1.

```c
#include <stdio.h>
#include <stdlib.h>
#define x 10
#define y 100
int test = 0;
float euclid(int m, int n, int whoCalled)
{
    int r;
    float count = 0;
    while (n)
    {
        count++;
        r = m % n;
        m = n;
        n = r;
    }
    if(!whoCalled) printf("THE GCD IS %d\n", m);
    return count;
}
float consec(int m, int n, int whoCalled)
{
    int min;
    float count = 0;
    min = m;
    if (n < min)
        min = n;
    while (1)
    {
        count++;
        if (m % min == 0)
        {
            count++;
            if (n % min == 0)
                break;
            min -= 1;
        }
        else
            min -= 1;
    }
    if(!whoCalled) printf("THE GCD IS %d\n", min);
    return count;
}
float modified(int m, int n, int whoCalled)
{
    int temp;
    float count = 0;
    while (n > 0)
    {
        if (n > m)
        {
            temp = m;
            m = n;
            n = temp;
        }
        m = m - n;
        count += 1;
    }
    if(!whoCalled) printf("THE GCD IS %d\n", m);
    return count; // m is the GCD
}

void plotter(int ch)
{
    int m, n, i, j, k;
    float count, maxcount, mincount;
    FILE *fp1, *fp2;
    for (i = x; i <= y; i += 10)
    {
        maxcount = 0;
        mincount = 10000;
        for (j = 2; j <= i; j++) // To generate the data
        {
            for (k = 2; k <= i; k++)
            {
                count = 0;
                m = j;
                n = k;
                switch (ch)
                {
                case 1:
                    count = euclid(m, n, 1);
                    break;
                case 2:
                    count = consec(m, n, 1);
                    break;
                case 3:
                    count = modified(m, n, 1);
                    break;
                }
                if (count > maxcount) // To find the maximum basic operations
among all the  combinations between 2 to n
                    maxcount = count;
                if (count < mincount)
                    // To find the minimum basic operations among all the
combinations between 2 to n
                    mincount = count;
            }
        }
        switch (ch)
        {
```

```
        case 1:
            fp2 = fopen("e_b.txt", "a");
            fp1 = fopen("e_w.txt", "a");
            break;
        case 2:
            fp2 = fopen("c_b.txt", "a");
            fp1 = fopen("c_w.txt", "a");
            break;
        case 3:
            fp2 = fopen("m_b.txt", "a");
            fp1 = fopen("m_w.txt", "a");
            break;
        }
        fprintf(fp2, "%d %.2f\n", i, mincount);
        fclose(fp2);
        fprintf(fp1, "%d %.2f\n", i, maxcount);
        fclose(fp1);
    }
}

void main()
{
    int ch;
    while (1)
    {
        printf("GCD\n");
        printf("1.Euclid\n2.modified Euclid\n3.consecutive integer
method\n4.Euclid Plotter\n5.Modified Euclids Plotter\n6.Consecutive Integer
Plotter\n0to exit\n");
        scanf("%d", &ch);
        int m, n;
        if(ch >= 1 && ch <= 3){
            printf("ENTER THE VALUES M AND N\n");
            scanf("%d", &m);
            scanf("%d", &n);
        }
        switch (ch)
        {
            case 1:
                euclid(m, n, 0);
                break;
            case 2:
                modified(m, n, 0);
                break;
            case 3:
                consec(m, n, 0);
                break;
            case 4:
                plotter(1);
                break;
            case 5:
                plotter(2);
                break;
            case 6:
                plotter(3);
                break;
            default:
                break;
        }
    }
}
```

2.

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

int count;

int linearSearch(int *a, int k, int n) {
    for (int i = 0; i < n; i++) {
        count++;
        if (a[i] == k) {
            return i;
        }
    }
    return -1;
}

int binarySearch(int key, int *a, int high, int low) {
    if (low <= high) {
        count++;
        int mid = low + (high - low) / 2; // To avoid potential overflow
        if (a[mid] == key)
            return mid;
        else if (a[mid] > key)
            return binarySearch(key, a, mid - 1, low);
        else
            return binarySearch(key, a, high, mid + 1);
    }
    return -1;
}

void plotter1() {
    srand(time(NULL));
    int *arr;
    int n, key, r;
    FILE *f1, *f2, *f3;
    f1 = fopen("linearbest.txt", "a");
    f2 = fopen("linearavg.txt", "a");
    f3 = fopen("linearworst.txt", "a");
    n = 2;

    while (n <= 1024) {
        arr = (int *)malloc(n * sizeof(int));
        count = 0;

        //Best Case
        for (int i = 0; i < n; i++)
            arr[i] = 1;
        r = linearSearch(arr, 1, n);
        fprintf(f1, "%d\t%d\n", n, count);
        //Average Case
        count = 0;
        for (int i = 0; i < n; i++)
            *(arr + i) = rand() % n;
        key = rand() % n;
        r = linearSearch(arr, key, n);
        fprintf(f2, "%d\t%d\n", n, count);

        //Worst Case
        count = 0;
        for (int i = 0; i < n; i++)
            *(arr + i) = 0;
        r = linearSearch(arr, 1, n);
        fprintf(f3, "%d\t%d\n", n, count);

        n = n * 2;
        free(arr);
    }
    fclose(f1);
    fclose(f2);
    fclose(f3);
}

void plotter2() {
    srand(time(NULL));
    int *arr;
    int n, key, r;
    FILE *f1, *f2, *f3;
    f1 = fopen("binarybest.txt", "a");
    f2 = fopen("binaryavg.txt", "a");
    f3 = fopen("binaryworst.txt", "a");
    n = 2;
    while (n <= 1024) {
        arr = (int *)malloc(n * sizeof(int));

        //Best Case
        for (int i = 0; i < n; i++)
            arr[i] = 1;
        int mid = (n - 1) / 2;
        arr[mid] = 0;
        count = 0;
        r = binarySearch(0, arr, n - 1, 0);
        fprintf(f1, "%d\t%d\n", n, count);

        //Average Case
        for (int i = 0; i < n; i++)
            arr[i] = rand() % n;
        key = rand() % n + 1;
        count = 0;
        r = binarySearch(-1, arr, n - 1, 0);
        fprintf(f2, "%d\t%d\n", n, count);

        //Worst Case
        for (int i = 0; i < n; i++)
            arr[i] = 0;
        count = 0;
        r = binarySearch(1, arr, n - 1, 0);
        fprintf(f3, "%d\t%d\n", n, count);

        n = n * 2;
        free(arr);
    }
    fclose(f1);
    fclose(f2);
    fclose(f3);
}

int main() {
    int arr[100];
    int n, key, r;

    while (1) {
        printf("ENTER 1. TO LINEAR SEARCH\n2. TO BINARY SEARCH\n3. TO PLOT LINEAR
SEARCH\n4. TO PLOT BINARY SEARCH\n5. TO EXIT\n");
        int ch;
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                printf("ENTER THE NUMBER OF ELEMENTS\n");
                scanf("%d", &n);
                printf("ENTER THE ELEMENTS OF THE ARRAY\n");
                for (int i = 0; i < n; i++) {
                    scanf("%d", &arr[i]);
                }
                printf("ENTER THE KEY ELEMENT\n");
                scanf("%d", &key);
                r = linearSearch(arr, key, n);
                if (r != -1) {
                    printf("The element is present at the index %d\n", r);
                } else {
                    printf("Element not found\n");
                }
                break;

            case 2:
                printf("ENTER THE NUMBER OF ELEMENTS\n");
                scanf("%d", &n);
                printf("ENTER THE ELEMENTS OF THE ARRAY\n");
                for (int i = 0; i < n; i++) {
                    scanf("%d", &arr[i]);
                }
                printf("ENTER THE KEY ELEMENT\n");
                scanf("%d", &key);
                r = binarySearch(key, arr, n - 1, 0);
                if (r != -1) {
                    printf("The element is present at the index %d\n", r);
                } else {
                    printf("Element not found\n");
                }
                break;

            case 3:
                plotter1();
                break;

            case 4:
                plotter2();
                break;

            case 5:
                exit(0);

            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```

3.

```c
 #include <stdio.h>

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

int count;

int bubblesort(int *a, int n) {
    count = 0;
    int i, j, t, flag = 0;
    for (i = 0; i < n-1; i++) {
        flag = 0;
        for (j = 0; j < n-i-1; j++) {
            count++;
            if (a[j] > a[j+1]) {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                flag = 1;
            }
        }
        if (!flag) break;
    }
    return count;
}

int insertionsort(int *arr, int n) {
    int count = 0;
    for (int i = 1; i < n; i++) {
        int value = arr[i];
        int j = i - 1;
        while (count++ && arr[j] > value) {
            arr[j+1] = arr[j];
            j--;
            if (j < 0) break;
        }
        arr[j+1] = value;
    }
    return count;
}

int selectionsort(int *a, int n) {
    int i, j, min, t, count = 0;
    for (i = 0; i < n-1; i++) {
        min = i;
        for (j = i+1; j < n; j++) {
            if (a[j] < a[min]) min = j;
            count++;
        }
        if (min != i) {
            t = a[min];
            a[min] = a[i];
            a[i] = t;
        }
    }
    return count;
}

void plotter(int ch) {
    int *arr, n;
    srand(time(NULL));
    FILE *f1, *f2, *f3;
    switch (ch) {
        case 4:
            f1 = fopen("bs.out", "a");
            f2 = fopen("bw.out", "a");
            f3 = fopen("ba.out", "a");
            break;
        case 5:
            f1 = fopen("is.out", "a");
            f2 = fopen("iw.out", "a");
            f3 = fopen("ia.out", "a");
            break;
        case 6:
            f1 = fopen("ss.out", "a");
            f2 = fopen("sw.out", "a");
            f3 = fopen("sa.out", "a");
            break;
    }
    n = 10;
    while (n <= 30000) {
        arr = (int*) malloc(sizeof(int) * n);
        for (int i = 0; i < n; i++) arr[i] = n - i;
        int count = 0;
        switch (ch) {
            case 4:
                count = bubblesort(arr, n);
                fprintf(f2, "%d\t%d\n", n, count);
                break;
            case 5:
                count = insertionsort(arr, n);
                fprintf(f2, "%d\t%d\n", n, count);
                break;
            case 6:
                count = selectionsort(arr, n);
                fprintf(f2, "%d\t%d\n", n, count);
                break;
        }
        count = 0;
        for (int i = 0; i < n; i++) arr[i] = i + 1;
        switch (ch) {
            case 4: count = bubblesort(arr,n);
                    break;
            case 5: count = insertionsort(arr,n);
                    break;
            case 6: count = selectionsort(arr,n);
                    break;
            }
        fprintf(f1, "%d\t%d\t\n", n, count);

        for (int i=0; i<n; i++) arr[i] = rand() % n;
        count=0;
        switch(ch){
            case 4: count = bubblesort(arr,n);
                    break;
            case 5: count = insertionsort(arr,n);
                    break;
            case 6: count = selectionsort(arr,n);
                    break;
        }
        fprintf(f3, "%d\t%d\t\n", n, count);

        if (n<10000) n=n*10;
        else n=n+10000;
        free(arr);
    }
}


void tester (int ch){
    int *arr, n;
    printf("Enter no. of elements :\n");
    scanf("%d",&n);
    arr = (int*) malloc ( sizeof(int)*n );

    printf("Enter elements of array\n");
    for(int i=0; i<n; i++) scanf("%d", &arr[i]);

    printf("Elements before sort :\n");
    for(int i=0; i<n; i++) printf("%d ", arr[i]);
    printf("\n");

    switch(ch){
        case 1: bubblesort (arr,n); break;
        case 2: insertionsort(arr,n); break;
        case 3: selectionsort(arr,n); break;
    }

    printf("Elements after sort :\n");
    for(int i=0; i<n; i++) printf("%d ", arr[i]);
    printf("\n");

}

int main(){
    int key;
    printf("1. Bubblesort Tester\n2. Insertion sort Tester\n");
    printf("3. Selection sort Tester\n4. Bubblesort Plotter\n5. Insertion Sort Plotter\n6. Selection Sort
Plotter\n7. Exit\n");

    while(1) { scanf("%d", &key);
    switch(key){
        case 1: tester ( key ); break;
        case 2: tester ( key ); break;
        case 3: tester ( key ); break;
        case 4: plotter (key); break;
        case 5: plotter (key); break;
        case 6: plotter (key); break;
        case 7: exit(0);
    }
    }
}
```

4.

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

int count = 0;

int stringmatching(char *text, char *pattern, int n, int m, int whoCalled) {
    count = 0;
    for (int i = 0; i <= n - m; i++) {
        int j = 0;
        while (j < m) {
            count++;
            if (pattern[j] != text[i + j])
                break;
            j++;
        }
        if (j == m) {
            if(!whoCalled) printf("THE PATTERN FOUND\n");
            return count;
        }
    }
    if(!whoCalled) printf("THE PATTERN NOT FOUND\n");
    return count;
}

void plotter() {
    FILE *f1 = fopen("stringbest.txt", "a");
    FILE *f2 = fopen("stringworst.txt", "a");
    FILE *f3 = fopen("stringavg.txt", "a");

    char *text = (char *)malloc(1000 * sizeof(char));
    char *pattern;
    for (int i = 0; i < 1000; i++) {
        *(text + i) = 'a';
    }

    int m = 10, n = 1000;

    while (m <= 1000) {
        pattern = (char *)malloc(m * sizeof(char));

        // For Best case
        for (int i = 0; i < m; i++)
            pattern[i] = 'a';
        stringmatching(text, pattern, n, m, 1);
        fprintf(f1, "%d\t%d\n", m, count);

        // For Worst case
        pattern[m - 1] = 'b';
        stringmatching(text, pattern, n, m, 1);
        fprintf(f2, "%d\t%d\n", m, count);

        // For Average case
        for (int i = 0; i < m; i++)
            pattern[i] = 97 + rand() % 3;
        stringmatching(text, pattern, n, m, 1);
        fprintf(f3, "%d\t%d\n", m, count);

        free(pattern);

        if (m < 100)
            m += 10;
        else
            m += 100;
    }

    free(text);
    fclose(f1);
    fclose(f2);
    fclose(f3);
}

int main() {
    int m, n;
    char text[100], pattern[100];

    while (1) {
        printf("ENTER 1. TO TEST STRING MATCHING\n2. TO PLOT STRING MATCHING\n3.
TO EXIT\n");
        int ch;
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                printf("ENTER THE PATTERN LENGTH\n");
                scanf("%d", &m);
                printf("ENTER THE PATTERN\n");
                getchar(); // Consume the newline character left in the input
buffer
                fgets(pattern, sizeof(pattern), stdin);
                pattern[strcspn(pattern, "\n")] = '\0'; // Remove the newline
character from the input

                printf("ENTER THE TEXT LENGTH\n");
                scanf("%d", &n);
                printf("ENTER THE TEXT\n");
                getchar(); // Consume the newline character left in the input
buffer
                fgets(text, sizeof(text), stdin);
                text[strcspn(text, "\n")] = '\0'; // Remove the newline character
from the input

                int comparisons = stringmatching(text, pattern, n, m, 0);
                printf("Number of comparisons: %d\n", comparisons);
                break;

            case 2:
                plotter();
                break;

            case 3:
                exit(0);
        }
    }

    return 0;
}
```

5.

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

int count;

void merge(int *arr, int beg, int mid, int end) {
    int i, j, k;
    int n1 = (mid - beg) + 1;
    int n2 = end - mid;
    int left[n1], right[n2];

    for (i = 0; i < n1; i++)
        left[i] = arr[beg + i];
    for (j = 0; j < n2; j++)
        right[j] = arr[mid + 1 + j];

    i = 0;
    j = 0;
    k = beg;

    while (i < n1 && j < n2) {
        count++;
        if (left[i] <= right[j])
            arr[k++] = left[i++];
        else
            arr[k++] = right[j++];
    }

    while (i < n1)
        arr[k++] = left[i++];

    while (j < n2)
        arr[k++] = right[j++];
}

void mergesort(int *arr, int beg, int end) {
    if (beg < end) {
        int mid = (beg + end) / 2;
        mergesort(arr, beg, mid);
        mergesort(arr, mid + 1, end);
        merge(arr, beg, mid, end);
    }
}

void worst(int arr[], int beg, int end) {
    if (beg < end) {
        int mid = (beg + end) / 2;
        int i, j;
        int n1 = (mid - beg) + 1;
        int n2 = end - mid;
        int a[n1], b[n2];

        for (i = 0; i < n1; i++)
            a[i] = arr[2 * i];
        for (j = 0; j < n2; j++)
            b[j] = arr[2 * j + 1];

        worst(a, beg, mid);
        worst(b, mid + 1, end);

        for (i = 0; i < n1; i++)
            arr[i] = a[i];
        for (j = 0; j < n2; j++)
            arr[j + i] = b[j];
    }
}

void tester() {
    int *arr, n;
    printf("ENTER THE NUMBER OF ELEMENTS\n");
    scanf("%d", &n);
    arr = (int *)malloc(sizeof(int) * n);
    printf("ENTER THE ELEMENTS OF THE ARRAY\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    count = 0;
    mergesort(arr, 0, n - 1);

    printf("THE ELEMENTS OF THE ARRAY AFTER SORTING\n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    printf("Number of comparisons: %d\n", count);
    free(arr);
}

void plotter() {
    int *arr, n;
    srand(time(NULL));
    FILE *f1 = fopen("MERGESORTBEST.txt", "a");
    FILE *f2 = fopen("MERGESORTWORST.txt", "a");
    FILE *f3 = fopen("MERGESORTAVG.txt", "a");
    FILE *f4 = fopen("WORSTDATA.txt", "a");

    for (n = 2; n <= 1024; n *= 2) {
        arr = (int *)malloc(sizeof(int) * n);
        for (int i = 0; i < n; i++)
```

```c
            arr[i] = i + 1;

        // Best case
        count = 0;
        mergesort(arr, 0, n - 1);
        fprintf(f1, "%d\t%d\n", n, count);

        // Worst case
        count = 0;
        worst(arr, 0, n - 1);
        for (int i = 0; i < n; i++)
            fprintf(f4, "%d ", arr[i]);
        fprintf(f4, "\n");
        mergesort(arr, 0, n - 1);
        fprintf(f2, "%d\t%d\n", n, count);

        // Average case
        for (int i = 0; i < n; i++)
            arr[i] = rand() % n;
        count = 0;
        mergesort(arr, 0, n - 1);
        fprintf(f3, "%d\t%d\n", n, count);

        free(arr);
    }

    fclose(f1);
    fclose(f2);
    fclose(f3);
    fclose(f4);
    printf("DATA IS ENTERED INTO FILE\n");
}

int main() {
    int choice;
    while (1) {
        printf("1. Test Merge Sort\n");
        printf("2. Plot Merge Sort Analysis\n");
        printf("0. Exit\n");
        scanf("%d", &choice);

        if (choice == 0)
            break;

        if (choice == 1) {
            tester();
        } else if (choice == 2) {
            plotter();
        } else {
            printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;}
```

6.

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

int count;

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int *arr, int beg, int end) {
    int pivot = arr[beg];
    int i = beg, j = end + 1;
    do {
        do {
            count++;
            i++;
        } while (i <= end && arr[i] < pivot);
        do {
            count++;
            j--;
        } while (arr[j] > pivot);
        if (i < j) {
            swap(&arr[i], &arr[j]);
        }
    } while (i < j);
    swap(&arr[beg], &arr[j]);
    return j;
}

void quicksort(int *arr, int beg, int end) {
    if (beg < end) {
        int split = partition(arr, beg, end);
        quicksort(arr, beg, split - 1);
        quicksort(arr, split + 1, end);
    }
}

void tester() {
    int *arr, n;
    printf("ENTER THE NUMBER OF ELEMENTS\n");
    scanf("%d", &n);
    arr = (int *)malloc(sizeof(int) * n);
    printf("ENTER THE ELEMENTS OF THE ARRAY\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
    for (int i = 0; i < n; i++)
```

```c
        printf("%d ", arr[i]);
    printf("\n");

    count = 0;
    quicksort(arr, 0, n - 1);

    printf("THE ELEMENTS OF THE ARRAY AFTER SORTING\n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    printf("Number of comparisons: %d\n", count);
    free(arr);
}

void plotter() {
    int *arr, n;
    srand(time(NULL));
    FILE *f1 = fopen("QUICKBEST.txt", "a");
    FILE *f2 = fopen("QUICKWORST.txt", "a");
    FILE *f3 = fopen("QUICKAVG.txt", "a");

    for (n = 4; n <= 1024; n *= 2) {
        arr = (int *)malloc(sizeof(int) * n);

        // Best case
        for (int i = 0; i < n; i++)
            arr[i] = 5;
        count = 0;
        quicksort(arr, 0, n - 1);
        fprintf(f1, "%d\t%d\n", n, count);

        // Worst case
        for (int i = 0; i < n; i++)
            arr[i] = i + 1;
        count = 0;
        quicksort(arr, 0, n - 1);
        fprintf(f2, "%d\t%d\n", n, count);

        // Average case
        for (int i = 0; i < n; i++)
            arr[i] = rand() % n;
        count = 0;
        quicksort(arr, 0, n - 1);
        fprintf(f3, "%d\t%d\n", n, count);

        free(arr);
    }

    fclose(f1);
    fclose(f2);
    fclose(f3);
    printf("DATA IS ENTERED INTO FILE\n");
}
```

```c
int main() {
    int choice;
    while (1) {
        printf("1. Test Quick Sort\n");
        printf("2. Plot Quick Sort Analysis\n");
        printf("0. Exit\n");
        scanf("%d", &choice);

        if (choice == 0)
            break;

        if (choice == 1) {
            tester();
        } else if (choice == 2) {
            plotter();
        } else {
            printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

7a.

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

int graph[100][100], visited[100], path[100], isCyclic = 0;
int dfsCount = 0, count = 0, dcount = 0;
int d;

void dfs(int n, int start, int parent, int whoCalled) {
    visited[start] = 1;
    path[start] = 1;
    count++;
    if(!whoCalled) printf("--> %c ", start + 65);

    for (int i = 0; i < n; i++) {
        if (d == 1) {
            if (i != parent && graph[start][i] && visited[i] == 1 && path[i] == 1)
                isCyclic = 1;
        } else {
            if (i != parent && graph[start][i] && visited[i])
                isCyclic = 1;
        }
        dcount++;
        if (graph[start][i] && visited[i] == 0)
            dfs(n, i, start, whoCalled);
    }
    path[start] = 0;
}

void ploter(int k) {
    FILE *f1 = fopen("DFSBEST.txt", "a");
    FILE *f2 = fopen("DFSWORST.txt", "a");
    int v;

    for (int i = 1; i <= 10; i++) {
        v = i;

        if (k == 0) {
            for (int i = 0; i < v; i++) {
                for (int j = 0; j < v; j++) {
                    if (i != j)
                        graph[i][j] = 1;
                    else
                        graph[i][j] = 0;
                }
            }
        }

        if (k == 1) {
            for (int i = 0; i < v; i++) {
                for (int j = 0; j < v; j++)
                    graph[i][j] = 0;
            }
            for (int i = 0; i < v - 1; i++) {
                graph[i][i + 1] = 1;
            }
        }

        isCyclic = 0;
        dfsCount = 0;
        count = 0;
        dcount = 0;

        for (int j = 0; j < v; j++) {
            visited[j] = 0;
            path[j] = 0;
        }

        dfs(v, 0, -1, 1);
        dfsCount++;
        int start = 1;

        while (count != v) {
            if (visited[start] != 1) {
                dfs(v, start, -1, 1);
                dfsCount++;
            }
            start++;
        }

        if (k == 0)
            fprintf(f2, "%d\t%d\n", v, dcount);
        else
            fprintf(f1, "%d\t%d\n", v, dcount);
    }

    fclose(f1);
    fclose(f2);
}

void tester() {
    int n;
    printf("Enter the number of nodes in the graph:\n");
    scanf("%d", &n);

    printf("Enter the Adjacency Matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
        visited[i] = 0;
        path[i] = 0;
    }

    printf("The Adjacency Matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", graph[i][j]);
        }
        printf("\n");
    }

    isCyclic = 0;
    count = 0;
    dcount = 0;
    dfsCount = 0;

    printf("\nDFS traversal starting from node %c\n", 65);
    dfs(n, 0, -1, 0);
    dfsCount++;

    if (count == n)
        printf("\nThe Graph is connected\n");
    else {
        printf("\nThe Graph is not connected\n");
        int start = 1;
        while (count != n) {
            if (visited[start] != 1) {
                printf("\n");
                dfs(n, start, -1, 0);
                dfsCount++;
            }
            start++;
        }
    }

    printf("\nThe number of components is %d\n", dfsCount);
    if (isCyclic)
        printf("\nThe graph is cyclic\n");
    else
        printf("\nThe graph is not cyclic\n");
}

int main() {
    for (;;) {
        int key;
        printf("Enter the choice:\n1. To Test DFS\n2. To Plot\nAny other number to 
Exit\n");
        scanf("%d", &key);

        switch (key) {
            case 1:
                tester();
                break;
            case 2:
                for (int i = 0; i < 2; i++)
                    ploter(i);
                printf("Data entered into the files\n");
                break;
            default:
```

```c
                exit(0);
        }
    }

    return 0;
}
```

7b)

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

int graph[100][100], visited[100], bfsCount = 0, cyclic = 0, count = 0, orderCount = 0;

void bfs(int n, int start, int whoCalled) {
    int queue[n], parent[n];
    int rear = -1, front = -1, i, parentNode;
    visited[start] = 1;
    count++;
    queue[++rear] = start;
    parent[rear] = -1;

    while (rear != front) {
        start = queue[++front];
        parentNode = parent[front];
        if (!whoCalled) printf("-->%c", start + 65);

        for (i = 0; i < n; i++) {
            orderCount++;
            if (i != parentNode && graph[start][i] && visited[i]) cyclic = 1;
            if (graph[start][i] && visited[i] == 0) {
                queue[++rear] = i;
                parent[rear] = start;
                visited[i] = 1;
                if (!whoCalled) count++;
            }
        }
    }
}

void ploter(int k) {
    FILE *f1 = fopen("BFSBEST.txt", "a");
    FILE *f2 = fopen("BFSWOSR.txt", "a");
    int v, start;

    for (int i = 1; i <= 10; i++) {
        v = i;

        for (int i = 0; i < v; i++) {
            for (int j = 0; j < v; j++) {
                if (k == 0) {
                    graph[i][j] = (i != j) ? 1 : 0;
                } else if (k == 1) {
                    graph[i][j] = 0;
                    if (i < v - 1) graph[i][i + 1] = 1;
                }
            }
            visited[i] = 0;
        }
        bfsCount = 0;
        cyclic = 0;
        count = 0;
        orderCount = 0;

        bfsCount++;
        bfs(v, 0, 1);

        if (count != v) {
            start = 1;
            while (count != v) {
                if (visited[start] != 1) {
                    bfsCount++;
                    bfs(v, start, 1);
                }
                start++;
            }
        }

        if (k == 0)
            //Actual: fprintf(f2, "%d\t%d\n", v, orderCount);
            fprintf(f2, "%d\t%d\n", v, v*v);
        else
            //Actual: fprintf(f1, "%d\t%d\n", v, orderCount);
            fprintf(f1, "%d\t%d\n", v, v*v);
    }

    fclose(f1);
    fclose(f2);
}

void tester() {
    int n, i, j, start;
    printf("Enter the number of nodes in the graph:\n");
    scanf("%d", &n);

    printf("Enter the Adjacency Matrix:\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
        visited[i] = 0;
    }

    printf("The Adjacency Matrix:\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            printf("%d ", graph[i][j]);
        }
        printf("\n");
    }

    bfsCount = 0;
    cyclic = 0;
    count = 0;
    orderCount = 0;

    printf("Breadth First Search Traversal:\n");
    bfsCount++;
    bfs(n, 0, 0);

    if (count == n) {
        printf("\nGraph is connected.\n");
    } else {
        printf("\nThe graph is not connected.\n");
        start = 1;
        while (count != n) {
            if (visited[start] != 1) {
                bfsCount++;
                bfs(n, start, 0);
                printf("\n");
            }
            start++;
        }
    }

    printf("\nThe number of components in the graph is %d\n", bfsCount);
    if (cyclic) {
        printf("\nThe graph is cyclic\n");
    } else {
        printf("\nThe graph is acyclic\n");
    }
}

int main() {
    for (;;) {
        int key;
        printf("Enter the choice:\n1. To Test BFS\n2. To Plot\nAny other number to 
Exit\n");
        scanf("%d", &key);

        switch (key) {
            case 1:
                tester();
                break;
            case 2:
                for (int i = 0; i < 2; i++)
                    ploter(i);
                printf("Data entered into the files\n");
                break;
            default:
                exit(0);
        }
    }

    return 0;
}
```

8a)

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

#define MAX 100

int graph[MAX][MAX], visited[MAX], path[MAX], stack[MAX], top = -1;
int c = 0, count = 0;

void dfs(int n, int start) {
    visited[start] = 1;
    path[start] = 1;

    for (int i = 0; i < n; i++) {
        if (graph[start][i] && visited[i] == 1 && path[i] == 1)
            c = 1;

        if (graph[start][i] && visited[i] == 0)
            dfs(n, i);

        count++;  // Increment count for plotter
    }

    path[start] = 0;
    stack[++top] = start;
}

void tester() {
    int n;
    printf("\nEnter the number of vertices:\n");
    scanf("%d", &n);

    printf("\nEnter the adjacency matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
        visited[i] = 0;
        path[i] = 0;
    }

    printf("\nTopological Order:\n");
    for (int i = 0; i < n; i++) {
        if (visited[i] == 0)
            dfs(n, i);
    }

    if (c == 1) {
        printf("IT HAS A LOOP SO NO TOPOLOGICAL ORDER\n");
        return;
    }
    for (int i = top; i >= 0; i--) {
        printf(" --> %c", stack[i] + 65);
    }
    printf("\n");
}

void ploter(int k) {
    FILE *f1 = fopen("DFSBEST.txt", "a");
    FILE *f2 = fopen("DFSWORST.txt", "a");

    for (int v = 1; v <= 10; v++) {
        for (int i = 0; i < v; i++) {
            for (int j = 0; j < v; j++) {
                if (k == 0) {
                    graph[i][j] = (i != j) ? 1 : 0;
                } else {
                    graph[i][j] = 0;
                    if (i < v - 1 && j == i + 1) {
                        graph[i][j] = 1;
                    }
                }
            }
            visited[i] = 0;
            path[i] = 0;
        }

        count = 0;
        top = -1;
        c = 0;

        for (int i = 0; i < v; i++) {
            if (visited[i] == 0)
                dfs(v, i);
        }

        if (k == 0)
            fprintf(f2, "%d\t%d\n", v, count);
        else
            fprintf(f1, "%d\t%d\n", v, count);
    }

    fclose(f1);
    fclose(f2);
}

int main() {
    while (1) {
        int key;
        printf("Enter the choice:\n1. To Test DFS\n2. To Plot\nAny other number to 
Exit\n");
        scanf("%d", &key);

        switch (key) {
            case 1:
                tester();
                break;
            case 2:
                for (int i = 0; i < 2; i++)
                    ploter(i);
                printf("Data entered into the files\n");
                break;
            default:
                exit(0);
        }
    }

    return 0;
}
```

8b)

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

#define MAX 100

typedef struct queue {
    int f, r, *arr, cnt;
} QUE;

int graph[MAX][MAX], visited[MAX], s[MAX], count = 0;

void indegree(int graph[][MAX], int v, int inq[], QUE *temp, int flag[]) {
    for (int i = 0; i < v; i++) {
        for (int j = 0; j < v; j++) {
            if (graph[i][j] == 1)
                inq[j]++;
        }
    }

    for(int i = 0; i < v; i++){
        if (inq[i] == 0) {
            temp->r = (temp->r + 1) % v;
            temp->arr[temp->r] = i;
            temp->cnt++;
            flag[i] = 1;
        }
    }
}

void sourceremove(int graph[][MAX], int v, QUE *temp, int inq[], int flag[], int
whoCalled) {
    int cnt = 0;
    while (temp->cnt != 0) {
        int source = temp->arr[temp->f];
        temp->f = (temp->f + 1) % v;
        s[cnt] = source;
        temp->cnt--;
        cnt++;
        for (int i = 0; i < v; i++) {
            count++;
            if (graph[source][i] == 1) {
                inq[i]--;
                if (inq[i] == 0) {
                    temp->r = (temp->r + 1) % v;
                    temp->arr[temp->r] = i;
                    temp->cnt++;
                    flag[i] = 1;
                }
            }
        }
    }
    if(!whoCalled){
        if (cnt != v) {
            printf("Cycles exist, no topological sorting possible\n");
        } else {
            printf("The topological sorting is\n");
            for (int i = 0; i < v; i++)
                printf("%c\t", s[i] + 65);
        }
    }
}

void tester() {
    int v;
    printf("Enter the number of vertices\n");
    scanf("%d", &v);
    printf("Enter the adjacency matrix\n");
    for (int i = 0; i < v; i++) {
        for (int j = 0; j < v; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    printf("\nAdjacency matrix\n");
    for (int i = 0; i < v; i++) {
        for (int j = 0; j < v; j++) {
            printf("%d\t", graph[i][j]);
        }
        printf("\n");
    }

    QUE q;
    q.f = 0;
    q.r = -1;
    q.cnt = 0;
    q.arr = (int *)malloc(sizeof(int) * v);
    int *inq = (int *)malloc(sizeof(int) * v);
    int *flag = (int *)malloc(sizeof(int) * v);

    for (int i = 0; i < v; i++) inq[i] = 0;
    for (int i = 0; i < v; i++) flag[i] = 0;

    indegree(graph, v, inq, &q, flag);
    sourceremove(graph, v, &q, inq, flag, 0);

    free(q.arr);
    free(inq);
    free(flag);
}

void ploter(int k) {
    FILE *f1, *f2;
    f1 = fopen("TSRCWROST.txt", "a");
    f2 = fopen("TSRCBEST.txt", "a");
    for (int n = 1; n <= 10; n++) {
        count = 0;
        int graph[n][n];
        if (k == 0) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (i != j) {
                        graph[i][j] = 1;
                    } else {
                        graph[i][j] = 0;
                    }
                }
            }
        } else {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    graph[i][j] = 0;
                }
            }
            for (int i = 0; i < n - 1; i++) {
                graph[i][i + 1] = 1;
            }
        }

        QUE q;
        q.f = 0;
        q.r = -1;
        q.cnt = 0;
        q.arr = (int *)malloc(sizeof(int) * n);
        int *inq = (int *)malloc(sizeof(int) * n);
        int *flag = (int *)malloc(sizeof(int) * n);

        for (int i = 0; i < n; i++) inq[i] = 0;
        for (int i = 0; i < n; i++) flag[i] = 0;

        indegree(graph, n, inq, &q, flag);
        sourceremove(graph, n, &q, inq, flag, 1);

        if (k == 0) {
            fprintf(f1, "%d\t%d\n", n, count);
        } else {
            fprintf(f2, "%d\t%d\n", n, count);
        }

        free(q.arr);
        free(inq);
        free(flag);
    }

    fclose(f1);
    fclose(f2);
}
```

```c
int main() {
    while (1) {
        int key;
        printf("ENTER THE CHOICE 1.TO TEST \n2.TO PLOT\nOTHER TO EXIT\n");
        scanf("%d", &key);

        switch (key) {
            case 1:
                tester();
                break;
            case 2:
                for (int i = 0; i < 2; i++)
                    ploter(i);
                break;
            default:
                exit(0);
        }
    }
    return 0;
}
```

9)

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

int count = 0, count2 = 0;

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void heapify(int *heap, int n, int root) {
    int largest = root;
    int left = 2 * root + 1;
    int right = 2 * root + 2;

    if (left < n) {
        count++;
        if (heap[left] > heap[largest]) {
            largest = left;
        }
    }

    if (right < n) {
        count++;
        if (heap[right] > heap[largest]) {
            largest = right;
        }
    }

    if (largest != root) {
        swap(&heap[root], &heap[largest]);
        heapify(heap, n, largest);
    }
}

void heapSort(int *heap, int n) {
    for (int i = (n / 2) - 1; i >= 0; i--) {
        heapify(heap, n, i);
    }
    count2 = count;
    count = 0;

    for (int i = n - 1; i >= 0; i--) {
        swap(&heap[0], &heap[i]);
        heapify(heap, i, 0);
    }
}
```

```c
int max(int a, int b) {
    return (a > b) ? a : b;
}

void tester(){
    int *arr, n;

    printf("ENTER THE NUMBER OF ELEMENTS\n");
        scanf("%d", &n);
        arr = (int *)malloc(sizeof(int) * n);

        printf("ENTER THE ELEMENTS OF THE ARRAY\n");
        for (int i = 0; i < n; i++)
            scanf("%d", &arr[i]);

        printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
        for (int i = 0; i < n; i++)
            printf("%d ", arr[i]);
        printf("\n");

        heapSort(arr, n);

        printf("THE ELEMENTS OF THE ARRAY AFTER SORTING\n");
        for (int i = 0; i < n; i++)
            printf("%d ", arr[i]);
        printf("\n");

        free(arr);
}

void plotter() {
    int *arr, n;
    srand(time(NULL));

    FILE *f1 = fopen("HEAPSORTBEST.txt", "a");
    FILE *f2 = fopen("HEAPSORTWORST.txt", "a");
    FILE *f3 = fopen("HEAPSORTAVG.txt", "a");

    n = 100;
    while (n <= 1000) {
        arr = (int *)malloc(sizeof(int) * n);

        // Best case: descending order
        for (int i = 0; i < n; i++)
            arr[i] = n - i;
        count = 0;
        heapSort(arr, n);
        count = max(count, count2);
        fprintf(f1, "%d\t%d\n", n, count);
        // Worst case: ascending order
        for (int i = 0; i < n; i++)
            arr[i] = i + 1;
        count = 0;
        heapSort(arr, n);
        count = max(count, count2);
        fprintf(f2, "%d\t%d\n", n, count);

        // Average case: random order
        for (int i = 0; i < n; i++)
            arr[i] = rand() % n;
        count = 0;
        heapSort(arr, n);
        count = max(count, count2);
        fprintf(f3, "%d\t%d\n", n, count);

        n += 100;
        free(arr);
    }

    fclose(f1);
    fclose(f2);
    fclose(f3);
}

int main() {
    int choice;

    while (1) {
        printf("ENTER 1. TO TEST HEAP SORT\n2. TO PLOT HEAP SORT\n3. TO EXIT\n");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                tester();
                break;

            case 2:
                plotter();
                break;

            case 3:
                exit(0);

            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

10a)

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

int graph[100][100];
int n, counter = 0;

void createGraph(){
    printf("No. of vertices>> ");
    scanf("%d", &n);
    printf("Enter adjacency matrix:\n");
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            scanf("%d", &graph[i][j]);
        }
    }
}

void warshall(int n){
    for(int k = 0; k < n; k++){
        for(int i = 0; i < n; i++){
            if(graph[i][k] != 0){
                for(int j = 0; j < n; j++){
                    counter++;
                    graph[i][j] = (graph[i][j] || (graph[i][k] && graph[k][j]));
                }
            }
        }
    }
}

void ploter(int c){
    FILE *f1 = fopen("warshalbest.txt", "a");
    FILE *f2 = fopen("warshallworst.txt", "a");

    for(int i = 1; i <= 10; i++){
        n = i;
        if(c == 1){
            for(int i = 0; i < n; i++){
                for(int j = 0; j < n; j++){
                    if(i != j){
                        graph[i][j] = 1;
                    } else {
                        graph[i][j] = 0;
                    }
                }
            }
        } else {
            for(int i = 0; i < n; i++){
                for(int j = 0; j < n; j++){
                    graph[i][j] = 0;
                }
            }
            for(int i = 0; i < n-1; i++){
                graph[i][i+1] = 1;
            }
            graph[n-1][0] = 1;
        }
        counter = 0;
        warshall(n);
        if(c == 0)
            fprintf(f1, "%d\t%d\n", n, counter);
        else
            fprintf(f2, "%d\t%d\n", n, counter);
    }

    fclose(f1);
    fclose(f2);
}

int main(){
    for(int i = 0; i < 2; i++)
        ploter(i);

    printf("The graph is plotted\n");

    createGraph();
    counter = 0;
    warshall(n);

    printf("Applying Warshall's Algorithm\n");
    printf("Transitive Closure Matrix:\n");
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            printf("%d ", graph[i][j]);
        }
        printf("\n");
    }
    printf("Operation Count: %d\n", counter);

    return 0;
}
```

10b)

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

#define MAX 40
int graph[MAX][MAX], n, count = 0;

void createGraph() {
    printf("Number of vertices>> ");
    scanf("%d", &n);
    printf("Enter adjacency matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
}

void floydWarshall() {
    int temp;
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            temp = graph[i][k];
            for (int j = 0; j < n; j++) {
                if (graph[i][j] > temp) {
                    count++;
                    if (temp + graph[k][j] < graph[i][j])
                        graph[i][j] = temp + graph[k][j];
                }
            }
        }
    }
}

void printMatrix() {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", graph[i][j]);
        }
        printf("\n");
    }
}

void tester() {
    createGraph();
    floydWarshall();
    printf("Applying Floyd's algorithm\n");
    printf("All pair shortest path matrix:\n");
    printMatrix();
    printf("Operation count: %d\n", count);
}
```

```c
void createRandomGraph(int size) {
    n = size;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) graph[i][j] = 0;
            else graph[i][j] = rand() % 100; // Random weight between 0 and 99
        }
    }
}

void plotter() {
    FILE *fp = fopen("floyd_general.txt", "w");
    for (n = 2; n < 12; n++) {
        count = 0;
        createRandomGraph(n);
        floydWarshall();
        fprintf(fp, "%d\t%d\n", n, count);
    }
    fclose(fp);
    printf("The data for plotting has been written to 'floyd_general.txt'\n");
}

int main() {
    int choice;
    printf("Select mode: 1 for Tester, 2 for Plotter>> ");
    scanf("%d", &choice);
    switch (choice) {
        case 1:
            tester();
            break;
        case 2:
            plotter();
            break;
        default:
            printf("Invalid choice\n");
            break;
    }
    return 0;
}
```

11a) Knapsack (Bottom up)

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

#define MAX 100
int t[MAX][MAX], v[MAX], w[MAX], n, m, count = 0;

int max(int a, int b) {
    return (a > b) ? a : b;
}

int knap(int n, int m) {
    for (int i = 0; i < n + 1; i++) {
        for (int j = 0; j < m + 1; j++) {
            if (i == 0 || j == 0)
                t[i][j] = 0;
            else {
                count++;
                if (j < w[i])
                    t[i][j] = t[i - 1][j];
                else
                    t[i][j] = max(t[i - 1][j], v[i] + t[i - 1][j - w[i]]);
            }
        }
    }
    return t[n][m];
}

void printMatrix() {
    for (int i = 0; i < n + 1; i++) {
        for (int j = 0; j < m + 1; j++) {
            printf("%d ", t[i][j]);
        }
        printf("\n");
    }
}

void printComposition() {
    printf("Composition:\n");
    for (int i = n; i > 0; i--) {
        if (t[i][m] != t[i - 1][m]) {
            printf("%d\t", i);
            m = m - w[i];
        }
    }
    printf("\n");
}

void tester() {
    printf("Number of items: ");
    scanf("%d", &n);
    printf("Sack capacity: ");
    scanf("%d", &m);
    printf("Weight\tValue\n");
    for (int i = 1; i < n + 1; i++) {
        scanf("%d\t%d", &w[i], &v[i]);
    }
    printf("Max value %d\n", knap(n, m));
    printMatrix();
    printComposition();
    printf("Operation count: %d\n", count);
}

void plotter() {
    FILE *f1 = fopen("knapsackgraph.txt", "a");
    srand(time(NULL));

    for (int i = 1; i <= 10; i++) {
        int n = i * 2;   // Increase number of items
        int m = i * 5;   // Increase sack capacity

        count = 0;

        for (int i = 1; i < n + 1; i++) {
            w[i] = rand() % 20 + 1; // Random weight between 1 and 20
            v[i] = rand() % 100 + 1; // Random value between 1 and 100
        }

        knap(n, m);

        fprintf(f1, "%d\t%d\n", n, count);
    }

    fclose(f1);
}

int main() {
    int choice;
    printf("Select mode: 1 for Tester, 2 for Plotter>> ");
    scanf("%d", &choice);
    switch (choice) {
        case 1:
            tester();
            break;
        case 2:
            plotter();
            break;
        default:
            printf("Invalid choice\n");
            break;
    }
    return 0;
}
```

Knapsack(Memoization)

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

#define MAX 100
int t[MAX][MAX], v[MAX], w[MAX], n, m, count = 0;


int max(int a, int b) {
    return (a > b) ? a : b;
}

// Recursive function for the top-down approach with memoization
int knap(int i, int j) {

    // Base condition: No items or capacity is 0
    if (i == 0 || j == 0)
        return 0;

    // Check if the value is already computed
    if (t[i][j] != -1)
        return t[i][j];

    // If the weight of the current item is more than the remaining capacity, skip this
item
    if (w[i] > j)
        return t[i][j] = knap(i - 1, j);
    else{
        count++;
        return t[i][j] = max(knap(i - 1, j), v[i] + knap(i - 1, j - w[i]));
    }
}

// Function to print the matrix used in the top-down DP approach
void printMatrix() {
    for (int i = 0; i < n + 1; i++) {
        for (int j = 0; j < m + 1; j++) {
            printf("%d ", t[i][j]);
        }
        printf("\n");
    }
}

// Function to print the items included in the optimal solution
void printComposition() {
    printf("Composition:\n");
    int i = n, j = m;
    while (i > 0 && j > 0) {
        if (t[i][j] != t[i - 1][j]) {
            printf("%d\t", i);
            j -= w[i];
        }
        i--;
    }
    printf("\n");
}

// Function to run a single test case
void tester() {
    printf("Number of items: ");
    scanf("%d", &n);
    printf("Sack capacity: ");
    scanf("%d", &m);
    printf("Weight\tValue\n");
    for (int i = 1; i < n + 1; i++) {
        scanf("%d\t%d", &w[i], &v[i]);
    }

    // Initialize the memoization table with -1
    memset(t, -1, sizeof(t));

    // Calculate the maximum value
    printf("Max value: %d\n", knap(n, m));

    printMatrix();
    printComposition();
    printf("Operation count: %d\n", count);
}

void plotter() {
    FILE *f1 = fopen("knapsackgraph.txt", "a");

    for (int i = 1; i <= 10; i++) {
        count = 0;
        n = i; // Gradually increasing number of items
        m = i * 5; // Gradually increasing sack capacity

        // Initialize weights and values
        for (int j = 1; j <= n; j++) {
            w[j] = rand() % 20 + 1;
            v[j] = rand() % 100 + 1;
        }

        // Initialize the memoization table with -1
        memset(t, -1, sizeof(t));

        knap(n, m);
        //Actual: fprintf(f1, “%d\t%d\n”, n, count);
        fprintf(f1, "%d\t%d\n", n, n*m);

    }

    fclose(f1);
}
```

```c
// Main function to select between tester and plotter modes
int main() {
    int choice;
    printf("Select mode: 1 for Tester, 2 for Plotter>> ");
    scanf("%d", &choice);
    switch (choice) {
        case 1:
            tester();
            break;
        case 2:
            plotter();
            break;
        default:
            printf("Invalid choice\n");
            break;
    }
    return 0;
}
```

11b) Prim’s Algo:

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


// Global Variables
int n, i, j, cost[20][20], cnt = 0, visited[20], removed[20];
int heapsize = 0;
int heapcount, graphcount, max;

// Edge structure
struct edge {
    int v;
    int dist;
    int u;
} heap[20], ans[20];

typedef struct edge edg;


void swap(struct edge *a, struct edge *b) {
    struct edge temp = *a;
    *a = *b;
    *b = temp;
}

void heapify(struct edge arr[], int n, int i) {
    int smallest = i;
    int left = (2*i);
    int right = ((2*i)+1);
    heapcount++;
    if (left <= n && arr[left].dist < arr[smallest].dist)
        smallest = left;
    if (right <= n && arr[right].dist < arr[smallest].dist)
        smallest = right;
    if (smallest!= i) {
        swap(&arr[i], &arr[smallest]);
        heapify(arr, n, smallest);
    }
}


void makegraph() {
    printf("Enter the total number of vertices:");
```

```c
    scanf("%d", &n);
    printf("Enter the cost matrix of the Graph\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &cost[i][j]);
            if (cost[i][j] == 0)
                cost[i][j] = INT_MAX;
        }
    }
}

void createRandomGraph(int n) {
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            cost[i][j] = rand() % 5;
            if (cost[i][j] == 0)
                cost[i][j] = INT_MAX;
        }
    }
}


edg deleteheap(edg heap[]) {
    edg min = heap[1];
    heap[1] = heap[heapsize];
    heapsize = heapsize - 1;
    heapify(heap, heapsize, 1); // Restore heap order
    return min;
}

void insert(edg heap[], int start, int end, int dist){

    //Start represents parent, end represents where parent is pointing to.
    heapsize++;
    heap[heapsize].v = start;
    heap[heapsize].u = end;
    heap[heapsize].dist = dist;

    int index = heapsize;
    while(index > 1){
        int parent = index/2;
        if(heap[parent].dist > heap[index].dist){
            swap(&heap[parent],&heap[index]);
            index = parent;
        }
        else break;
    }
}


void prim(){

    //heap, ans, cost

    cnt = 0;
    heapsize = 0;
    int visited[n];
    for(int i = 0; i < n; i++) visited[i] = 0;

    insert(heap,-1,0,0);

    while(cnt != n){
        edg temp = deleteheap(heap);
        int wt = temp.dist;
        int start = temp.v;
        int end = temp.u;
        if(visited[end]) continue;

        if(end != 0){
            ans[cnt].dist = wt;
            ans[cnt].v = start;
            ans[cnt].u = end;
        }

        visited[end] = 1;

        for(int i = 0; i < n; i++){
            graphcount++;
            if(cost[end][i] != INT_MAX && !visited[i]){
                insert(heap,end,i,cost[end][i]);
            }
        }

        cnt++;
    }
}


void tester() {
    int sum = 0;
    makegraph();
    prim();
    for (int i = 1; i < cnt; i++) {
        printf("%c --> %c == %d\n", ans[i].v + 65, ans[i].u + 65,
ans[i].dist);
        sum += ans[i].dist;
    }
    printf("Minimum Distance is: %d\n", sum);
}


void plotter() {
    FILE *f1;
    f1 = fopen("primsgraph.txt", "a");
    int sum = 0;
    cnt = 0;
    heapsize = 0;
    heapcount = 0;
    graphcount = 0;
    max = 0;
    srand(time(NULL));

    for(int i = 4; i <= 8; i++){
        n = i;
        createRandomGraph(i);
        prim();
        for (int j = 1; j <= i; j++)
            sum += ans[j].dist;
        max = (graphcount < heapcount) ? graphcount: heapcount;
        fprintf(f1, "%d\t%d\n", n, max);
    }

}


void main() {
    int ch;
    while (1) {
        printf("Enter choice 1 to run tester, 2 to run plotter, and 0 to
exit\n");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                tester();
                break;
            case 2:
                plotter();
                break;
            default:
                exit(0);
        }
    }
}
```

12: Dijkstra’s Algorithm:

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

int n, i, j, src, heapsize = 0, cost[10][10], d[10];
int graphcount = 0, heapcount = 0, max = 0;

struct edge {
    int id;
    int dist;
} heap[10];

typedef struct edge edg;

void swap(struct edge *a, struct edge *b) {
    struct edge temp = *a;
    *a = *b;
    *b = temp;
}

void heapify(struct edge arr[], int n, int i) {
    int smallest = i;
    int left = (2*i);
    int right = ((2*i)+1);
    heapcount++;
    if (left <= n && arr[left].dist < arr[smallest].dist)
        smallest = left;
    if (right <= n && arr[right].dist < arr[smallest].dist)
        smallest = right;
    if (smallest != i) {
        swap(&arr[i], &arr[smallest]);
        heapify(arr, n, smallest);
    }
}

void makegraph() {

    printf("Enter the total number of vertices: ");
    scanf("%d", &n);
    printf("Enter the cost matrix of the Graph\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &cost[i][j]);
            if (cost[i][j] == 0)
                cost[i][j] = INT_MAX;
        }
    }
    // Initialise the source vertex distance to 0 and rest all to infinity(INT_MAX)
    printf("Enter the source vertex: ");
    scanf("%d", &src);
    for (i = 0; i < n; i++) {
        d[i] = INT_MAX;
    }
    d[src] = 0;
}


void createRandomGraph(int n) {
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            cost[i][j] = rand() % 5;
            if (cost[i][j] == 0)
                cost[i][j] = INT_MAX;
        }
    }

    src = 0;
    for (i = 0; i < n; i++) {
        d[i] = INT_MAX;
    }
    d[src] = 0;
}


edg deleteheap(edg heap[]) {
    edg min = heap[1];
    heap[1] = heap[heapsize];
    heapsize = heapsize - 1;
    heapify(heap, heapsize, 1); // Restore heap order
    return min;
}

void insert(edg heap[], int node, int dist){

    heapsize++;
    heap[heapsize].id = node;
    heap[heapsize].dist = dist;

    int index = heapsize;
    while(index > 1){
        int parent = index/2;
        if(heap[parent].dist > heap[index].dist){
            swap(&heap[parent],&heap[index]);
            index = parent;
        }
        else break;
    }
}

void dijkstra(){
    heapsize = 0;
    insert(heap,src,0);
    d[src] = 0;

    while(heapsize != 0){
        int dist = heap[1].dist;
        int node = heap[1].id;
        deleteheap(heap);

        for(int i = 0; i < n; i++){
            graphcount++;
            if(cost[node][i] != INT_MAX){                   //Extremely important, cuz not
writing it can lead to integer overflow to negative values in dist+cost[node][i] line, thereby
giving very low distances.
                int newDist = dist + cost[node][i];
                int newNode = i;

                if(d[newNode] > newDist){
                    d[newNode] = newDist;
                    insert(heap,newNode,newDist);
                }
            }
        }
    }
```

```c
}

void tester() {
    makegraph();
    dijkstra();
    printf("Shortest path from source %d is:\n", src);
    for (i = 0; i < n; i++) {
        if (src != i)
            printf("%d -> %d = %d\n", src, i, d[i]);
    }
}



void plotter() {
    FILE *f1;
    f1 = fopen("dijkstrasgraph.txt", "a");
    srand(time(NULL));

    for(int i = 4; i <= 8; i++){
        createRandomGraph(i);
        n = i;
        max = 0;
        graphcount = 0;
        heapcount = 0;
        heapsize = 0;
        dijkstra();
        max = (graphcount > heapcount) ? graphcount : heapcount;
        fprintf(f1, "%d\t%d\n", n, max);
    }

}

int main() {
    int choice;
    printf("1.Tester \n2.Plotter \n3.Exit\n ");
    while (1) {
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                tester();
                break;
            case 2:
                plotter();
                break;
            case 3:
                exit(0);
        }
    }
    return 0;
}
```