

DAY1

QUICK SORT

1. Int random_partition(int * arr, int left, int right)
2. Int ramdom_quicksort(int * arr, int left, int right)
3. All_permutation_seq(int * arr, int n)
4. Run quick sort on each input sequence from n factorial set of elements and generate an output file out.txt
5. Count the no of comparisons and store in same out.txt file
6. Integrate all functions call through main()

Algorithm

RANDOMIZED-PARTITION(A, p, r)

```
1   $i = \text{RANDOM}(p, r)$ 
2  exchange  $A[r]$  with  $A[i]$ 
3  return PARTITION( $A, p, r$ )
```

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

RANDOMIZED-QUICKSORT(A, p, r)

```
1  if  $p < r$ 
2       $q = \text{RANDOMIZED-PARTITION}(A, p, r)$ 
3      RANDOMIZED-QUICKSORT( $A, p, q - 1$ )
4      RANDOMIZED-QUICKSORT( $A, q + 1, r$ )
```

CODE

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

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

```
//#include <time.h>
```

```
FILE *fptr;
```

```
FILE *fptr2;
```

```
int count,counter;
```

```
void swap(int *arr,int left,int right){
```

```
    int temp=arr[left];
```

```

        arr[left]=arr[right];

        arr[right]=temp;
    }

int All_permutation_generation(int*arr,int left , int right){

    int i;

    if (left==right){

        for(i=0;i<=right;i++)

            fprintf(fptr,"%d ",arr[i]);

        fprintf(fptr,"\n");

        counter++;

    }

    else{

        for(i=left;i<=right;i++)

        {

            swap(arr,left,i);

            All_permutation_generation(arr,left+1 , right);

            swap(arr,left,i);

        }

    }

}

int randomized_partition(int *arr,int left,int right){

    int high=right;

    int low=left;

    int i,j;

```

```

//srand(time(0));

int pivot,random_index;

random_index=left + rand()%(high-low+1);

swap(arr,random_index,left);

pivot=arr[left];

i=low;

for(j=low+1;j<=high;j++){

    count++;

    if(arr[j]<=pivot){

        i++;

        swap(arr,i,j);

    }

}

swap(arr,left,i);

return i;

}

void rqs(int *arr,int left,int right)

{

    if(left<right){

        int p=randomized_partition(arr,left,right);

        rqs(arr,left,p-1);

        rqs(arr,p+1,right);

    }

}

```

```

    }
}

void outputgenration(int *arr,int n){

    int i,j=0;

    float sum;

    while(!feof(fptr)){

        for(i=0;i<n;i++){

            fscanf(fptr,"%d",&arr[i]);

            count=0;

            rqs(arr,0,n-1);

            if(j<counter){

                for(i=0;i<n;i++)

                    fprintf(fp2,"%d ",arr[i]);

                fprintf(fp2,"count: %d\n",count);

                sum+=count;

                j++;

            }

        }

        printf("\naverage number of comparisions requires is %f ",sum/counter);

    }

}

int main(){

    int i,n;

    int *arr;

    fptr=fopen("inputsequence.txt","w");

    printf("enter the sequence of numbers");

```

```

scanf("%d",&n);

arr=(int*)malloc(n*sizeof(int));

for(i=0;i<n;i++)

    arr[i]=i+1;

counter=0;

All_permutation_generation(arr,0,n-1);

fclose(fp1);

fp1=fopen("inputsequence.txt","r");

fp2=fopen("outputsequence.txt","w");

outputgeneration(arr,n);

fclose(fp2);

fclose(fp1);

return 0;

}

```

OUTPUT

```

enter the sequence of numbers3
average number of comparisions requires is 2.666667


```

Open ▾  **inputsequence.txt**
~/04_jyotiraditya

```

1 1 2 3
2 1 3 2
3 2 1 3
4 2 3 1
5 3 2 1
6 3 1 2

```

Open ▾  **outputsequence.txt**
~/04_jyotiraditya

```

1 1 2 3 count: 2
2 2 2 3 count: 2
3 2 3 3 count: 3
4 1 3 3 count: 3
5 3 3 3 count: 3
6 2 3 3 count: 3

```

DAY2

MERGE SORT

1. Merge_sort(int * arr, int left, int right)
2. Merging(int * arr, int left, int mid,int right)
3. Run_mergesort_all_permutation(int * arr, int n)
4. Integrate through main()

Also compute no of comparisons and space complexity

Algorithm

```
MERGE_FUNCTION (arr, low, mid, high)
1. Let n1  $\leftarrow$  (mid - low) + 1
2. Let n2  $\leftarrow$  high - mid
3. Let arr1[1...(n1 + 1)] & arr  $\leftarrow$  2[1...(n2 + 1)] be
   the new arrays
4. for i  $\leftarrow$  1 to n1
5. arr1[i]  $\leftarrow$  arr[low + i - 1]
6. for j  $\leftarrow$  1 to n2
7. arr2[j]  $\leftarrow$  arr[mid + j]
8. arr1[n1 + 1]  $\leftarrow$   $\infty$ 
9. arr2[n2 + 1]  $\leftarrow$   $\infty$ 
10. i  $\leftarrow$  1
11. j  $\leftarrow$  1
12. for k  $\leftarrow$  low to high
13. if arr1[i]  $\leq$  arr2[j]
14. arr[k]  $\leftarrow$  arr1[i]
15. i  $\leftarrow$  i + 1
16. else arr[k] = arr2[j]
17. j  $\leftarrow$  j + 1
```

CODE

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

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

```
FILE *fptr;
```

```
FILE *fptr2;
```

```
int count,counter;
```

```
int count2=0,count3=0;
```

```

void swap(int *arr,int left,int right){

    int temp=arr[left];

    arr[left]=arr[right];

    arr[right]=temp;

}

int All_permutation_generation(int*arr,int left , int right){

    int i;

    if (left==right){

        for(i=0;i<=right;i++)

            fprintf(fptr,"%d\t",arr[i]);

        fprintf(fptr,"\n");

        counter++;

    }

    else{

        for(i=left;i<=right;i++)

        {

            swap(arr,left,i);

            All_permutation_generation(arr,left+1 , right);

            swap(arr,left,i);

        }

    }

}

void merge(int *arr, int l, int m, int r)

{

```

```

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int *L, *R;

L=(int*)malloc(n1*sizeof(int));

R=(int*)malloc(n2*sizeof(int));

for (i = 0; i < n1; i++)

    L[i] = arr[l + i];

for (j = 0; j < n2; j++)

    R[j] = arr[m + 1 + j];

i = 0;

j = 0;


// Initial index of merged subarray

k = l;

while (i < n1 && j < n2)

{

    if (L[i] <= R[j])

    {

        arr[k] = L[i];

        i++;

        count2++;

    }

    else

    {

```



```
        arr[k] = R[j];

        j++;

        count2++;

    }

    k++;

}

    count3+=n1+n2;

// Copy the remaining elements
// of L[], if there are any
while (i < n1) {

    arr[k] = L[i];

    i++;

    k++;

}


// Copy the remaining elements of
// R[], if there are any
while (j < n2)

{

    arr[k] = R[j];

    j++;

    k++;

}

}
```

```

void rqs(int *arr,int left,int right)
{
    if(left<right){
        int mid=(left+right)/2;
        rqs(arr,left,mid);
        rqs(arr,mid+1,right);
        merge(arr,left,mid,right);
    }
}

void outputgenration(int *arr,int n){
    int i,j=0;
    float sum;
    while(!feof(fptr)){
        for(i=0;i<n;i++)
            fscanf(fptr,"%d",&arr[i]);

        //count=0;
        rqs(arr,0,n-1);
        if(j<counter){
            for(i=0;i<n;i++)
                fprintf(fp2,"%d\t",arr[i]);

            fprintf(fp2,"\n");

            //sum+=count;

            j++;
        }
    }
}

```

```
}
```

```
}
```

```
int main(){  
    int i,n;  
    int *arr;  
    fptr=fopen("inputsequence.txt","w");  
    printf("enter the sequence of numbers");  
    scanf("%d",&n);  
    arr=(int*)malloc(n*sizeof(int));  
    for(i=0;i<n;i++)  
        arr[i]=i+1;  
    counter=0;  
    All_permutation_generation(arr,0,n-1);  
    fclose(fptr);  
    fptr=fopen("inputsequence.txt","r");  
    fptr2=fopen("outputsequence.txt","w");  
    outputgenration(arr,n);  
    fprintf(fptr2,"time: %d\n",count2);  
    fprintf(fptr2,"space: %d\n",count3);  
    fclose(fptr2);  
    fclose(fptr);  
    return 0;  
}
```

OUTPUT

Enter the sequence of numbers 3

	Open	+	inputsequence.txt
			~/04_jyotiraditya
1	1	2	3
2	1	3	2
3	2	1	3
4	2	3	1
5	3	2	1
6	3	1	2

	Open	+	outputsequence.txt
			~/04_jyotiraditya
1	1	2	3
2	1	2	3
3	1	2	3
4	1	2	3
5	1	2	3
6	1	2	3
7	time:	19	
8	space:	35	

HEAP SORT

1. Heap_sort(int *arr, int size)
2. Buildheap((int *arr, int size)
3. Maxheapify((int *arr, int root_index,int size)
4. Run_heapsort_all_permutation(int * arr, int n)
5. Integrate all functions call through main()

Algorithm

MAX-HEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4       $\text{largest} = l$ 
5  else  $\text{largest} = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7       $\text{largest} = r$ 
8  if  $\text{largest} \neq i$ 
9      exchange  $A[i]$  with  $A[\text{largest}]$ 
10     MAX-HEAPIFY( $A, \text{largest}$ )
```

BUILD-MAX-HEAP(A)

```
1   $A.\text{heap-size} = A.\text{length}$ 
2  for  $i = \lfloor A.\text{length}/2 \rfloor$  downto 1
3      MAX-HEAPIFY( $A, i$ )
```

HEAPSORT(A)

```
1  BUILD-MAX-HEAP( $A$ )
2  for  $i = A.\text{length}$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.\text{heap-size} = A.\text{heap-size} - 1$ 
5      MAX-HEAPIFY( $A, 1$ )
```

CODE

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

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

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

```
#include <ctype.h>

int count = 0;

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

    *a = *b;

    *b = temp;
}

void maxheapify(int *arr, int root_index, int size)
{
    int left_index = 2 * root_index + 1;

    int right_index = 2 * root_index + 2;

    int largest_index = root_index;

    if(left_index < size && arr[left_index] > arr[largest_index])
    {
        largest_index = left_index;

        count++;
    }

    if(right_index < size && arr[right_index] > arr[largest_index])
    {
        largest_index = right_index;

        count++;
    }

    if(largest_index != root_index)
    {
```

```
        swap(&arr[largest_index], &arr[root_index]);  
        maxheapify(arr, largest_index, size);  
    }  
}
```

```
void buildheap(int *arr, int size)  
{  
    int parent_index = (int)(size - 2)/2;  
    while(parent_index >= 0)  
    {  
        maxheapify(arr, parent_index, size);  
        parent_index--;  
    }  
}
```

```
void heapsort(int *arr, int size)  
{  
    buildheap(arr, size);  
    while(size > 1)  
    {  
        swap(&arr[0], &arr[size - 1]);  
        size--;  
        maxheapify(arr, 0, size);  
    }  
}
```

```

void permutation(int *arr, int l, int r)
{
    FILE *fptr = fopen("heapsort_permutation.txt", "a");

    if(l == r)
    {
        for (int i = 0; i <= r; i++)

            fprintf(fptr, "%d\t", arr[i]);

        fprintf(fptr, "\n");
    }
    else
    {
        for(int i = l; i <= r; i++)
        {
            swap(&arr[i], &arr[l]);

            permutation(arr, l+1, r);

            swap(&arr[i], &arr[l]);
        }
    }

    fclose(fptr);
}

void heapsort_run(int n)
{
    FILE *fptr = fopen("heapsort_permutation.txt", "r"), *fout = fopen("heapsort_out.txt", "w");

    char *c = (char *)malloc(sizeof(char) * (2*n + 2)), *token;

```



```

int *a = (int *)malloc(sizeof(int) * n), i = 0;

while(fgets(c, (2*n+2), fptr))
{
    token = strtok(c, "\t\n");

    while(token != NULL)
    {
        *(a+i) = atoi(token);

        token = strtok(NULL, "\t\n");

        i++;
    }

    heapsort(a, n);

    for(i = 0; i < n; i++)
        fprintf(fout, "%d\t", a[i]);

    fprintf(fout, "%d\n", count);

    i = 0;

    count = 0;
}

fclose(fout);

fclose(fptr);
}

int main()
{
    FILE *fptr = fopen("heapsort_permutation.txt", "w");

    fclose(fptr);

    int n, *arr;

```

```

printf("Enter size of array : ");

scanf("%d", &n);

arr = (int *)malloc(sizeof(int) * n);

printf("Enter elements into the array :\n");

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

    scanf("%d", &*(arr+i));

permutation(arr, 0, n-1);

heapsort_run(n);

return 0;

}

```

OUTPUT

```

Enter size of array : 4
Enter elements into the array :
3
7
1
6

```

heapsort_permutation.txt				
~/04_jyotiraditya				
1	3	7	1	6
2	3	7	6	1
3	3	1	7	6
4	3	1	6	7
5	3	6	1	7
6	3	6	7	1
7	7	3	1	6
8	7	3	6	1
9	7	1	3	6
10	7	1	6	3
11	7	6	1	3
12	7	6	3	1
13	1	7	3	6
14	1	7	6	3
15	1	3	7	6
16	1	3	6	7
17	1	6	3	7
18	1	6	7	3
19	6	7	1	3
20	6	7	3	1
21	6	1	7	3
22	6	1	3	7
23	6	3	1	7
24	6	3	7	1

heapsort_out.txt				
1	1	3	6	7
2	1	3	6	7
3	1	3	6	7
4	1	3	6	7
5	1	3	6	7
6	1	3	6	7
7	1	3	6	7
8	1	3	6	7
9	1	3	6	7
10	1	3	6	7
11	1	3	6	7
12	1	3	6	7
13	1	3	6	7
14	1	3	6	7
15	1	3	6	7
16	1	3	6	7
17	1	3	6	7
18	1	3	6	7
19	1	3	6	7
20	1	3	6	7
21	1	3	6	7
22	1	3	6	7
23	1	3	6	7
24	1	3	6	7

Day 3

Adjacency list, adjacency matrix

Code

```
include <stdio.h>

#include <stdlib.h>

FILE *fp;

typedef struct Node{

    int vertex;

    struct node *next;

    int weight;

}node;

void am(int n,int flag){

    int i,j,vi,vj,w;

    int *m=(int*)malloc(n*sizeof(int));

    for(i=0;i<n;i++)

        m[i]=(int*)malloc(n*sizeof(int));

    for(i=0;i<n;i++)

        for(j=0;j<n;j++)

            m[i][j]=0;

    while(!feof(fp)){

        fscanf(fp,"%d",&vi);

        fscanf(fp,"%d",&vj);

        fscanf(fp,"%d",&w);

        m[vi][vj]=1;

        if(flag==0)

            m[vj][vi]=1;
```

```

    }

    printf("\nthe matrix is\n");

    for(i=0;i<n;i++){

        printf("\n");

        for(j=0;j<n;j++)

            printf("%d\t",m[i][j]);

    }

}

void insertBeg(node **head, int ver, int wt)

{

    node *newnode=(node*)malloc(sizeof(node));

    newnode->vertex = ver;

    newnode->weight=wt;

    newnode->next = *head;

    *head = newnode;

}

void createlist(int n,int type){

    node **p=(node**)malloc(n*sizeof(node*));

    int u,v,w,i;

    for(i=0;i<n;i++){

        p[i]=(node *)malloc(sizeof(node));

        p[i]=NULL;}

    FILE *fptr = fopen("graph.txt", "r");

    while(!feof(fptr))

        {

```

```

        fscanf(fptr, "%d %d %d\n", &u, &v, &w);

        insertBeg(&p[u], v, w);

        if(type == 0)

            insertBeg(&p[v], u, w);

    }

    fclose(fptr);

    printf("\nadjacency list:\n");

    for(i=0;i<n;i++){

        printf("\n%d",i);

        node*ptr=p[i];

        while(ptr!=NULL){

            printf("->%d-%d",ptr->vertex,ptr->weight);

            ptr=ptr->next;

        }

    }

    printf("\n");

}

int main(){

    int choice;

    printf("enter 0 for directed graph and 1 for undirected graph");

    scanf("%d",&choice);

    fp=fopen("graph.txt","r");

    am(5,choice);

    createlist(5,choice);

    fclose(fp);

```

```
        return 0;  
    }  
}
```

OUTPUT

```
enter 0 for directed graph and 1 for undirected graph0  
the matrix is  
0      1      0      0      0  
1      0      1      1      0  
0      1      0      1      1  
0      1      1      0      1  
0      0      1      1      0  
adjacency list:  
0->1-9  
1->3-7->0-9->2-4  
2->4-6->3-5->1-4  
3->1-7->4-2->2-5  
4->2-6->3-2
```

kth rank

Algorithm

```
RANDOMIZED-SELECT( $A, p, r, i$ )
1  if  $p == r$ 
2      return  $A[p]$ 
3   $q = \text{RANDOMIZED-PARTITION}(A, p, r)$ 
4   $k = q - p + 1$ 
5  if  $i == k$            // the pivot value is the answer
6      return  $A[q]$ 
7  elseif  $i < k$ 
8      return RANDOMIZED-SELECT( $A, p, q - 1, i$ )
9  else return RANDOMIZED-SELECT( $A, q + 1, r, i - k$ )
```

CODE

```
#include <stdio.h>

#include <stdlib.h>

FILE *fptr;

FILE *fptr2;

int counter;

void swap(int *arr,int left,int right){

    int temp=arr[left];

    arr[left]=arr[right];

    arr[right]=temp;

}

int All_permutation_generation(int*arr,int left , int right){

    int i;

    if (left==right){

        for(i=0;i<=right;i++)

            fprintf(fptr,"%d ",arr[i]);
```

```

        fprintf(fp_ptr, "\n");

        counter++;

    }

    else{

        for(i=left; i<=right; i++)

        {

            swap(arr, left, i);

            All_permutation_generation(arr, left+1 , right);

            swap(arr, left, i);

        }

    }

}

int randomized_partition(int *arr, int left, int right){

    int high=right;

    int low=left;

    int i, j;

    //srand(time(0));

    int pivot, random_index;

    random_index=left + rand()%(high-low+1);

    swap(arr, random_index, left);

    pivot=arr[left];

    i=low;

    for(j=low+1; j<=high; j++){

        if(arr[j]<=pivot){

```



```

        i++;
        swap(arr,i,j);
    }
}

swap(arr,left,i);

return i;
}

int random_selection(int* arr, int start, int end, int k)
{
    if(start == end)
        return arr[start];

    if(k == 0) return -1;

    if(start < end)
    {

        int mid = randomized_partition(arr, start, end);

        int i = mid - start + 1;

        if(i == k)
            return arr[mid];

        else if(k < i)
            return random_selection(arr, start, mid-1, k);

        else
            return random_selection(arr, mid+1, end, k-i);
    }
}

```

```
}
```

```
}
```

```
void outputgenration(int *arr,int n,int rank){  
    int i,j=0,k;  
    while(!feof(fptr)){  
        for(i=0;i<n;i++){  
            fscanf(fptr,"%d",&arr[i]);  
            fprintf(fp2,"%d ",arr[i]);  
        }  
        k=random_selection(arr,0,n-1,rank);  
        fprintf(fp2,"value: %d\n",k);  
    }  
}
```

```
int main(){  
    int i,n,rank;  
    int *arr;  
    fptr=fopen("inputsequence.txt","w");  
    printf("enter the sequence of numbers");  
    scanf("%d",&n);  
    arr=(int*)malloc(n*sizeof(int));  
    for(i=0;i<n;i++)  
        arr[i]=i+1;  
    counter=0;
```

```

All_permutation_generation(arr,0,n-1);

fclose(fptr);

fptr=fopen("inputsequence.txt","r");

fptr2=fopen("outputsequence.txt","w");

printf("\nenter rank");

scanf("%d",&rank);

outputgenration(arr,n,rank);

fclose(fptr2);

fclose(fptr);

return 0;

}


```

OUTPUT

```

enter the sequence of numbers4
enter rank2


```

Open ▾  inputsequence.txt
~/04_jyotiraditya

```

1 1 2 3 4
2 1 2 4 3
3 1 3 2 4
4 1 3 4 2
5 1 4 3 2
6 1 4 2 3
7 2 1 3 4
8 2 1 4 3
9 2 3 1 4
10 2 3 4 1
11 2 4 3 1
12 2 4 1 3
13 3 2 1 4
14 3 2 4 1
15 3 1 2 4
16 3 1 4 2
17 3 4 1 2
18 3 4 2 1
19 4 2 3 1
20 4 2 1 3
21 4 3 2 1
22 4 3 1 2
23 4 1 3 2
24 4 1 2 3

```

Open ▾  *outputsequence.txt
~/04_jyotiraditya

```

1 1 2 3 4 value: 2
2 1 2 4 3 value: 2
3 1 3 2 4 value: 2
4 1 3 4 2 value: 2
5 1 4 3 2 value: 2
6 1 4 2 3 value: 2
7 2 1 3 4 value: 2
8 2 1 4 3 value: 2
9 2 3 1 4 value: 2
10 2 3 4 1 value: 2
11 2 4 3 1 value: 2
12 2 4 1 3 value: 2
13 3 2 1 4 value: 2
14 3 2 4 1 value: 2
15 3 1 2 4 value: 2
16 3 1 4 2 value: 2
17 3 4 1 2 value: 2
18 3 4 2 1 value: 2
19 4 2 3 1 value: 2
20 4 2 1 3 value: 2
21 4 3 2 1 value: 2
22 4 3 1 2 value: 2
23 4 1 3 2 value: 2
24 4 1 2 3 value: 2

```

Day 4

Matrix chain multiplication

1. void Optimum_Parenthesisation_Matrix_Chain (int ** mcount, int **split,int *matrix_dimention, int chain_size).
2. void Display_Optimum_Parenthesisation (int ** mcount, int **split, int chain_size).
3. int **multiply_matrixchain_optimally (int ** mcount, int **split, int chain_size).
4. int **create_and_input_matrix (int row, int col).
5. Display_matrix (int **mat, int row, int col).
6. Integrated all functions through main().

6x5=30

ALGORITHM

MATRIX-CHAIN-ORDER(p)

```
1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  and  $s[1..n-1, 2..n]$  be new tables
3  for  $i = 1$  to  $n$ 
4       $m[i, i] = 0$ 
5  for  $l = 2$  to  $n$            //  $l$  is the chain length
6      for  $i = 1$  to  $n - l + 1$ 
7           $j = i + l - 1$ 
8           $m[i, j] = \infty$ 
9          for  $k = i$  to  $j - 1$ 
10              $q = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
11             if  $q < m[i, j]$ 
12                  $m[i, j] = q$ 
13                  $s[i, j] = k$ 
14  return  $m$  and  $s$ 
```

PRINT-OPTIMAL-PARENS(s, i, j)

```
1  if  $i == j$ 
2      print " $A_i$ "
3  else print "("
4      PRINT-OPTIMAL-PARENS( $s, i, s[i, j]$ )
5      PRINT-OPTIMAL-PARENS( $s, s[i, j] + 1, j$ )
6      print ")"
```

CODE

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// Function to multiply two matrices

int **matrixmultiply(int **mat1, int **mat2, int r1, int r2, int c2)

{

    int i, j, k;

    int **result = (int **)malloc(r1 * sizeof(int *));

    for (i = 0; i < r1; i++)

        result[i] = (int *)malloc(c2 * sizeof(int));

    for (i = 0; i < r1; i++) {

        for (j = 0; j < c2; j++) {

            result[i][j] = 0;

            for (k = 0; k < r2; k++) {

                result[i][j] += mat1[i][k] * mat2[k][j];

            }

        }

    }

    return result;

}

// Function to create and input a matrix

int **create_and_input_matrix(int row, int col)
```

```

{
    int i, j;

    //time_t t;

    //srand((unsigned)time(&t));

    int **mat = (int **)malloc(row * sizeof(int *));

    for (i = 0; i < row; i++)

        mat[i] = (int *)malloc(col * sizeof(int));

    for (i = 0; i < row; i++) {

        for (j = 0; j < col; j++) {

            mat[i][j]=rand()%10;

        }

    }

    return mat;
}

```

```

void display_matrix(int **mat, int row, int col)

{
    int i, j;

    printf("Matrix:\n");

    for (i = 0; i < row; i++) {

        for (j = 0; j < col; j++) {

            printf("%d ", mat[i][j]);

        }

        printf("\n");

    }
}

```

```
}
```

```
// Function to calculate optimal parenthesization of matrix chain
```

```
void optimum_parenthesization_matrix_chain(int **mcount, int **split, int *matrix_dimension, int chain_size)
```

```
{
```

```
    int i, j, k, l, q;
```

```
    for (i = 1; i <= chain_size; i++)
```

```
        mcount[i][i] = 0;
```

```
    for (l = 2; l <= chain_size; l++) {
```

```
        for (i = 1; i <= chain_size - l + 1; i++) {
```

```
            j = i + l - 1;
```

```
            mcount[i][j] = INT_MAX;
```

```
            for (k = i; k <= j - 1; k++) {
```

```
                q = mcount[i][k] + mcount[k + 1][j] + matrix_dimension[i - 1] * matrix_dimension[k] *  
matrix_dimension[j];
```

```
                if (q < mcount[i][j]) {
```

```
                    mcount[i][j] = q;
```

```
                    split[i][j] = k;
```

```
                }
```

```
            }
```

```
        }
```

```
    }
```

```
}
```

```
void display_optimization_parenthesization(int **mcount, int **split, int i, int chain_size)
```

```

{
    if (i == chain_size) {
        printf(" A%d ", i);
    }
    else {
        printf("(");
        display_optimization_parenthesization(mcount, split, i, split[i][chain_size]);
        display_optimization_parenthesization(mcount, split, split[i][chain_size] + 1, chain_size);
        printf(")");
    }
}

// Function to multiply matrix chain optimally

int **multiply_matrixchain_optimally(int **mcount, int **split, int *matrix_dimension, int ***chain, int
chain_size, int left, int right)
{
    if (left == right)
        return chain[left];

    else
    {
        int i;

        int **left_matrix = multiply_matrixchain_optimally(mcount, split, matrix_dimension, chain,
chain_size, left, split[left][right]);

        int **right_matrix = multiply_matrixchain_optimally(mcount, split, matrix_dimension, chain,
chain_size, split[left][right] + 1, right);

        int **result = (int **)malloc(matrix_dimension[left-1] * sizeof(int *));

```



```

    for (i = 0; i < matrix_dimension[left-1]; i++) {

        result[i] = (int *)malloc(matrix_dimension[right] * sizeof(int));

    }

    result=matrixmultiply(left_matrix, right_matrix, matrix_dimension[left-1],
matrix_dimension[split[left][right]], matrix_dimension[right]);

    *chain = result;

    return result;

}

}

int main() {

    int chain_size,*matrix_dimension,i;

    printf("Enter chain size:");

    scanf("%d",&chain_size);

    matrix_dimension=malloc(sizeof(int));

    printf("Enter the dimensions of the matrices:");

    for(i=0;i<=chain_size;i++)

        scanf("%d",&matrix_dimension[i]);

    int **mcount = (int **)malloc((chain_size+1) * sizeof(int *));

    int **split = (int **)malloc((chain_size+1) * sizeof(int *));

    int ***chain = (int ***)malloc((chain_size+1) * sizeof(int **));

    int j;

    for (i = 0; i <= chain_size; i++) {

        mcount[i] = (int *)malloc((chain_size+1) * sizeof(int));

        split[i] = (int *)malloc((chain_size+1) * sizeof(int));

        chain[i] = (int **)malloc((chain_size+1) * sizeof(int *));

    }

```

```

for (i = 1; i <= chain_size; i++)

    chain[i] = create_and_input_matrix(matrix_dimension[i-1], matrix_dimension[i]);

optimum_parenthesization_matrix_chain(mcount, split, matrix_dimension, chain_size);

printf("Minimum number of multiplications: %d\n", mcount[1][chain_size]);

printf("Optimal Parenthesization: ");

display_optimization_parenthesization(mcount, split, 1, chain_size);

printf("\n");

int **result = multiply_matrixchain_optimally(mcount, split, matrix_dimension, chain, chain_size, 1,
chain_size);

printf("Resulting matrix:\n");

display_matrix(result, matrix_dimension[0], matrix_dimension[chain_size]);

return 0;

}

```

OUTPUT

```

Enter chain size:5
Enter the dimensions of the matrices:6
7
8
3
4
3
Minimum number of multiplications: 384
Optimal Parenthesization: (( A1 ( A2 A3 ))( A4 A5 ))
Resulting matrix:
Matrix:
2319283 1522747 1615796
1800440 1178968 1253607
1557538 1024816 1089002
1915893 1237957 1334464
2843572 1868144 1979675
2154217 1420305 1502061

```

Day 5

Belman Ford and Floyd Warshall

Implement the Bellman Ford's algorithm to find the shortest path from a given (input) source vertex to all other vertices in the graph, using adjacency list representation of graph. Follow the following function prototype in implementations:

- a) void Initialize_Single_Source_Shortest_Path(int sourceVertex, int vertexCount, int distance[], int predecessor[]) (5)
- b) void BellmanFord_ShortestPath(int sourceVertex, struct node ** graph, int vertexCount, int distance[], int predecessor[]) (10)
- c) void Display_Path_and_distance (int sourceVertex, int vertexCount, int distance[], int predecessor[]) (5)
- d) Integrate all functions through main() along with older functions create_adjacency_ListGraph and display_adjacencyList, already given in previous assignment. (10)

[N.B. Marks will be not be given on execution of program only. Marks will be given, on basis of the fact that whether the student will be able to explain the executed code satisfactorily.]

ALGORITHM

INITIALIZE-SINGLE-SOURCE(G, s)

```
1  for each vertex  $v \in G.V$ 
2       $v.d = \infty$ 
3       $v.\pi = \text{NIL}$ 
4   $s.d = 0$ 
```

RELAX(u, v, w)

```
1  if  $v.d > u.d + w(u, v)$ 
2       $v.d = u.d + w(u, v)$ 
3       $v.\pi = u$ 
```

BELLMAN-FORD(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  for  $i = 1$  to  $|G.V| - 1$ 
3      for each edge  $(u, v) \in G.E$ 
4          RELAX( $u, v, w$ )
5  for each edge  $(u, v) \in G.E$ 
6      if  $v.d > u.d + w(u, v)$ 
7          return FALSE
8  return TRUE
```

CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
FILE *fptr;
int arr[3];
```

```
typedef struct node
```

```
{  
    int weight;  
    int vertex;  
    struct node *next;  
} node;
```

```
node *insertAtBeg(node *head, int vertex, int wt)
```

```
{  
    node *newnode = (node *)malloc(sizeof(node));  
    newnode->weight = wt;  
    newnode->vertex = vertex;
```

```
    if (!head)
```

```
    {  
        head = newnode;  
        newnode->next = NULL;  
    }
```

```
    else
```

```
    {  
        newnode->next = head;  
        head = newnode;  
    }
```

```
    return head;
```

```
}
```

```
void displayList(node *head)
```

```
{  
    node *ptr = head;  
    while (ptr)  
    {  
        if (!ptr->next)  
        {  
            printf("[%d,%d]", ptr->vertex, ptr->weight);  
        }  
        else  
        {  
            printf("[%d,%d]->", ptr->vertex, ptr->weight);  
        }  
        ptr = ptr->next;
```

```
}  
}
```

```
node **createAdjacencyList(int n, int type)  
{  
    node **p = (node **)malloc(n * sizeof(node *));  
    int i;  
    for (i = 0; i < n; i++)  
    {  
        p[i] = NULL;  
    }  
    if (type == 1)  
    {  
        while (!feof(fptr))  
        {  
            for (i = 0; i < 3; i++)  
            {  
                fscanf(fptr, "%d ", &arr[i]);  
            }  
            p[arr[0]] = insertAtBeg(p[arr[0]], arr[1], arr[2]);  
        }  
    }  
    else  
    {  
        while (!feof(fptr))  
        {  
            for (i = 0; i < 3; i++)  
            {  
                fscanf(fptr, "%d ", &arr[i]);  
            }  
            p[arr[0]] = insertAtBeg(p[arr[0]], arr[1], arr[2]);  
            p[arr[1]] = insertAtBeg(p[arr[1]], arr[0], arr[2]);  
        }  
    }  
    printf("ADJACENCY LIST!!\n");  
    for (i = 0; i < n; i++)  
    {  
        printf("%d->", i);  
        displayList(p[i]);  
        printf("\n");  
    }  
    return p;  
}
```

```
}
```

```
void Initialize_Single_Source_Shortest_Path(int sourceVertex, int vertexCount, int distance[], int predecessor[])
```

```
{
```

```
    int i;
```

```
    for (i = 0; i < vertexCount; i++)
```

```
    {
```

```
        distance[i] = INT_MAX; // initialize all distances to infinity
```

```
        predecessor[i] = -1; // set all predecessors to null
```

```
    }
```

```
    distance[sourceVertex] = 0; // set distance of source vertex to 0
```

```
}
```

```
void BellmanFord_ShortestPath(int sourceVertex, struct node **graph, int vertexCount, int distance[], int predecessor[])
```

```
{
```

```
    int i, j, k;
```

```
    struct node *temp;
```

```
    // Step 1: Initialize the distance and predecessor arrays
```

```
    Initialize_Single_Source_Shortest_Path(sourceVertex, vertexCount, distance, predecessor);
```

```
    // Step 2: Relax edges repeatedly
```

```
    for (i = 1; i <= vertexCount - 1; i++)
```

```
    {
```

```
        for (j = 0; j < vertexCount; j++)
```

```
        {
```

```
            temp = graph[j];
```

```
            while (temp != NULL)
```

```
            {
```

```
                k = temp->vertex;
```

```
                if (distance[j] != INT_MAX && distance[k] > distance[j] + temp->weight)
```

```
                {
```

```
                    distance[k] = distance[j] + temp->weight;
```

```
                    predecessor[k] = j;
```

```
                }
```

```
                temp = temp->next;
```

```
            }
```

```
        }
```

```
    }
```

```
    // Step 3: Check for negative-weight cycles
```

```
    for (i = 0; i < vertexCount; i++)
```

```

{
    temp = graph[i];
    while (temp != NULL)
    {
        j = temp->vertex;
        if (distance[i] != INT_MAX && distance[j] > distance[i] + temp->weight)
        {
            printf("Graph contains negative weight cycle.\n");
            exit(1);
        }
        temp = temp->next;
    }
}
}

void Display_Path(int predecessor[], int sourceVertex, int currentVertex)
{
    if (currentVertex == sourceVertex)
    {
        printf("%d", sourceVertex);
    }
    else if (predecessor[currentVertex] == -1)
    {
        printf("No path exists from %d to %d", sourceVertex, currentVertex);
    }
    else
    {
        Display_Path(predecessor, sourceVertex, predecessor[currentVertex]);
        printf(" -> %d", currentVertex);
    }
}

void Display_Path_and_distance(int sourceVertex, int vertexCount, int distance[], int predecessor[])
{
    int i;
    for (i = 0; i < vertexCount; i++)
    {
        printf("\nShortest path from %d to %d: ", sourceVertex, i);
        Display_Path(predecessor, sourceVertex, i);
        if (distance[i] != INT_MAX) {
            printf(" with distance %d\n", distance[i]);
        }
    }
}

```

```

    }
}

int main()
{
    int i, n;

    fptr = fopen("./graph.txt", "r");
    fscanf(fptr, "%d", &n);

    int *distance = (int *)malloc(sizeof(int) * n);
    int *predecessor = (int *)malloc(sizeof(int) * n);
    node **p = createAdjacencyList(n, 1);
    BellmanFord_ShortestPath(0, p, n, distance, predecessor);
    Display_Path_and_distance(0, n, distance, predecessor);
    fclose(fptr);
}

```

OUTPUT

```

graph.txt
~/04_jyotiraditya

1 5
2 1 2 4
3 2 3 5
4 1 0 9
5 3 4 2
6 2 4 6
7 3 1 7
8

```

```

ADJACENCY LIST!!
0->
1->[0,9]->[2,4]
2->[4,6]->[3,5]
3->[1,7]->[4,2]
4->

Shortest path from 1 to 0: 1 -> 0 with distance 9
Shortest path from 1 to 1: 1 with distance 0
Shortest path from 1 to 2: 1 -> 2 with distance 4
Shortest path from 1 to 3: 1 -> 2 -> 3 with distance 9
Shortest path from 1 to 4: 1 -> 2 -> 4 with distance 10

```


2. Implement the Floyd Warshall's algorithm to find the all pair shortest path between every pair of vertices using adjacency matrix representation of graph. Follow the following function prototype in implementations:
- a) void Initialize_allPair_Shortest_Path(int vertexCount, int ** distance, int ** predecessor) (5)
 - b) void Floyd_Warshall_All_pair_shortestPath(int ** adjMatgraph, int vertexCount) (10)
//N.B. Allocate distance and predecessor matrix in this function dynamically
 - c) void Display_Path_and_distance(int sourceVertex, int vertexCount, int distance[], int predecessor[]) (5)
 - d) Integrate all functions through main() along with older functions create_adjacency_Matrix already given in previous assignment. (5)

ALGORITHM

```

FLOYD-WARSHALL( $W$ )
1   $n = W.rows$ 
2   $D^{(0)} = W$ 
3  for  $k = 1$  to  $n$ 
4      let  $D^{(k)} = (d_{ij}^{(k)})$  be a new  $n \times n$  matrix
5      for  $i = 1$  to  $n$ 
6          for  $j = 1$  to  $n$ 
7               $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
8  return  $D^{(n)}$ 

```

CODE

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

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

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

```
FILE *fptr;
```

```
int arr[3];
```

```
int **createAdjacencyMatrix(int n, int type)
```

```
{
```

```
    int **mat, i, j;
```

```
mat = (int **)malloc(n * sizeof(int *));

for (i = 0; i < n; i++)

{

    mat[i] = (int *)malloc(n * sizeof(int));

}

for (i = 0; i < n; i++)

{

    for (j = 0; j < n; j++)

    {

        if (i == j)

        {

            mat[i][j] = 0;

        }

        else

            mat[i][j] = INT_MAX;

    }

}

if (type == 1)

{

    while (!feof(fp_ptr))

    {

        for (i = 0; i < 3; i++)

        {

            fscanf(fp_ptr, "%d ", &arr[i]);
```

```

    }

    mat[arr[0]][arr[1]] = arr[2];

}

else

{

    while (!feof(fptr))

    {

        for (i = 0; i < 3; i++)

        {

            fscanf(fptr, "%d ", &arr[i]);

        }

        mat[arr[0]][arr[1]] = arr[2];

        mat[arr[1]][arr[0]] = arr[2];

    }

}

printf("ADJACENCY MATRIX : \n");

for (i = 0; i < n; i++)

{

    for (j = 0; j < n; j++)

    {

        if (mat[i][j] == INT_MAX)

            printf("INF\t");

        else

```

```

        printf("%d\t", mat[i][j]);

    }

    printf("\n");

}

return mat;

}

void initialize_all_pair_shortest_path(int **dist, int **pred, int **adjM, int n)
{
    int j, i;

    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            dist[i][j] = adjM[i][j];

            if (dist[i][j] != INT_MAX || dist[i][j] == 0)
            {
                pred[i][j] = i;
            }

            else
            {
                pred[i][j] = -1;
            }
        }
    }
}

```

```

    }
}

void print_path(int i, int j, int **pred)
{
    if(i!=j)
    {
        print_path(i, pred[i][j], pred);

        printf("v%d->", pred[i][j]);
    }
}

void printSolution(int **mat, int n)
{
    printf("\nShortest distances between every pair of vertices:\n");

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (mat[i][j] == INT_MAX)
            {
                printf("INF\t");
            }

            else
            {
                printf("%d\t", mat[i][j]);
            }
        }
    }
}

```

```

        }

    }

    printf("\n");

}

}

void allpairFloydWarshall(int **adjM, int n)

{

    int newdist, i, j, k;

    int **dist = (int **)malloc(n * sizeof(int *));

    for (i = 0; i < n; i++)

    {

        dist[i] = (int *)malloc(n * sizeof(int));

    }

    int **pred = (int **)malloc(n * sizeof(int *));

    for (i = 0; i < n; i++)

    {

        pred[i] = (int *)malloc(n * sizeof(int));

    }

    initialize_all_pair_shortest_path(dist, pred, adjM, n);

    for (k = 0; k < n; k++)

    {

        for (i = 0; i < n; i++)

        {

            for (j = 0; j < n; j++)

```

```

    {
        if (!(dist[i][k] == INT_MAX || dist[k][j] == INT_MAX))
        {
            newdist = dist[i][k] + dist[k][j];

            if (dist[i][j] > newdist)
            {
                dist[i][j] = newdist;
                pred[i][j] = pred[k][j];
            }
        }
    }
}

printf("\nPath Matrix:\n");

for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
        printf("%d\t", pred[i][j]);

    printf("\n");
}

printSolution(dist, n);

printf("enter source and destination");

scanf("%d%d",&i,&j);

print_path(i,j,pred);

printf("v%d\n",j);

```

```

}

int main(){

    int n, **p;

    fptr = fopen("graph.txt", "r");

    p = createAdjacencyMatrix(5, 1);

    allpairFloydWarshall(p, 5);

    fclose(fptr);

}

```

OUTPUT

graph.txt
~/04_jyotiraditya

```

1 1 2 4
2 2 3 5
3 1 0 9
4 3 4 2
5 2 4 6
6 3 1 7
7

```

```

ADJACENCY MATRIX :
0      INF      INF      INF      INF
9      0        4        INF      INF
INF     INF     0        5        6
INF     7       INF     0        2
INF     INF     INF     INF     0

Path Matrix:
0      -1       -1       -1       -1
1      1        1        2        2
1      3        2        2        2
1      3        1        3        3
-1     -1       -1       -1       4

Shortest distances between every pair of vertices:
0      INF      INF      INF      INF
9      0        4        9        10
21     12       0        5        6
16     7        11       0        2
INF     INF     INF     INF     0
enter source and destination3 1
v3->v1

```


Day 6

Dijkstra's single source shortest path

Implement Dijkstra single source shortest path

- a) void Initialize_singlesourceDjikstras (int *dist,int *pred,int vcount,int somevertex) [5]
- b) void DijkstraShortestPathSingleSource(struct node ** adlList, int *pred,int *dist,int vcount,int source) [10]
- c) PrintPath&Distance(int *pred, int * dist, int vi, int vj) [5]
- d) Integrate all the above function with main(). [10]

ALGORITHM

```
DIJKSTRA( $G, w, s$ )
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S = \emptyset$ 
3   $Q = G.V$ 
4  while  $Q \neq \emptyset$ 
5       $u = \text{EXTRACT-MIN}(Q)$ 
6       $S = S \cup \{u\}$ 
7      for each vertex  $v \in G.Adj[u]$ 
8          RELAX( $u, v, w$ )
```

CODE

```
#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

typedef struct Node{

    int weight;

    int vertex;

    struct Node *next;

}node;
```

```

void insertBeg(node **head, int ver, int wt)
{
    node *newnode=(node*)malloc(sizeof(node));

    newnode->vertex = ver;

    newnode->weight=wt;

    newnode->next = *head;

    *head = newnode;
}

node ** createlist(int n,int type){

    node **p=(node**)malloc(n*sizeof(node));

    int u,v,w,i;

    for(i=0;i<n;i++){

        p[i]=(node *)malloc(sizeof(node));

        p[i]=NULL;}

    FILE *fptr = fopen("graph.txt", "r");

    while(!feof(fptr))

    {

        fscanf(fptr, "%d %d %d\n", &u, &v, &w);

        insertBeg(&p[u], v, w);

        if(type == 0)

            insertBeg(&p[v], u, w);

    }

    fclose(fptr);

    printf("adjacency list:\n");
}

```

```

        for(i=0;i<n;i++){

            printf("\n%d",i);

            node*ptr=p[i];

            while(ptr!=NULL){

                printf("->%d-%d",ptr->vertex,ptr->weight);

                ptr=ptr->next;

            }

        }

        return p;

    }

void Initialize_Single_Source_Shortest_Path(int sourcevertex,int vertexcount,int distance[],int
predecessor[])

{

    int i;

    for (i = 0; i < vertexcount; i++) {

        distance[i] = INT_MAX;

        predecessor[i] = -1;

    }

    distance[sourcevertex] = 0;

}

int relax_dist(int dist[],int u,int v,int w)

{

    if((dist[u]+w)<dist[v])

```

```
        return 1;

    else

        return 0;

}
```

```
void dijkstras(int sourcevertex,node **graph,int vertexcount,int distance[],int predecessor[])

{

    Initialize_Single_Source_Shortest_Path(sourcevertex, vertexcount, distance, predecessor);

    int visited[vertexcount];

    int i, u, v, w;

    for(i = 0; i < vertexcount; i++)

        visited[i] = 0;

    for(i = 0; i < vertexcount-1; i++) {

        int min_distance = INT_MAX;

        for(v = 0; v < vertexcount; v++) {

            if(!visited[v] && distance[v] < min_distance) {

                min_distance = distance[v];

                u = v;

            }

        }

        visited[u] = 1;

        node *ptr = graph[u];

        while(ptr != NULL) {

            v = ptr->vertex;
```

```

w = ptr->weight;

if(relax_dist(distance, u, v, w)) {

    distance[v] = distance[u] + w;

    predecessor[v] = u;

}

ptr = ptr->next;

}

}

}

```

```

void Display_Path_and_distance(int sourcevertex,int vertexcount,int distance[],int predecessor[])

{

int i,j;

printf("Source node: %d\n", sourcevertex);

for (i = 0; i < vertexcount; i++) {

    printf("Shortest path from %d to %d: ", sourcevertex, i);

    if (distance[i] == INT_MAX)

    {

        printf("No path\n");

    }

    else

    {

        int *path;

```

```

    path=(int *)malloc(vertexcount*sizeof(int));

    int count = 0;

    int curr = i;

    while (curr != sourcevertex) {

        path[count] = curr;

        curr = predecessor[curr];

        count++;

    }

    path[count] = sourcevertex;

    for (j = count; j >= 0; j--) {

        printf("%d ", path[j]);

    }

    printf("with distance %d\n", distance[i]);

}

}

int main()

{

    int vertexcount, sourcevertex, dir, *distance, *predecessor, k;

    printf("Enter 0 for undirected and 1 for directed:");

    scanf("%d", &dir);

    printf("Enter number of vertices: ");

    scanf("%d", &vertexcount);

```

```

distance=(int *)malloc(vertexcount*sizeof(int));

predecessor=(int *)malloc(vertexcount*sizeof(int));


for(k=0;k<vertexcount;k++)

{

    dijkstras(k, createlist(5,dir), vertexcount, distance, predecessor);

    Display_Path_and_distance(k, vertexcount, distance, predecessor);

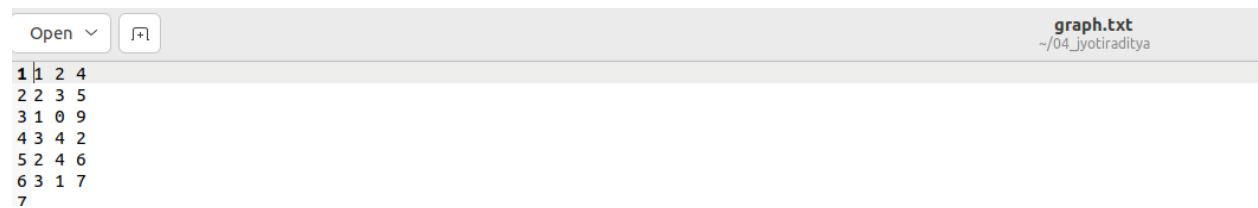
}

return 0;

}

```

OUTPUT

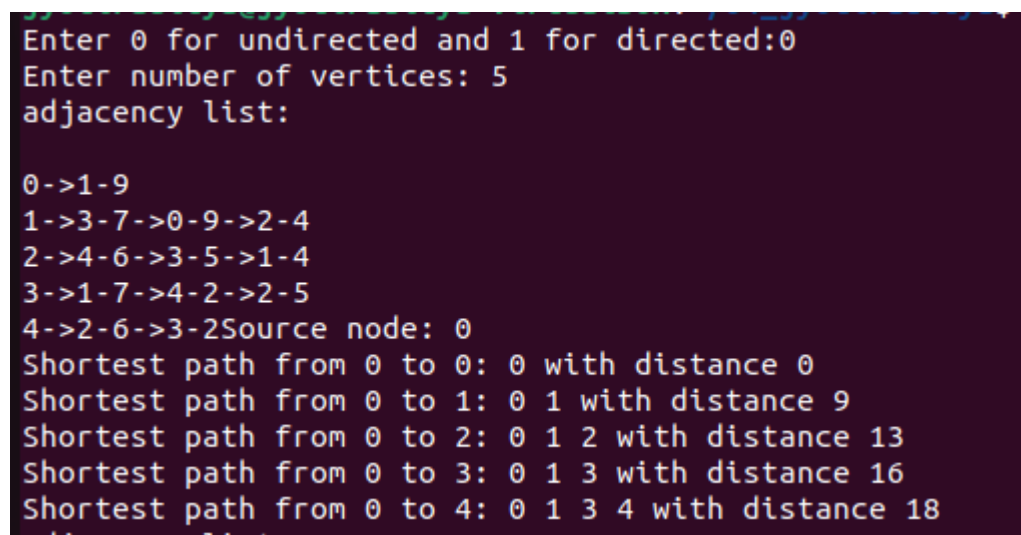


The screenshot shows a code editor window with a file named **graph.txt** located at `~/04_jyotiraditya`. The file contains the following adjacency list representation of a graph:

```

1 | 1 2 4
2 | 2 3 5
3 | 1 0 9
4 | 3 4 2
5 | 2 4 6
6 | 3 1 7
7

```



The terminal window shows the following output:

```

Enter 0 for undirected and 1 for directed:0
Enter number of vertices: 5
adjacency list:

0->1-9
1->3-7->0-9->2-4
2->4-6->3-5->1-4
3->1-7->4-2->2-5
4->2-6->3-2
Source node: 0
Shortest path from 0 to 0: 0 with distance 0
Shortest path from 0 to 1: 0 1 with distance 9
Shortest path from 0 to 2: 0 1 2 with distance 13
Shortest path from 0 to 3: 0 1 3 with distance 16
Shortest path from 0 to 4: 0 1 3 4 with distance 18

```

Day 7

Prim's algo

Implement following functions and Prim's Algorithm to find minimum spanning tree (MST) using adjacency list representation of an undirected graph.

- a) Void Prim_MST (struct node ** adjList, int * dist, int * ||_pred, int v_count) [10]
- b) Int arraybased_run_priorityQ(int *dist, int v_count) [5]
- c) Int printMST_edgelist(int * ||_pred,int u, int v) [5]
- d) Integrate all the above functions through main() [10]

ALGORITHM

```
MST-PRIM( $G, w, r$ )
1  for each  $u \in G.V$ 
2       $u.key = \infty$ 
3       $u.\pi = \text{NIL}$ 
4   $r.key = 0$ 
5   $Q = G.V$ 
6  while  $Q \neq \emptyset$ 
7       $u = \text{EXTRACT-MIN}(Q)$ 
8      for each  $v \in G.Adj[u]$ 
9          if  $v \in Q$  and  $w(u, v) < v.key$ 
10              $v.\pi = u$ 
11              $v.key = w(u, v)$ 
```

CODE

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

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

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

```
struct node
```

```
{
```

```
    int v;
```

```
    int wt;
```

```
    struct node *next;
```

```
};
```



```

void insert_beg(struct node **start,int vertex,int weight){

    struct node *ptr;

    ptr=(struct node *)malloc(sizeof(struct node));

    ptr->v=vertex;

    ptr->wt=weight;

    ptr->next=*start;

    *start=ptr;

}

struct node **adj_list(int ver_count){

    int i,u,v,w;

    struct node **list=(struct node**)malloc(ver_count*sizeof(struct node*)),*ptr;

    for(i=0;i<ver_count;i++) {

        list[i]=(struct node*)malloc(sizeof(struct node));

        list[i]=NULL;

    }

    FILE *fp;

    fp=fopen("Graph.txt","r");

    if(fp!=NULL) {

        while(!feof(fp)) {

            fscanf(fp,"%d %d %d\n",&u,&v,&w);

            insert_beg(&list[u],v,w);

            insert_beg(&list[v],u,w);

        }

    }

```

```

fclose(fp);

printf("\nAdjacency List:");

for(i=0;i<ver_count;i++) {

    ptr=list[i];

    if(ptr!=NULL)

        printf("\n%d",i);

    while(ptr!=NULL) {

        printf("->%d %d",ptr->v,ptr->wt);

        ptr=ptr->next;

    }

}

return list;
}

void Initialise_MST_Prim(int *pred,int *prqueue,int root_ver,int ver_count,int *visited,int *dist){

    int i;

    for(i=0;i<ver_count;i++) {

        pred[i]=-1;

        prqueue[i]=INT_MAX;

        visited[i]=0;

        dist[i]=INT_MAX;

    }

    pred[root_ver]=root_ver;

    prqueue[root_ver]=0;

    dist[root_ver]=0;

```

```

}

int Extract_Min(int *prqueue,int ver_count,int *visited)
{
    int i,min_ind=0;

    for(i=0;i<ver_count;i++)
    {
        if(prqueue[i]<prqueue[min_ind] && visited[i]!=1)
            min_ind=i;
    }

    visited[min_ind]=1;

    return min_ind;
}

void MST_Prim_Algorithm(struct node **list,int ver_count,int *pred,int *prqueue,int root_ver,int
*visited,int *dist)
{
    Initialise_MST_Prim(pred,prqueue,root_ver,ver_count,visited,dist);

    int u,prqsize=ver_count;

    while(prqsize>=0) {
        u=Extract_Min(prqueue,ver_count,visited);

        struct node *curr=list[u];

        while(curr!=NULL) {
            if(prqueue[curr->v]>curr->wt && visited[curr->v]!=1)
            {
                prqueue[curr->v]=curr->wt;

                pred[curr->v]=u;
            }
        }
    }
}

```

```

        dist[curr->v]=curr->wt;

    }

    curr=curr->next;

}

prqsize--;

}

}

void Print_MST_Edge_Distance(int *pred,int *dist,int ver_count,int source_ver){

    int i,tot=0;

    for(i=0;i<ver_count;i++) {

        if(i!=source_ver) {

            printf("%d--%d Distance=%d\n",i,pred[i],dist[i]);

            tot=tot+dist[i];

        }

    }

    printf("\nMinimum Weight:%d\n",tot);

}

int main(){

    int ver_count,root_ver;

    printf("\nEnter number of vertices: ");

    scanf("%d",&ver_count);

    printf("\nEnter root vertex: ");

    scanf("%d",&root_ver);

    int pred[ver_count],prqueue[ver_count],visited[ver_count],dist[ver_count];

```

```

struct node **list=adj_list(ver_count);

printf("\n");

MST_Prim_Algorithm(list,ver_count,pred,prqueue,root_ver,visited,dist);

printf("\n");

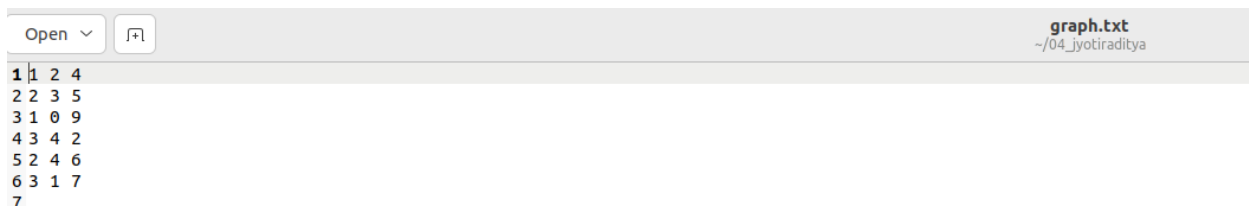
Print_MST_Edge_Distance(pred,dist,ver_count,root_ver);

return 0;

}

```

OUTPUT



```

graph.txt
~/04_jyotiraditya

1 1 2 4
2 2 3 5
3 1 0 9
4 3 4 2
5 2 4 6
6 3 1 7
7

```

Enter number of vertices: 5

Enter root vertex: 1

Adjacency List:

```

0->1 9
1->3 7->0 9->2 4
2->4 6->3 5->1 4
3->1 7->4 2->2 5
4->2 6->3 2

```

```

0--1 Distance=9
2--1 Distance=4
3--2 Distance=5
4--3 Distance=2

```

Minimum Weight:20

Day 8

Kruskal's Algo

Implement following functions in Kruskal's Algorithm to find minimum spanning tree (MST) using adjacency list representation of an undirected graph.

[N.B. struct forest_node will contain 3 members- (a) Element (i.e., vertex_no), (b) parent, (c) Rank (height)

N.B. struct edge contain edge information (a) origin vertex, (b) end vertex, (c) weight of edge

a) Void Kruskal_MST (struct node ** adjList, int * dist, int * PI_pred, int v_count) [10]

b) void make_disjoint_Set_Forest (struct forest_node* forest_array, int v_count) [5]

c) Sort_edge_List(struct edge * edges_array, int edge_count) [5]

d) int Find_set_representative(struct forest_node* forest_array, int vertex) [5]

e) void union_of_disjoint_sets(struct forest_node* forest_array, int vertex1, int vertex2) [5]

f) Int print_MST_edgelist(int * PI_pred ,int u, int v) [5]

g) Integrate all the above functions through main() [10]

Day 9

0-1 Knapsack, LCS, KMP

Day 10

Connected component, DFS, BFS

Day 11

Ford Fulkerson

Day 12

Skip list