Linear and Binary search (reclusive)

```
#include <stdio.h>
int linearSearch(int arr[], int size, int element) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == element) {
            return i;
        }
    }
    return -1;
}
int binarySearch(int arr[], int left, int right, int element) {
    if (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == element) {
            return mid;
        }
        if (arr[mid] > element) {
            return binarySearch(arr, left, mid - 1, element);
        }
        return binarySearch(arr, mid + 1, right, element);
    }

    return -1;
}
int main() {
    int n, element, linearResult, binaryResult;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Enter the element to search: ");
    scanf("%d", &element);
    linearResult = linearSearch(arr, n, element);
    if (linearResult != -1) {
        printf("Linear Search Result: Element found at index %d\n", linearResult);
    } else {
        printf("Linear Search Result: Element not found\n");
    }
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[i] > arr[j]) {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
    binaryResult = binarySearch(arr, 0, n - 1, element);
    if (binaryResult != -1) {
        printf("Binary Search Result: Element found at index %d (sorted array)\n", binaryResult);
    } else {
        printf("Binary Search Result: Element not found\n");
    }
    return 0;
}
```

Quick sort

```c
#include <stdio.h>
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}
void quickSort(int arr[], int low, int high, int pass) {
    if (low < high) {
        int pi = partition(arr, low, high);
        printf("After Pass %d: ", pass);
        for (int i = 0; i <= high; i++) {
            printf("%d ", arr[i]);
        }
        printf("\n");
        quickSort(arr, low, pi - 1, pass + 1);
        quickSort(arr, pi + 1, high, pass + 1);
    }
}
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Initial Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    quickSort(arr, 0, n - 1, 1);
    printf("Sorted Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Merge sort

```
#include <stdio.h>
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int leftArr[n1], rightArr[n2];
    for (int i = 0; i < n1; i++)
        leftArr[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
```

```
        rightArr[j] = arr[mid + 1 + j];
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}
void mergeSort(int arr[], int left, int right, int pass) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid, pass + 1);
        mergeSort(arr, mid + 1, right, pass + 1);
        merge(arr, left, mid, right);
        printf("After Pass %d: ", pass);
        for (int i = left; i <= right; i++) {
            printf("%d ", arr[i]);
        }
        printf("\n");
    }
}
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Initial Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    mergeSort(arr, 0, n - 1, 1);
    printf("Sorted Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Prims Algorithm

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

```
#define V 5
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;
}
void primMST(int graph[V][V]) {
  int parent[V];
  int key[V];
  int mstSet[V];
  for (int i = 0; i < V; i++) {
    key[i] = INT_MAX;
    mstSet[i] = 0;
  }
  key[0] = 0;
  parent[0] = -1;
  for (int count = 0; count < V - 1; count++) {
    int u = minKey(key, mstSet);
    mstSet[u] = 1;
    for (int v = 0; v < V; v++)
      if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
        parent[v] = u;
        key[v] = graph[u][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]]);
}
int main() {
    int graph[V][V] = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };
    primMST(graph);
    return 0;
}
```

Kruskal’s Algorithm

```
#include <stdio.h>
#define V 5
#define INF 9999
struct Edge {
    int src, dest, weight;
};
int find(int parent[], int i) {
    if (parent[i] != i)
        parent[i] = find(parent, parent[i]);
    return parent[i];
}
void unionSets(int parent[], int rank[], int x, int y) {
    int xroot = find(parent, x);
    int yroot = find(parent, y);
    if (rank[xroot] < rank[yroot]) {
        parent[xroot] = yroot;
    } else if (rank[xroot] > rank[yroot]) {
        parent[yroot] = xroot;
    } else {
        parent[yroot] = xroot;
        rank[xroot]++;
    }
}
void kruskalMST(struct Edge edges[], int edgeCount) {
    struct Edge result[V];
    int parent[V], rank[V];
    for (int i = 0; i < V; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
    int e = 0;
    int i = 0;
    while (e < V - 1 && i < edgeCount) {
        struct Edge nextEdge = edges[i++];
        int x = find(parent, nextEdge.src);
        int y = find(parent, nextEdge.dest);
        if (x != y) {
            result[e++] = nextEdge;
            unionSets(parent, rank, x, y);
        }
    }
    printf("Edge \tWeight\n");
    for (int i = 0; i < e; i++)
        printf("%d - %d \t%d\n", result[i].src, result[i].dest, result[i].weight);
}
int main() {
    struct Edge edges[] = {
        {0, 1, 2}, {1, 2, 3}, {0, 3, 6}, {1, 3, 8},
        {1, 4, 5}, {2, 4, 7}, {3, 4, 9}
    };
    int edgeCount = sizeof(edges) / sizeof(edges[0]);
    kruskalMST(edges, edgeCount);
    return 0;
}
```

Dijstra Algorithm

```
#include <stdio.h>
#include <limits.h>
#define V 5
int minDistance(int dist[], int visited[]) {
    int min = INT_MAX, minIndex;
    for (int v = 0; v < V; v++)
        if (!visited[v] && dist[v] <= min) {
            min = dist[v];
            minIndex = v;
        }
    return minIndex;
}
void dijkstra(int graph[V][V], int src) {
    int dist[V];
    int visited[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        visited[i] = 0;
    }
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, visited);
        visited[u] = 1;
        for (int v = 0; v < V; v++)
            if (!visited[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    printf("Vertex \t Distance from Source %d\n", src);
    for (int i = 0; i < V; i++)
        printf("%d \t\t %d\n", i, dist[i]);
}
int main() {
    int graph[V][V] = {
        {0, 10, 0, 5, 0},
        {10, 0, 1, 2, 0},
        {0, 1, 0, 0, 4},
        {5, 2, 0, 0, 3},
        {0, 0, 4, 3, 0}
    };
    int source = 0;
    dijkstra(graph, source);
    return 0;
}
```

Fractional Knapsack

```
#include <stdio.h>
#include <stdlib.h>
struct Item {
    int weight;
    int value;
    float ratio;
};
int compare(const void *a, const void *b) {
    struct Item *item1 = (struct Item *)a;
    struct Item *item2 = (struct Item *)b;
    if (item2->ratio > item1->ratio) return 1;
    if (item2->ratio < item1->ratio) return -1;
    return 0;
}
void knapsack(int capacity, struct Item items[], int n) {
    qsort(items, n, sizeof(struct Item), compare);
    float totalValue = 0.0;
    int currentWeight = 0;
    for (int i = 0; i < n; i++) {
        if (currentWeight + items[i].weight <= capacity) {
            currentWeight += items[i].weight;
            totalValue += items[i].value;
            printf("Taking full item %d (value: %d, weight: %d)\n", i + 1, items[i].value, items[i].weight);
        } else {
            int remainingCapacity = capacity - currentWeight;
            float fraction = (float)remainingCapacity / items[i].weight;
            totalValue += items[i].value * fraction;
            printf("Taking %.2f fraction of item %d (value: %d, weight: %d)\n", fraction, i + 1, items[i].value, items[i].weight);
            break;
        }
    }
    printf("Total value in knapsack: %.2f\n", totalValue);
}
int main() {
    int capacity = 50;
    struct Item items[] = {
        {10, 60, 0}, {20, 100, 0}, {30, 120, 0}
    };
    int n = sizeof(items) / sizeof(items[0]);
    for (int i = 0; i < n; i++) {
        items[i].ratio = (float)items[i].value / items[i].weight;
    }
    knapsack(capacity, items, n);
    return 0;
}
```

Knapsack Dynamic

```
#include <stdio.h>
int max(int a, int b) {
    return (a > b) ? a : b;
}
int knapsack(int capacity, int weights[], int values[], int n) {
    int dp[n + 1][capacity + 1];
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= capacity; w++) {
            if (i == 0 || w == 0) {
                dp[i][w] = 0;
            } else if (weights[i - 1] <= w) {
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
    return dp[n][capacity];
}
int main() {
    int values[] = {60, 100, 120};
    int weights[] = {10, 20, 30};
    int capacity = 50;
    int n = sizeof(values) / sizeof(values[0]);
    int max_value = knapsack(capacity, weights, values, n);
    printf("Maximum value in knapsack = %d\n", max_value);
    return 0;
}
```

Floid-Warshal

```
#include <stdio.h>
#include <limits.h>
#define V 4
void floydWarshall(int graph[][V]) {
    int dist[V][V];
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (graph[i][j] == 0 && i != j)
                dist[i][j] = INT_MAX;
            else
                dist[i][j] = graph[i][j];
        }
    }
    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] != INT_MAX && dist[k][j] != INT_MAX &&
                    dist[i][j] > dist[i][k] + dist[k][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
    printf("The 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] == INT_MAX)
                printf("INF\t");
            else
                printf("%d\t", dist[i][j]);
        }
        printf("\n");
    }
}
int main() {
    int graph[V][V] = {
        {0, 5, 0, 10},
        {0, 0, 3, 0},
        {0, 0, 0, 1},
        {0, 0, 0, 0}
    };
    floydWarshall(graph);
    return 0;
}
```

Optimal Merge Pattern

```
#include <stdio.h>
#include <stdlib.h>
void minHeapify(int heap[], int n, int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && heap[left] < heap[smallest])
        smallest = left;
    if (right < n && heap[right] < heap[smallest])
        smallest = right;
    if (smallest != i) {
        int temp = heap[i];
        heap[i] = heap[smallest];
        heap[smallest] = temp;
        minHeapify(heap, n, smallest);
    }
}
int extractMin(int heap[], int* n) {
    int min = heap[0];
    heap[0] = heap[*n - 1];
    (*n)--;
    minHeapify(heap, *n, 0);
    return min;
}
void insertMinHeap(int heap[], int* n, int value) {
    (*n)++;
    int i = *n - 1;
    heap[i] = value;
    while (i != 0 && heap[(i - 1) / 2] > heap[i]) {
        int temp = heap[i];
        heap[i] = heap[(i - 1) / 2];
        heap[(i - 1) / 2] = temp;
        i = (i - 1) / 2;
    }
}
int optimalMerge(int files[], int n) {
    int heap[n];
    for (int i = 0; i < n; i++) {
        heap[i] = files[i];
    }
    for (int i = n / 2 - 1; i >= 0; i--) {
        minHeapify(heap, n, i);
    }
    int totalCost = 0;
    while (n > 1) {
        int min1 = extractMin(heap, &n);
        int min2 = extractMin(heap, &n);
        int mergedCost = min1 + min2;
        totalCost += mergedCost;
        insertMinHeap(heap, &n, mergedCost);
    }
    return totalCost;
}
int main() {
    int files[] = {10, 20, 30, 40, 50};
    int n = sizeof(files) / sizeof(files[0]);
    int totalMergeCost = optimalMerge(files, n);
    printf("Total cost of merging files: %d\n", totalMergeCost);
    return 0;
}
```

N-Queens

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 100
int board[MAX][MAX];
void printSolution(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", board[i][j]);
        }
        printf("\n");
    }
}
bool isSafe(int row, int col, int n) {
    int i, j;
    for (i = 0; i < row; i++) {
        if (board[i][col] == 1)
            return false;
    }
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1)
            return false;
    }
    for (i = row, j = col; i >= 0 && j < n; i--, j++) {
        if (board[i][j] == 1)
            return false;
    }
    return true;
}
bool solveNQueens(int n, int row) {
    if (row >= n)
        return true;
    for (int col = 0; col < n; col++) {
        if (isSafe(row, col, n)) {
            board[row][col] = 1;
            if (solveNQueens(n, row + 1))
                return true;
            board[row][col] = 0;
        }
    }
    return false;
```

```
}
int main() {
    int n;
    printf("Enter the number of queens: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            board[i][j] = 0;
        }
    }
    if (solveNQueens(n, 0)) {
        printf("Solution found:\n");
        printSolution(n);
    } else {
        printf("No solution exists.\n");
    }
    return 0;
}
```