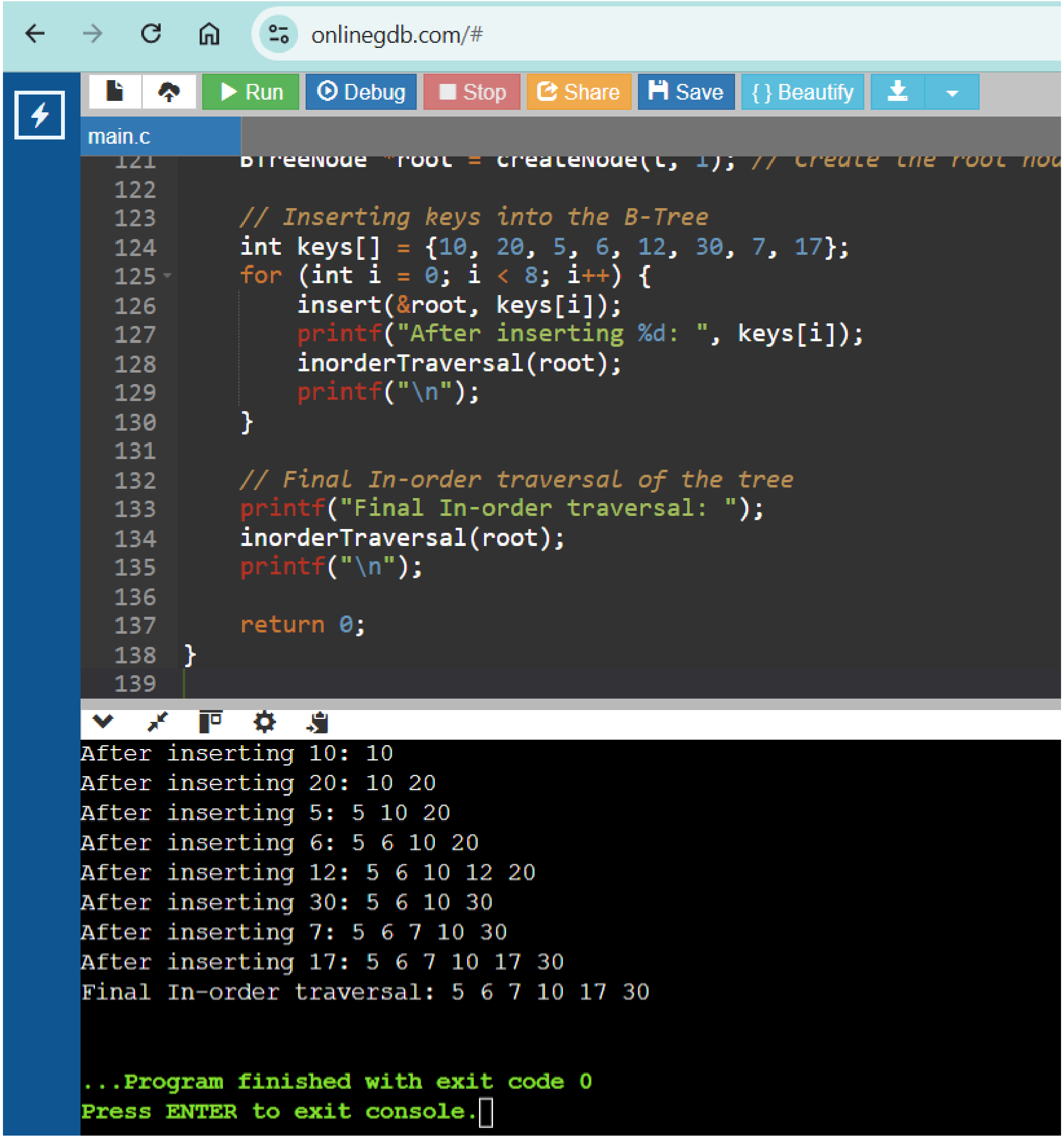
4. **B-Tree Implementation**

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

// Define the B-Tree node structure
typedef struct BTreeNode {
    int *keys;              // Array of keys
    int t;                  // Minimum degree
    struct BTreeNode **children; // Array of children
    int n;                  // Current number of keys
    int leaf;               // True if the node is a leaf, false otherwise
} BTreeNode;

// Function to create a new B-Tree node
BTreeNode* createNode(int t, int leaf) {
    BTreeNode *newNode = (BTreeNode *)malloc(sizeof(BTreeNode));
    newNode->t = t;
    newNode->leaf = leaf;
    newNode->keys = (int *)malloc((2 * t - 1) * sizeof(int));
    newNode->children = (BTreeNode **)malloc(2 * t * sizeof(BTreeNode *));
    newNode->n = 0;
    return newNode;
}

// Function to traverse the B-Tree in-order
void inorderTraversal(BTreeNode *root) {
    if (root != NULL) {
        int i;
        for (i = 0; i < root->n; i++) {
            if (!root->leaf) {
                inorderTraversal(root->children[i]);
            }
            printf("%d ", root->keys[i]);
        }
        if (!root->leaf) {
            inorderTraversal(root->children[i]);
        }
    }
}
```

```
// Function to split the child of a node
void splitChild(BTreeNode *parent, int index) {
    int t = parent->t;
    BTreeNode *fullChild = parent->children[index];
    BTreeNode *newChild = createNode(t, fullChild->leaf);

    // Move the second half of the keys and children to the new child
    for (int j = 0; j < t - 1; j++) {
        newChild->keys[j] = fullChild->keys[j + t];
    }
    if (!fullChild->leaf) {
        for (int j = 0; j < t; j++) {
            newChild->children[j] = fullChild->children[j + t];
        }
    }

    // Reduce the number of keys in the full child
    fullChild->n = t - 1;

    // Shift children of parent to make space for the new child
    for (int j = parent->n; j >= index + 1; j--) {
        parent->children[j + 1] = parent->children[j];
    }
    parent->children[index + 1] = newChild;

    // Shift keys of parent to make space for the middle key
    for (int j = parent->n - 1; j >= index; j--) {
        parent->keys[j + 1] = parent->keys[j];
    }
    parent->keys[index] = fullChild->keys[t - 1];
    parent->n++;
}

// Function to insert a key into a non-full node
void insertNonFull(BTreeNode *node, int key) {
    int i = node->n - 1;

    if (node->leaf) {
        while (i >= 0 && key < node->keys[i]) {
```

```
            node->keys[i + 1] = node->keys[i];
            i--;
        }
        node->keys[i + 1] = key;
        node->n++;
    } else {
        while (i >= 0 && key < node->keys[i]) {
            i--;
        }
        i++;
        if (node->children[i]->n == (2 * node->t - 1)) {
            splitChild(node, i);
            if (key > node->keys[i]) {
                i++;
            }
        }
        insertNonFull(node->children[i], key);
    }
}

// Function to insert a key into the B-Tree
void insert(BTreeNode **root, int key) {
    BTreeNode *r = *root;
    if (r->n == (2 * r->t - 1)) {
        BTreeNode *newRoot = createNode(r->t, 0);
        newRoot->children[0] = r;
        *root = newRoot;
        splitChild(newRoot, 0);
        insertNonFull(newRoot, key);
    } else {
        insertNonFull(r, key);
    }
}

// Function to delete a key from the B-Tree (not fully implemented)
void deleteKey(BTreeNode *node, int key) {
    printf("Delete operation not implemented.\n");
}

// Main function to test B-Tree operations
```

```
int main() {
    int t = 3; // Minimum degree
    BTreeNode *root = createNode(t, 1); // Create the root node as a leaf

    // Inserting keys into the B-Tree
    int keys[] = {10, 20, 5, 6, 12, 30, 7, 17};
    for (int i = 0; i < 8; i++) {
        insert(&root, keys[i]);
        printf("After inserting %d: ", keys[i]);
        inorderTraversal(root);
        printf("\n");
    }

    // Final In-order traversal of the tree
    printf("Final In-order traversal: ");
    inorderTraversal(root);
    printf("\n");

    return 0;
}
```


onlinegdb.com/#
Run
Debug
Stop
Share
Save
{ } Beautify
main.c
BTreeNode *root = createNode(t, 1); // Create the root nod
// Inserting keys into the B-Tree
int keys[] = {10, 20, 5, 6, 12, 30, 7, 17};
for (int i = 0; i < 8; i++) {
insert(&root, keys[i]);
printf("After inserting %d: ", keys[i]);
inorderTraversal(root);
printf("\n");
}
// Final In-order traversal of the tree
printf("Final In-order traversal: ");
inorderTraversal(root);
printf("\n");
return 0;
}
After inserting 10: 10
After inserting 20: 10 20
After inserting 5: 5 10 20
After inserting 6: 5 6 10 20
After inserting 12: 5 6 10 12 20
After inserting 30: 5 6 10 30
After inserting 7: 5 6 7 10 30
After inserting 17: 5 6 7 10 17 30
Final In-order traversal: 5 6 7 10 17 30
...Program finished with exit code 0
Press ENTER to exit console.

5. **Prim's Algorithm**

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

#define V 9 // Number of vertices in the graph

// Function to find the vertex with the minimum key value that is not yet included in MST
int minKey(int key[], int mstSet[]) {
   int min = INT_MAX, minIndex;

   for (int v = 0; v < V; v++)
      if (mstSet[v] == 0 && key[v] < min)
         min = key[v], minIndex = v;

   return minIndex;
}

// Function to print the constructed MST
void printMST(int parent[], int graph[V][V]) {
   printf("Edge \tWeight\n");
   for (int i = 1; i < V; i++)
      printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}

// Function to implement Prim's algorithm
void primMST(int graph[V][V]) {
   int parent[V]; // Array to store the MST
   int key[V];    // Key values used to pick the minimum weight edge
   int mstSet[V]; // To represent the set of vertices included in MST

   // Initialize all keys as INFINITE and mstSet[] as false
   for (int i = 0; i < V; i++)
      key[i] = INT_MAX, mstSet[i] = 0;

   // Include the first vertex in the MST
   key[0] = 0;      // Make key 0 so that this vertex is picked first
   parent[0] = -1;  // First node is always the root of the MST
```

```
    // The MST will have V-1 edges
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST set
        mstSet[u] = 1;

        // Update the key and parent arrays of the adjacent vertices
        for (int v = 0; v < V; v++)
            // Update the key only if graph[u][v] is smaller, v is not in MST, and graph[u][v]
is non-zero
            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // Print the constructed MST
    printMST(parent, graph);
}

int main() {
    // Graph represented as an adjacency matrix
    int graph[V][V] = {
        {0, 4, 0, 0, 0, 0, 0, 8, 0},
        {4, 0, 8, 0, 0, 0, 0, 11, 0},
        {0, 8, 0, 7, 0, 4, 0, 0, 2},
        {0, 0, 7, 0, 9, 14, 0, 0, 0},
        {0, 0, 0, 9, 0, 10, 0, 0, 0},
        {0, 0, 4, 14, 10, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 1, 6},
        {8, 11, 0, 0, 0, 0, 1, 0, 7},
        {0, 0, 2, 0, 0, 0, 6, 7, 0},
    };

    // Call Prim's algorithm
    primMST(graph);

    return 0;
}
```

Output:

```
Edge   Weight
0 - 1    4
1 - 2    8
2 - 3    7
3 - 4    9
2 - 5    4
5 - 6    2
6 - 7    1
2 - 8    2
```

6. **Kruskal's Algorithm**

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

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

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

// Create a 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->edge = (struct Edge*)malloc(E * sizeof(struct Edge));
    return graph;
}

// Structure to represent a subset for union-find
struct Subset {
    int parent, rank;
```

```
};

// Find the parent of a node (with path compression)
int find(struct Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

// Union of two subsets by rank
void Union(struct Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare function for qsort
int compareEdges(const void* a, const void* b) {
    struct Edge* edgeA = (struct Edge*)a;
    struct Edge* edgeB = (struct Edge*)b;
    return edgeA->weight > edgeB->weight;
}

// Kruskal's algorithm
void KruskalMST(struct Graph* graph) {
    int V = graph->V;
    struct Edge result[V]; // To store the result MST
    int e = 0; // Index for result[]
    int i = 0; // Index for sorted edges

    // Step 1: Sort all edges in non-decreasing order of weight
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), compareEdges);
```

```
    // Allocate memory for subsets
    struct Subset* subsets = (struct Subset*)malloc(V * sizeof(struct Subset));
    for (int v = 0; v < V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1) {
        struct Edge nextEdge = graph->edge[i++];
        int x = find(subsets, nextEdge.src);
        int y = find(subsets, nextEdge.dest);

        if (x != y) {
            result[e++] = nextEdge;
            Union(subsets, x, y);
        }
    }

    printf("Edges in the MST:\n");
    for (i = 0; i < e; i++)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);

    free(subsets);
}

int main() {
    int V = 5; // Number of vertices
    int E = 7; // Number of edges

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

    // Adding edges
    graph->edge[0] = (struct Edge){0, 1, 4};
    graph->edge[1] = (struct Edge){0, 3, 6};
    graph->edge[2] = (struct Edge){1, 2, 10};
    graph->edge[3] = (struct Edge){1, 3, 2};
    graph->edge[4] = (struct Edge){2, 3, 6};
    graph->edge[5] = (struct Edge){2, 4, 3};
```

```
    graph->edge[6] = (struct Edge){3, 4, 1};

    KruskalMST(graph);

    free(graph->edge);
    free(graph);
    return 0;
}
```

7. **Bellman-Ford Algorithm**

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

#define INF INT_MAX

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

// Bellman-Ford algorithm
void BellmanFord(int V, int E, struct Edge edges[], int src) {
    int distance[V];

    // Step 1: Initialize distances
    for (int i = 0; i < V; i++)
        distance[i] = INF;
    distance[src] = 0;

    // Step 2: Relax all edges V-1 times
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = edges[j].src;
            int v = edges[j].dest;
            int weight = edges[j].weight;
            if (distance[u] != INF && distance[u] + weight < distance[v])
                distance[v] = distance[u] + weight;
```

```
        }
    }

    // Step 3: Check for negative-weight cycles
    for (int j = 0; j < E; j++) {
        int u = edges[j].src;
        int v = edges[j].dest;
        int weight = edges[j].weight;
        if (distance[u] != INF && distance[u] + weight < distance[v]) {
            printf("Graph contains negative weight cycle\n");
            return;
        }
    }

    // Print the distance array
    printf("Vertex\tDistance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d\t%d\n", i, distance[i]);
}

int main() {
    int V = 5; // Number of vertices
    int E = 5; // Number of edges

    struct Edge edges[] = {
        {0, 1, 5},
        {1, 3, 2},
        {4, 3, -1},
        {2, 4, 1},
        {1, 2, 1},
    };

    int src = 0; // Source vertex
    BellmanFord(V, E, edges, src);

    return 0;
}
```

8. Floyd-Warshall Algorithm

```
#include <stdio.h>
#define INF 99999
#define V 5

// Function to print the solution matrix
void printSolution(int dist[V][V]) {
    printf("Shortest distances between every pair of vertices:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

// Floyd-Warshall Algorithm
void floydWarshall(int graph[V][V]) {
    int dist[V][V];

    // Initialize the solution matrix same as the input graph matrix
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    // Update the solution matrix
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distances
    printSolution(dist);
}
```

```
int main() {
    int graph[V][V] = {
        {0, 4, 2, INF, INF},
        {INF, 0, INF, INF, 6},
        {1, INF, 0, 3, INF},
        {INF, INF, INF, 0, 2},
        {INF, INF, INF, INF, 0},
    };

    floydWarshall(graph);
    return 0;
}
```

9. Matrix Chain Multiplication

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

// Function to find the minimum number of multiplications
void matrixChainOrder(int p[], int n) {
    int m[n][n];

    // Initialize the cost of multiplying one matrix as zero
    for (int i = 1; i < n; i++)
        m[i][i] = 0;

    // Fill the table for chains of length L
    for (int L = 2; L < n; L++) {
        for (int i = 1; i < n - L + 1; i++) {
            int j = i + L - 1;
            m[i][j] = INT_MAX;

            for (int k = i; k <= j - 1; k++) {
                int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

    printf("Minimum number of multiplications is %d\n", m[1][n - 1]);
```

```
}

int main() {
    int arr[] = {1, 2, 3, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    matrixChainOrder(arr, n);
    return 0;
}
```

10. Longest Common Subsequence

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

// Function to find the length of the Longest Common Subsequence
int lcs(char* X, char* Y, int m, int n) {
    int L[m + 1][n + 1];

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                L[i][j] = 0;
            else if (X[i - 1] == Y[j - 1])
                L[i][j] = L[i - 1][j - 1] + 1;
            else
                L[i][j] = (L[i - 1][j] > L[i][j - 1]) ? L[i - 1][j] : L[i][j - 1];
        }
    }

    return L[m][n];
}

int main() {
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    printf("Length of LCS is %d\n", lcs(X, Y, m, n));
```

```
    return 0;
}
```