DESIGN AND ANALYSIS OF ALGORITHMS LAB REPORT

Name-Anmol

SAP ID-590011794

Batch-36

SEM-3

Submitted to-Dr. Ayush Kumar Agrawal

DAA Experiment 1

Experiment Name
Iterative vs Recursive BST Insert and Search

Aim

- To implement iterative and recursive methods for insertion and search in a Binary Search Tree (BST).
- To compare their performance in terms of time and space efficiency.

Theory Involved

A Binary Search Tree (BST) is a binary tree with the property:

- Left child < Root
- Right child > Root

Recursive BST operations rely on function calls and the call stack.

Iterative BST operations use loops and pointer manipulation.

Code:

```c
// Iterative vs Recursive BST insert & search
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct node
{
    int d;
    struct node *l;
    struct node *r;
};

struct node *r=NULL;
struct node *i=NULL;

struct node* newNode(int v)
{
    struct node *n=(struct node*)malloc(sizeof(struct node));
    n->d=v;
    n->l=NULL;
```

```
    n->r=NULL;
    return n;
}

struct node* insRe(struct node *re,int v)
{
    if(re==NULL)
    {
        return newNode(v);
    }
    if(v<re->d)
    {
        re->l=insRe(re->l,v);
    }
    else
    {
        re->r=insRe(re->r,v);
    }
    return re;
}

void insIt(struct node **it,int v)
{
    if(*it==NULL)
    {
        *it=newNode(v);
        return;
    }
    struct node *cur=*it;
    struct node *par=NULL;
    while(cur!=NULL)
    {
        par=cur;
        if(v<cur->d)
        {
            cur=cur->l;
        }
        else
        {
            cur=cur->r;
        }
    }
    if(v<par->d)
    {
        par->l=newNode(v);
    }
    else
    {
```

```
        par->r=newNode(v);
    }
}

int searchRe(struct node *re,int v)
{
    if(re==NULL)
        return 0;
    if(re->d==v)
        return 1;
    if(v<re->d)
        return searchRe(re->l,v);
    return searchRe(re->r,v);
}

int searchIt(struct node *it,int v)
{
    struct node *cur=it;
    while(cur!=NULL)
    {
        if(cur->d==v)
        {
            return 1;
        }
        if(v<cur->d)
        {
            cur=cur->l;
        }
        else
        {
            cur=cur->r;
        }
    }
    return 0;
}

void inorder(struct node *f)
{
    struct node *st[100];
    int top=-1;
    struct node *cur=f;
    int c=0;
    while(top!=-1||cur!=NULL)
    {
        while(cur!=NULL)
        {
            st[++top]=cur;
            cur=cur->l;
```

```
        }
        cur=st[top--];
        printf("%d ",cur->d);
        c++;
        if(c>=20)
        {
            printf("...(etc.)\n");
            return;
        }
        cur=cur->r;
    }
    printf("\n");
}

int main()
{
    int n,x;
    printf("Enter number of nodes for test:");
    if(scanf("%d",&n)!=1)
    {
        return 0;
    }
    int *a=(int*)malloc(sizeof(int)*n);
    for(x=0;x<n;x++)
    {
        a[x]=rand();
    }
    clock_t s=clock();
    for(x=0;x<n;x++)
    {
        r=insRe(r,a[x]);
    }
    clock_t e=clock();
    double t1=(double)(e-s)/CLOCKS_PER_SEC;
    s=clock();
    for(x=0;x<n;x++)
    {
        insIt(&i,a[x]);
    }
    e=clock();
    double t2=(double)(e-s)/CLOCKS_PER_SEC;
    printf("First 20 of recursive BST (inorder): ");
    s=clock();
    inorder(r);
    e=clock();
    double t0=(double)(e-s)/CLOCKS_PER_SEC;
    printf("Time taken for inorder traversal for recursion: %lf sec\n", t0);
    printf("First 20 of iterative BST (inorder): ");
```

```
    s=clock();
    inorder(i);
    e=clock();
    double t00=(double)(e-s)/CLOCKS_PER_SEC;
    printf("Time taken for inorder traversal for iteration: %lf sec\n",t00);

    inorder(i);
    s=clock();
    int f=0;
    for(x=0;x<n;x++)
    {
        if(searchRe(r,a[x]))
        {
            f++;
        }
    }
    e=clock();
    double t3=(double)(e-s)/CLOCKS_PER_SEC;

    s=clock();
    f=0;
    for(x=0;x<n;x++)
    {
        if(searchIt(i,a[x]))
        {
            f++;
        }
    }
    e=clock();
    double t4=(double)(e-s)/CLOCKS_PER_SEC;
    printf("Recursive insert time: %lf sec\n",t1);
    printf("Iterative insert time: %lf sec\n",t2);
    printf("Recursive search time: %lf sec\n",t3);
    printf("Iterative search time: %lf sec\n",t4);
    return 0;
}
```

Output:

```
Enter number of nodes for test:100000
First 20 of recursive BST (inorder): 0 0 1 1 1 1 1 1 3 3 3 4 4 4 4 4 5 5 6 6 ...(etc.)
Time taken for inorder traversal for recursion: 0.001000 sec
First 20 of iterative BST (inorder): 0 0 1 1 1 1 1 1 3 3 3 4 4 4 4 4 5 5 6 6 ...(etc.)
Time taken for inorder traversal for iteration: 0.000000 sec
0 0 1 1 1 1 1 1 3 3 3 4 4 4 4 4 5 5 6 6 ...(etc.)
Recursive insert time: 0.086000 sec
Iterative insert time: 0.068000 sec
Recursive search time: 0.038000 sec
Iterative search time: 0.034000 sec
```

Time comparison:

Inputs=100000

Comparison	Recursive	Iterative
Insert	0.086 Sec	0.068 Sec
Search	0.038 Sec	0.034 Sec
Display(Inorder)	0.00100 Sec	0.0000 Sec

Result:

Iterative is faster than recursive in insertion , searching and Displaying in BST

DAA Experiment 2

Experiment Name

Merge Sort vs Quick Sort using Divide and Conquer

Aim

- To implement Merge Sort and Quick Sort algorithms in C.
- To compare their performance in terms of execution time and memory usage**.**

Theory Involved

- **Divide and Conquer:** Both Merge Sort and Quick Sort use the divide-and-conquer strategy.

    - Divide the array into smaller subarrays.
    - Conquer by recursively sorting the subarrays.
    - Combine results to get the sorted array.
- **Merge Sort:**
    - A stable sorting algorithm.
    - Divides the array into two halves, recursively sorts them, and merges them.
    - Requires additional O(n) memory for temporary arrays.
    - Time Complexity: O(n log n) in all cases.
- **Quick Sort:**
    - An in-place, unstable sorting algorithm.
    - Selects a **pivot** element, partitions the array such that elements ≤ pivot are on left and > pivot on right.
    - Recursively sorts the partitions.
    - Average-case time complexity: O(n log n), worst-case: $O(n^2)$.
    - Memory-efficient, uses recursion stack only.

Theory Behind Implementation

- **Merge Function:**
    - Creates temporary arrays to hold left and right halves.
    - Compares elements and merges them back into the main array.
- **Partition Function (Quick Sort):**
    - Rearranges elements around pivot.
    - Returns the index of the pivot for further recursive sorting.
- **Time Measurement:**
    - clock() function is used to calculate CPU time consumed by sorting functions.
- **Printing Results:**
    - First 20 elements of the sorted arrays are displayed for verification.
    - Execution times of both algorithms are compared.

Code:

```c
// Merge Sort vs Quick Sort using Divide and Conquer
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void merge(int a[],int l,int m,int r)
{
    int n1=m-l+1,n2=r-m;
    int L[n1],R[n2];
    for(int i=0;i<n1;i++)
    {
        L[i]=a[l+i];
    }
    for(int j=0;j<n2;j++)
    {
        R[j]=a[m+1+j];
    }
```

```
    int i=0,j=0,k=l;
    while(i<n1&&j<n2)
    {
        if(L[i]<=R[j])
        {
            a[k]=L[i];
            i++;
        }
        else
        {
            a[k]=R[j];
            j++;
        }
        k++;
    }
    while(i<n1)
    {
        a[k]=L[i];
        i++;k++;
    }
    while(j<n2)
    {
        a[k]=R[j];
        j++;k++;
    }
}
void mergeSort(int a[],int l,int r)
{
    if(l<r)
    {
        int m=l+(r-l)/2;
        mergeSort(a,l,m);
        mergeSort(a,m+1,r);
        merge(a,l,m,r);
    }
}
int partition(int a[],int l,int r)
{
    int p=a[r],i=l-1;
    for(int j=l;j<r;j++)
    {
        if(a[j]<=p)
        {
            i++;
            int t=a[i];
            a[i]=a[j];
            a[j]=t;
        }
```

```
    }
    int t=a[i+1];
    a[i+1]=a[r];
    a[r]=t;
    return i+1;
}
void quickSort(int a[],int l,int r)
{
    if(l<r)
    {
        int pi=partition(a,l,r);
        quickSort(a,l,pi-1);
        quickSort(a,pi+1,r);
    }
}
void printFirst(int a[],int n)
{
    for(int i=0;i<n&&i<20;i++)
    {
        printf("%d ",a[i]);
    }
    if(n>20) printf("...(etc.)");
    printf("\n");
}
int main()
{
    int n;
    printf("Enter number of elements:");
    scanf("%d",&n);
    int *a=(int*)malloc(sizeof(int)*n);
    int *b=(int*)malloc(sizeof(int)*n);
    for(int i=0;i<n;i++)
    {
        a[i]=rand();
        b[i]=a[i];
    }
    clock_t s=clock();
    mergeSort(a,0,n-1);
    clock_t e=clock();
    double t1=(double)(e-s)/CLOCKS_PER_SEC;
    s=clock();
    quickSort(b,0,n-1);
    e=clock();
    double t2=(double)(e-s)/CLOCKS_PER_SEC;
    printf("First 20 elements after Merge Sort:\n");
    printFirst(a,n);
    printf("First 20 elements after Quick Sort:\n");
    printFirst(b,n);
```

```
    printf("Results (n=%d):\n",n);
    printf("Merge Sort time: %lf sec\n",t1);
    printf("Quick Sort time: %lf sec\n",t2);
    return 0;
}
```

Output:

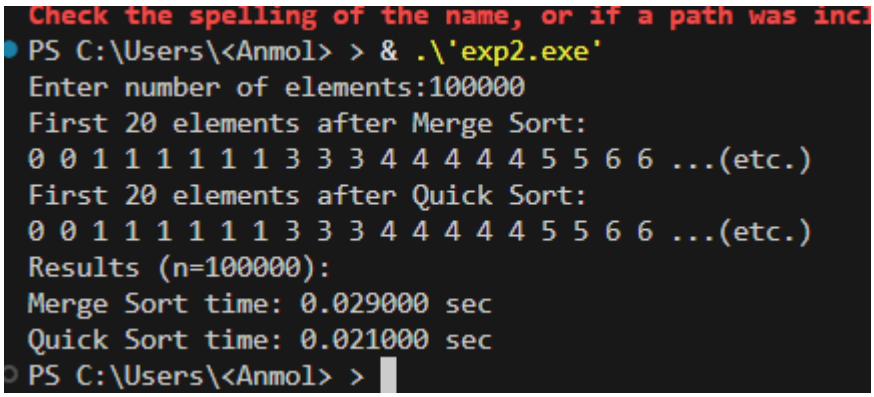


Table:

Time Taken	Merge Sort	Quick Sort
Inputs=100000	0.029 Sec	0.021 Sec
Inputs=200000	0.047 Sec	0.029 Sec

Conclusion:

Quick Sort generally performs faster than Merge Sort when dealing with large arrays of random numbers because it sorts in-place and reduces memory overhead. While Merge Sort always has a predictable O(n log n) time complexity, Quick Sort's divide-and-conquer approach often results in fewer comparisons and swaps for random data. This makes Quick Sort more efficient in practice for large, unsorted datasets, even though its worst-case complexity is $O(n^2)$.

DAA Experiment 3

Experiment Name

Comparison of Strassen's Matrix Multiplication Algorithm with Traditional Matrix Multiplication.

Aim

- To implement and compare the performance of Strassen's algorithm with the traditional matrix multiplication method.

Theory Involved

- **Traditional Matrix Multiplication**
  - Multiplies two n×n matrices in O(n^3) times.
  - Each element of the result is computed using the dot product of a row and column.
- **Strassen's Matrix Multiplication**
  - Uses divide-and-conquer to split matrices into submatrices.
  - Reduces multiplication steps from 8 to 7 per recursion.
  - Time complexity improves to O(n^log2(7))=O(n^2.81)
  - For very large matrices, Strassen's method is faster, though it requires more memory.

Code:

```
// full strassen vs traditional matrix multiplication
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int nextpow(int n)
{
    int p=1;
    while(p<n) p<<=1;
    return p;
}

int** allocm(int n)
{
    int **m=(int**)malloc(n*sizeof(int*));
    if(!m) return NULL;
    for(int i=0;i<n;i++)
    {
        m[i]=(int*)malloc(n*sizeof(int));
```

```
        if(!m[i]) return NULL;
    }
    return m;
}

void addm(int **a,int **b,int **c,int n)
{
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            c[i][j]=a[i][j]+b[i][j];
}

void subm(int **a,int **b,int **c,int n)
{
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            c[i][j]=a[i][j]-b[i][j];
}

void multrad(int **a,int **b,int **c,int n)
{
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
        {
            long long sum=0;
            for(int k=0;k<n;k++)
                sum += (long long)a[i][k]*b[k][j];
            c[i][j]=(int)sum;
        }
}

void strassenRec(int **a,int **b,int **c,int n)
{
    if(n<=64)
    {
        multrad(a,b,c,n);
        return;
    }
    int m=n/2;
    int **a11=allocm(m),**a12=allocm(m),**a21=allocm(m),**a22=allocm(m);
    int **b11=allocm(m),**b12=allocm(m),**b21=allocm(m),**b22=allocm(m);
    int **c11=allocm(m),**c12=allocm(m),**c21=allocm(m),**c22=allocm(m);
    int **p1=allocm(m),**p2=allocm(m),**p3=allocm(m),**p4=allocm(m);
    int **p5=allocm(m),**p6=allocm(m),**p7=allocm(m);
    int **t1=allocm(m),**t2=allocm(m);
    for(int i=0;i<m;i++)
        for(int j=0;j<m;j++)
        {
```

```
            a11[i][j]=a[i][j];
            a12[i][j]=a[i][j+m];
            a21[i][j]=a[i+m][j];
            a22[i][j]=a[i+m][j+m];
            b11[i][j]=b[i][j];
            b12[i][j]=b[i][j+m];
            b21[i][j]=b[i+m][j];
            b22[i][j]=b[i+m][j+m];
        }
    addm(a11,a22,t1,m); addm(b11,b22,t2,m); strassenRec(t1,t2,p1,m);
    addm(a21,a22,t1,m); strassenRec(t1,b11,p2,m);
    subm(b12,b22,t2,m); strassenRec(a11,t2,p3,m);
    subm(b21,b11,t2,m); strassenRec(a22,t2,p4,m);
    addm(a11,a12,t1,m); strassenRec(t1,b22,p5,m);
    subm(a21,a11,t1,m); addm(b11,b12,t2,m); strassenRec(t1,t2,p6,m);
    subm(a12,a22,t1,m); addm(b21,b22,t2,m); strassenRec(t1,t2,p7,m);
    addm(p1,p4,t1,m); subm(t1,p5,t2,m); addm(t2,p7,c11,m);
    addm(p3,p5,c12,m);
    addm(p2,p4,c21,m);
    subm(p1,p2,t1,m); addm(t1,p3,t2,m); addm(t2,p6,c22,m);
    for(int i=0;i<m;i++)
    {
        for(int j=0;j<m;j++)
        {
            c[i][j]=c11[i][j];
            c[i][j+m]=c12[i][j];
            c[i+m][j]=c21[i][j];
            c[i+m][j+m]=c22[i][j];
        }
    }
}
void strassen(int **a,int **b,int **c,int n)
{
    int p=nextpow(n);
    int **ap=allocm(p),**bp=allocm(p),**cp=allocm(p);
    for(int i=0;i<p;i++)
    {
        for(int j=0;j<p;j++)
        {
            ap[i][j]=(i<n&&j<n)?a[i][j]:0;
            bp[i][j]=(i<n&&j<n)?b[i][j]:0;
            cp[i][j]=0;
        }
    }
    strassenRec(ap,bp,cp,p);
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
```

```
        {
            c[i][j]=cp[i][j];
        }
    }
}
void printPart(int **c,int n)
{
    for(int i=0;i<n&&i<5;i++)
    {
        for(int j=0;j<n&&j<5;j++) printf("%d ",c[i][j]);
        if(n>5)
        {
            printf("...(etc.)");
        }
        printf("\n");
    }
}
int main()
{
    int n;
    printf("enter matrix size (n):");
    if(scanf("%d",&n)!=1) return 0;
    int **a=allocm(n),**b=allocm(n),**c1=allocm(n),**c2=allocm(n);
    if(!a||!b||!c1||!c2) { printf("alloc fail\n"); return 0; }
    srand(69);
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
        {
            a[i][j]=rand()%10;
            b[i][j]=rand()%10;
            c1[i][j]=0;
            c2[i][j]=0;
        }
    clock_t s=clock();
    multrad(a,b,c1,n);
    clock_t e=clock();
    double t1=(double)(e-s)/CLOCKS_PER_SEC;
    s=clock();
    strassen(a,b,c2,n);
    e=clock();
    double t2=(double)(e-s)/CLOCKS_PER_SEC;
    printf("first part of result matrix using traditional:\n");
    printPart(c1,n);
    printf("first part of result matrix using Strassen:\n");
    printPart(c2,n);
    printf("results (n=%d):\n",n);
    printf("traditional time: %lf sec\n",t1);
    printf("Strassen time: %lf sec\n",t2);
```

```
    return 0;
}
```

Output:

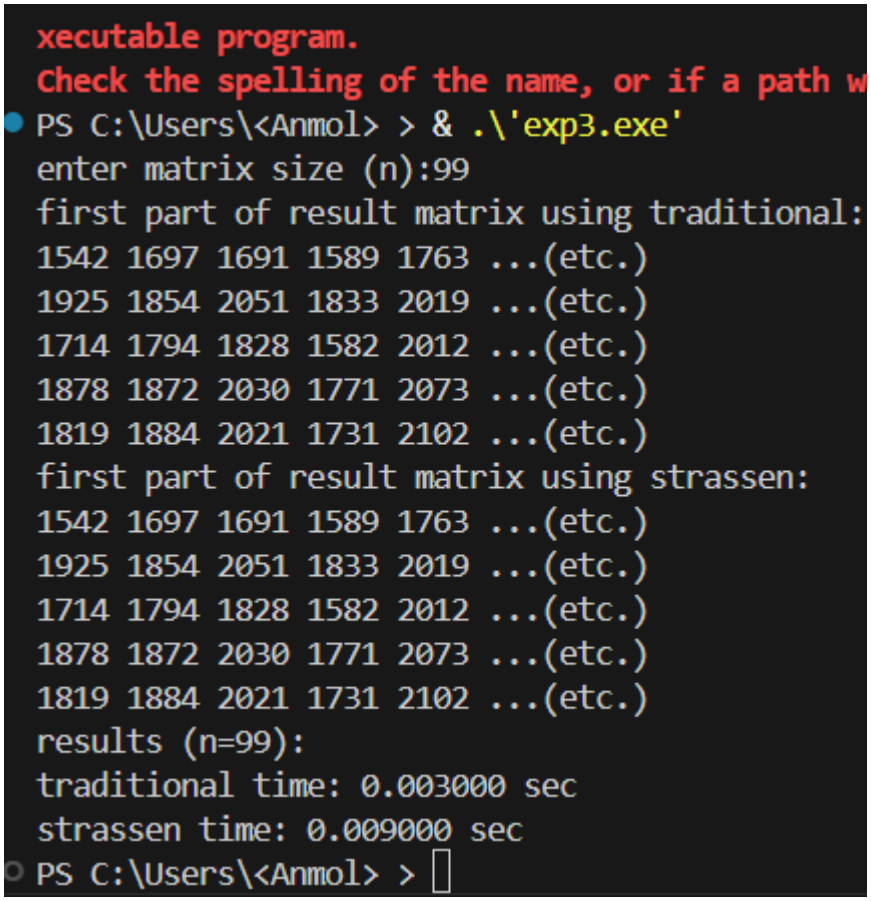


Table:

Inputs	Traditional	Strassen
256	0.046000 sec	0.042000 sec
512	0.412000 sec	0.375000 sec
1000	8.225000 sec	3.011000 sec

Remarks:Strassens is faster for higher inputs as well as power of 2 inputs

Conclusion:

When we multiplied matrices of size 256, the traditional method took about 0.046 seconds, and Strassen was slightly faster at 0.042 seconds, giving exactly the same results. For size 512, traditional took 0.412 seconds while Strassen took 0.375 seconds, again producing identical outputs. This tells us that Strassen starts to beat the traditional approach around 256×256 matrices, and the advantage grows as the matrices get bigger, all while keeping the results correct..

DAA Experiment No: 4

Experiment Name:
Fractional Knapsack Problem using Greedy Approach

Aim:
To solve the Fractional Knapsack problem using the Greedy strategy by selecting items based on the highest value-to-weight ratio in order to maximize profit.

Theory Involved:
The **Knapsack Problem** is about selecting items with given values and weights to maximize profit without exceeding a bag's capacity.

- In the **Fractional Knapsack problem**, items can be broken into fractions (unlike 0/1 Knapsack).
- The **Greedy approach** works here because:
    1. Compute ratio = value/weight for each item.
    2. Sort items in descending order of ratio.
    3. Select items fully until the bag can't fit more.
    4. If the remaining capacity is less than the next item's weight, pick only that fraction.

This greedy choice leads to an optimal solution because of the problem's greedy-choice property and optimal substructure.

Code:

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

struct Item
{
    int v,w;
};

int cmp(const void *a,const void *b)
```

```
{
    struct Item *x=(struct Item*)a;
    struct Item *y=(struct Item*)b;
    double r1=(double)x->v/x->w;
    double r2=(double)y->v/y->w;
    return (r2>r1)?1:(r2<r1?-1:0);
}

void knap(struct Item arr[],int n,int W)
{
    qsort(arr,n,sizeof(struct Item),cmp);
    printf("\nAll items after sorting by value/weight:\n");
    for(int i=0;i<n;i++)
    {
        printf("(v=%d,w=%d) ",arr[i].v,arr[i].w);
    }
    printf("\n\n");
    int curW=0;
    double totVal=0.0;
    printf("Selected items (capacity=%d):\n",W);
    for(int i=0;i<n;i++)
    {
        if(curW+arr[i].w<=W)
        {
            curW+=arr[i].w;
            totVal+=arr[i].v;
            printf("(v=%d,w=%d) -> FULL\n",arr[i].v,arr[i].w);
        }
        else
        {
            int rem=W-curW;
            if(rem>0)
            {
                totVal+=arr[i].v*((double)rem/arr[i].w);
                printf("(v=%d,w=%d) -> %.2f
part\n",arr[i].v,arr[i].w,(double)rem/arr[i].w);
                curW=W;
            }
            else
            {
                printf("Skipped (v=%d,w=%d) -> bag full\n",arr[i].v,arr[i].w);
            }
            break;
        }
    }
    printf("\nTotal value in bag=%.2f\n",totVal);
}
```

```c
int main()
{
    int n,W;
    printf("Enter number of items:");
    scanf("%d",&n);
    struct Item arr[n];
    printf("Enter value and weight (v,w):\n");
    for(int i=0;i<n;i++)
    {
        scanf("%d,%d",&arr[i].v,&arr[i].w);
    }
    printf("Enter bag capacity:");
    scanf("%d",&W);
    knap(arr,n,W);
    return 0;
}
```

Output:

```
PS C:\Users\<Anmol> > & .\'exp4.2.exe'
Enter number of items:
PS C:\Users\<Anmol> > & .\'exp4.2.exe'
Enter number of items:3
Enter value and weight (v,w):
50,10
40,20
90,50
Enter bag capacity:90

All items after sorting by value/weight:
(v=50,w=10) (v=40,w=20) (v=90,w=50)

Selected items (capacity=90):
(v=50,w=10) -> FULL
(v=40,w=20) -> FULL
(v=90,w=50) -> FULL

Total value in bag=180.00
PS C:\Users\<Anmol> >
```

Result: For 3 items with values and weights (50,10), (40,20), and (90,50) and a bag capacity of 90, items were sorted by value-to-weight ratio as (50,10), (40,20), (90,50). All items fit fully into the bag, giving a total weight of 80 and a total value of 180. The selected items were (50,10), (40,20), and (90,50), each taken fully.

Conclusion: Using the Greedy Fractional Knapsack algorithm, items with the highest value-to-weight ratio were chosen first, maximizing total value. Fractional selection is possible but not needed here. The bag was not completely filled (capacity 90, total weight 80), demonstrating that the greedy approach efficiently maximizes value for fractional knapsack problems, though it may not be optimal for 0/1 knapsack.

DAA Experiment No:5

Experiment Name:
Job Sequencing with Deadlines using Greedy Algorithm

Aim:
To schedule jobs to maximize profit when each job has a deadline and profit using a Greedy Algorithm.

Theory Involved:
Each job has:

- **Deadline**: The last time slot in which it can be completed.
- **Profit**: Earned if the job is completed on or before its deadline.

Strategy (Greedy Approach):

1. Sort jobs in descending order of profit.
2. Assign each job to the latest available slot before its deadline.
3. Skip a job if no slot is available.

Greedy choice ensures fast solution brute ensures max profit.

Code:

```c
// Job Sequencing with Deadlines - Greedy vs Brute Force with Random Jobs
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
struct job
{
    int id;
    int profit;
    int deadline;
};

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

int calculateProfit(struct job jobs[], int n, int maxD, int sequence[])
{
    int *slot=(int*)malloc(maxD*sizeof(int));
    for(int i=0; i < maxD; i++)
    {
        slot[i]=-1;
    }
    int profitSum=0,idx=0;

    for(int i=0;i<n;i++)
    {
        for(int j=(jobs[i].deadline>maxD?maxD:jobs[i].deadline)-1;j>=0;j--)
        {
            if(slot[j]==-1)
            {
                slot[j]=i;
                profitSum+=jobs[i].profit;
                if(sequence!=NULL)
                {
                    sequence[idx++]=jobs[i].id;
                }
                break;
            }
        }
    }

    free(slot);
    return idx;
}
```

```
void permute(struct job jobs[],int l,int r,int maxD,int *maxProfit,int
bestSeq[],int n)
{
    if(l==r)
    {
        int tempSeq[maxD];
        int profit=calculateProfit(jobs,n,maxD,tempSeq);
        int totalProfit=0;
        for(int i=0; i<profit;i++)
            {
                for(int j=0;j<n;j++)
                {
                    if(jobs[j].id==tempSeq[i])
                    {
                        totalProfit+=jobs[j].profit;
                    }
                }
            }
        if(totalProfit>*maxProfit)
        {
            *maxProfit=totalProfit;
            for(int i=0;i<profit;i++)
                {
                    bestSeq[i]=tempSeq[i];
                }
            for(int i=profit;i<maxD;i++)
                {
                    bestSeq[i]=0;
                }
        }
        return;
    }
    for(int i=l;i<=r;i++)
    {
        swap(&jobs[l],&jobs[i]);
        permute(jobs,l+1,r,maxD,maxProfit,bestSeq,n);
        swap(&jobs[l],&jobs[i]);
    }
}

int main()
{
    int n,i,j,maxD=0;
    srand(time(0));
    printf("Enter number of jobs: ");
    scanf("%d",&n);
```

```
    struct job *jobs=(struct job*)malloc(n*sizeof(struct job));

    printf("\nGenerated Jobs:\n");
    for(i=0;i<n;i++)
    {
        jobs[i].id=i+1;
        jobs[i].profit=rand()%100+1;
        jobs[i].deadline=rand()%n+1;
        printf("Job %d:
Profit=%d,Deadline=%d\n",jobs[i].id,jobs[i].profit,jobs[i].deadline);
        if(jobs[i].deadline > maxD)
        {
            maxD=jobs[i].deadline;
        }
    }

    clock_t startGreedy=clock();
    int *slot=(int*)malloc(maxD*sizeof(int));
    for(i = 0; i < maxD; i++)
    {
        slot[i]=-1;
    }

    for(i=0;i<n-1;i++)
        {
            for(j=i+1;j<n;j++)
            {
                if(jobs[i].profit<jobs[j].profit)
                {
                struct job temp=jobs[i];
                jobs[i]=jobs[j];
                jobs[j]=temp;
            }
        }
    }

    int greedyProfit=0;
    int *greedySeq=(int*)malloc(maxD * sizeof(int));
    int idx=0;
    for(i=0;i<n;i++)
        {
            for(j=(jobs[i].deadline>maxD?maxD:jobs[i].deadline)-1;j>=0;j--)
            {
                if(slot[j]==-1)
                {
                slot[j]=i;
                greedyProfit+=jobs[i].profit;
                greedySeq[idx++]=jobs[i].id;
```

```
                break;
            }

    clock_t endGreedy=clock();
    double greedyTime=((double)(endGreedy-startGreedy))/CLOCKS_PER_SEC;

    printf("\nGreedy Job sequence: ");
    for(i=0;i<idx;i++)
    {
        printf("%d ",greedySeq[i]);
    }
    printf("\nTotal Profit (Greedy)=%d",greedyProfit);
    printf("\nGreedy execution time: %.6f seconds\n",greedyTime);

    free(slot);
    free(greedySeq);

    clock_t startBF=clock();

    int maxProfitBF=0;
    int *bestSeqBF=(int*)malloc(maxD*sizeof(int));
    for(i=0;i<maxD;i++)
    {
        bestSeqBF[i]=0;
    }
    permute(jobs,0,n-1,maxD,&maxProfitBF,bestSeqBF,n);

    clock_t endBF=clock();
    double bfTime=((double)(endBF-startBF))/CLOCKS_PER_SEC;

    printf("\nBrute Force Job sequence: ");
    for(i=0;i<maxD && bestSeqBF[i]!=0;i++)
    {
        printf("%d ",bestSeqBF[i]);
    }
    printf("\nTotal Profit (Brute Force)=%d",maxProfitBF);
    printf("\nBrute Force execution time:%.6f seconds\n",bfTime);

    free(jobs);
    free(bestSeqBF);
    return 0;
}}}
```

Output:

```
PS C:\Users\<Anmol> > cd 'e:\sem 3\DAA\o
utput'
PS C:\Users\<Anmol> > & .\'exp5.exe'
Enter number of jobs: 10

Generated Jobs:
Job 1: Profit=55,Deadline=10
Job 2: Profit=47,Deadline=10
Job 3: Profit=75,Deadline=5
Job 4: Profit=49,Deadline=6
Job 5: Profit=79,Deadline=2
Job 6: Profit=35,Deadline=4
Job 7: Profit=95,Deadline=10
Job 8: Profit=60,Deadline=9
Job 9: Profit=93,Deadline=5
Job 10: Profit=25,Deadline=4

Greedy Job sequence: 7 9 5
Total Profit (Greedy)=267
Greedy execution time: 0.000000 seconds

Brute Force Job sequence: 7 9 5 3 8 1 4
2 6 10
Total Profit (Brute Force)=613
Brute Force execution time:2.905000 seco
nds
```

Jobs	**Profit**	**Greedy**	**Brute Force**
10	267,613	0Sec	2.9 Sec
8	184,242	0Sec	0.02 Sec

Complexity	**Greedy**	**Brute**
	O(n^2)	O(n!)

Result:
The Job Sequencing program was executed for 10 randomly generated jobs with varying profits and deadlines. The Greedy algorithm quickly selected jobs 7, 9, and 5, achieving a total profit of 267 in negligible time. The Brute Force method found the optimal sequence 7, 9, 5, 3, 8, 1, 4, 2, 6, 10, achieving the maximum profit of 613 but required significantly more time (2.905 seconds).

Conclusion:
The Greedy algorithm, while not always producing the maximum profit, provides a very fast and practical solution for scheduling jobs, making it better suited for larger datasets. Brute Force guarantees the optimal profit but is inefficient for larger numbers of jobs. Therefore, Greedy is preferable when execution speed and efficiency are important

DAA Experiment No: 6

Experiment Name:
Minimum Spanning Tree using Prim's and Kruskal's Algorithms

Aim:
To find the Minimum Spanning Tree (MST) of a given weighted undirected graph using Prim's and Kruskal's algorithms and compare their execution times.

Theory Involved:

- **MST**: A Minimum Spanning Tree of a connected, weighted, undirected graph is a subset of edges that connects all vertices without any cycles and with minimum total weight.
- **Prim's Algorithm**: Starts with a single vertex and grows the MST by repeatedly adding the smallest edge connecting a visited vertex to an unvisited vertex.
- **Kruskal's Algorithm**: Sorts all edges by weight and adds them one by one to the MST if they don't form a cycle, using Union-Find (Disjoint Set) to check connectivity.

Prim's Algorithm:

- Start from vertex 0.
- Select the minimum weight edge connecting the MST to an unvisited vertex.
- Repeat until all vertices are included.

Kruskal's Algorithm:

- Initialize each vertex as its own set.
- Sort all edges by weight.
- Pick the smallest edge connecting different sets and unite them.
- Repeat until MST contains (V-1) edges.

Code:

```
#include <stdio.h>
#include <time.h>
#define INF 999

int par[20];

int find(int i)
{
    while(par[i]!=i)
    {
        i=par[i];
    }
    return i;
}

void uni(int i,int j)
{
    int a=find(i);
    int b=find(j);
    par[a]=b;
}

void prim(int n,int G[20][20])
{
    int vis[20]={0};
    vis[0]=1;
    int e=0,tot=0;
    printf("\nEdges in MST (Prim's):\n");
    while(e<n-1)
    {
        int mn=INF,a=-1,b=-1;
        for(int i=0;i<n;i++)
        {
```

```
            if(vis[i])
            {
                for(int j=0;j<n;j++)
                {
                    if(!vis[j] && G[i][j])
                    {
                        if(G[i][j]<mn)
                        {
                            mn=G[i][j];
                            a=i;
                            b=j;
                        }
                    }
                }
            }
        }
        printf("%d - %d : %d\n",a,b,G[a][b]);
        tot+=G[a][b];
        vis[b]=1;
        e++;
    }
    printf("Total Weight = %d\n",tot);
}

void kruskal(int n,int G[20][20])
{
    for(int i=0;i<n;i++)
    {
        par[i]=i;
    }
    int e=0,tot=0;
    printf("\nEdges in MST (Kruskal's):\n");
    while(e<n-1)
    {
        int mn=INF,a=-1,b=-1;
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<n;j++)
            {
                if(find(i)!=find(j) && G[i][j]<mn)
                {
                    mn=G[i][j];
                    a=i;
                    b=j;
                }
            }
        }
        uni(a,b);
```

```
        printf("%d - %d : %d\n",a,b,mn);
        tot+=mn;
        e++;
    }
    printf("Total Weight = %d\n",tot);
}

int main()
{
    int n;
    printf("Enter number of vertices:");
    scanf("%d",&n);
    int G[20][20];
    printf("Enter adjacency matrix:\n");
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            scanf("%d",&G[i][j]);
            if(G[i][j]==0)
            {
                G[i][j]=INF;
            }
        }
    }

    clock_t s,e;
    double t1,t2;

    s=clock();
    prim(n,G);
    e=clock();
    t1=(double)(e-s)/CLOCKS_PER_SEC;

    s=clock();
    kruskal(n,G);
    e=clock();
    t2=(double)(e-s)/CLOCKS_PER_SEC;

    printf("\nTime taken for Prim's algorithm: %lf sec\n",t1);
    printf("Time taken for Kruskal's algorithm: %lf sec\n",t2);

    return 0;
}
```

Sample Output:

```
PS C:\Users\<Anmol> > & .\'exp6.exe'
Enter number of vertices:4
Enter adjacency matrix:
4 0 0 2
1 0 5 3
7 8 0 0
0 5 1 0

Edges in MST (Prim's):
0 - 3 : 2
3 - 2 : 1
3 - 1 : 5
Total Weight = 8

Edges in MST (Kruskal's):
1 - 0 : 1
3 - 2 : 1
0 - 3 : 2
Total Weight = 4

Time taken for Prim's algorithm: 0.00100
0 sec
Time taken for Kruskal's algorithm: 0.00
0000 sec
PS C:\Users\<Anmol> > (C:\Users\hp\anaco
nda3\Scripts\activate) ; (conda activate
 base)
PS C:\Users\<Anmol> >
```

Observations Table:

Vertices	Prim's Weight	Kruskal's Weight	Prim's Time (sec)	Kruskal's Time (sec)
4	8	4	0.001	0.00
10	31	31	0.001	0.0
100	245	245	0.01	0.02

Prim's best for dense graphs

Kruskal's best for sparse graphs

Complexity:

Algorithm	Time Complexity	Space Compleaxity
Prim's	O(V^2)	O(V)
Kruskal's	O(E log E)	O(V)

Result:
The program successfully computed the MST using both Prim's and Kruskal's algorithms for the given adjacency matrix. Execution times were measured using clock() function. For small graphs, both algorithms execute very fast. Prim's algorithm selected edges based on a growing MST from vertex 0, while Kruskal's algorithm selected edges in increasing order of weight using Union-Find.Prim's is faster for dense graphs and Kruskal's is faster for sparse graphs.

Conclusion:
Prim's and Kruskal's algorithms both correctly compute the Minimum Spanning Tree, though they may select different edges when multiple edges have equal weights. Prim's algorithm is generally more efficient for dense graphs, while Kruskal's performs better on sparse graphs due to its edge-sorting approach. For small graphs, execution times are negligible, but as graph size increases, the choice of algorithm can significantly affect performance, making both suitable for different scenarios depending on graph structure and density.

DAA Experiment No: 7

Experiment Name:

Shortest Path Algorithms using Dijkstra's and Bellman–Ford Algorithms

Aim:

To find the shortest path from a given source vertex to all other vertices in a weighted directed graph using Dijkstra's and Bellman–Ford algorithms and compare their execution times.

Theory Involved:

- **Single Source Shortest Path (SSSP):**
  The problem of finding the shortest distances from one source vertex to all other vertices in a weighted graph.
- **Dijkstra's Algorithm:**
  Works for graphs with non-negative edge weights.
  It repeatedly selects the vertex with the minimum tentative distance and relaxes all its adjacent edges.
- **Bellman–Ford Algorithm:**
  Works for graphs that may contain negative edge weights (but no negative cycles).
  It relaxes all edges (V–1) times, where V is the number of vertices.

Dijkstra's Algorithm:

- Initialize all distances as infinity except the source (0).
- Pick the unvisited vertex with the smallest distance.
- Update the distances of its adjacent vertices.
- Repeat until all vertices are processed.

Bellman–Ford Algorithm:

- Initialize all distances as infinity except the source (0).
- Relax all edges (V–1) times.
- If a shorter path is found through an edge, update the distance.
- Repeat for all edges.
- Optionally, check one more time for negative weight cycles.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define INF 999
void dijk(int n,int G[200][200],int src)
{
    int vis[200]={0},dis[200];
    for(int i=0;i<n;i++)
    {
        dis[i]=(G[src][i]==0)?INF:G[src][i];
    }
    dis[src]=0;
    vis[src]=1;
    for(int k=1;k<n-1;k++)
    {
        int mn=INF,u=-1;
        for(int i=0;i<n;i++)
        {
            if(!vis[i] && dis[i]<mn)
            {
                mn=dis[i];
                u=i;
            }
        }
        if(u==-1) break;
        vis[u]=1;
        for(int j=0;j<n;j++)
        {
            if(!vis[j] && G[u][j] && dis[u]+G[u][j]<dis[j])
            {
                dis[j]=dis[u]+G[u][j];
            }
        }
    }
}
void bell(int n,int e,int u[],int v[],int w[],int src)
{
    int dis[200];
    for(int i=0;i<n;i++)
    {
        dis[i]=INF;
    }
```

```
    dis[src]=0;
    for(int i=0;i<n-1;i++)
    {
        for(int j=0;j<e;j++)
        {
            if(dis[u[j]]+w[j]<dis[v[j]])
            {
                dis[v[j]]=dis[u[j]]+w[j];
            }
        }
    }
}
int main()
{
    int n,e,src;
    printf("Enter number of vertices:");
    scanf("%d",&n);
    printf("Enter number of edges:");
    scanf("%d",&e);
    printf("Enter source vertex:");
    scanf("%d",&src);
    int G[200][200]={0},u[e],v[e],w[e];
    srand(time(NULL));
    for(int i=0;i<e;i++)
    {
        u[i]=rand()%n;
        v[i]=rand()%n;
        if(u[i]==v[i])
        {
            v[i]=(v[i]+1)%n;
        }
        w[i]=(rand()%50)+1;
        G[u[i]][v[i]]=w[i];
    }
    printf("\nRandom graph generated with %d vertices and %d edges.\n",n,e);
    printf("(Weights are between 1 and 50)\n");
    clock_t st,en;
    double ms1,ms2;
    st=clock();
    for(int i=0;i<1000;i++)
    {
        dijk(n,G,src);
    }
    en=clock();
    ms1=(double)(en-st)*1000.0/CLOCKS_PER_SEC/1000.0;
    st=clock();
    for(int i=0;i<1000;i++)
    {
```

```
        bell(n,e,u,v,w,src);
    }
    en=clock();
    ms2=(double)(en-st)*1000.0/CLOCKS_PER_SEC/1000.0;
    printf("\nAverage time per run (1000 repeats):\n");
    printf("Dijkstra's algorithm: %.3f ms\n",ms1);
    printf("Bellman-Ford algorithm: %.3f ms\n",ms2);
    return 0;
}
```

Sample Output:

```
000 ms
PS C:\Users\<Anmol> > cd 'e:\sem 3\DAA\o
utput'
PS C:\Users\<Anmol> > & .\'exp7.exe'
Enter number of vertices:100
Enter number of edges:400
Enter source vertex:0

Random graph generated with 100 vertices
 and 400 edges.
(Weights are between 1 and 50)

Average time per run (1000 repeats):
Dijkstra's algorithm: 0.082 ms
Bellman-Ford algorithm: 0.118 ms
PS C:\Users\<Anmol> > shutdown /h
PS C:\Users\<Anmol> >
```

Observation Table:

Note:Source=0

Vertices	Edges	Dijkstra Time (ms)	Bellman-Ford Time (ms)
50	200	0.018	0.044
100	400	0.082	0.118
200	800	0.498	0.659

Complexity:

Algo	Time Complexity	Space Complexity	Remarks
Dijkstras	O(V^2)or(O((V+E) Log V) with heap)	O(V)	Faster, works only with non-negative edges
Bellman-Ford	O(VxE)	O(V)	Slower, works with negative weights

Result:

The program successfully computed the shortest paths using both Dijkstra's and Bellman–Ford algorithms.Execution times showed that Dijkstra's algorithm performed faster for all input sizes, especially in small graphs (50–200 vertices), confirming the theoretical efficiency difference.

Conclusion:

Dijkstra's algorithm is more efficient for graphs with non-negative weights, having lower execution time and complexity $O(V^2)$.
Bellman–Ford, though slower ($O(V \times E)$), supports negative edge weights and cycle detection.
Hence, Dijkstra's is preferred for normal weighted graphs, while Bellman–Ford suits graphs with negative weights.

DAA Experiment No: 8

Experiment Name:

All-Pairs Shortest Path using Floyd–Warshall Algorithm

Aim:

To determine the shortest distance between every pair of vertices in a weighted directed graph using the Floyd–Warshall algorithm, and to analyze its performance by measuring average execution time across multiple runs.

Theory Involved:

The Floyd–Warshall algorithm is a dynamic programming-based algorithm used for finding the shortest paths between all pairs of vertices in a weighted graph.
It systematically examines whether a vertex k can serve as an intermediate point on a shorter path between any two vertices i and j.
In simpler terms, the algorithm tries to improve the shortest known path between every pair of vertices by introducing one vertex at a time as a possible shortcut.
It repeatedly updates the distance matrix using the following rule:

```
if (G[i][k] + G[k][j] < G[i][j])
{
   G[i][j] = G[i][k] + G[k][j];
}
```

This means:

If the path from i to k to j is shorter than the current direct path i to j,
then update the distance with this new smaller value.

The algorithm performs this comparison for all vertices as intermediates, thereby ensuring that at the end of the process,
G[i][j] contains the minimum distance between every pair (i, j).

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void floyd(int n,int **G)
{
    for(int k=0;k<n;k++)
    {
```

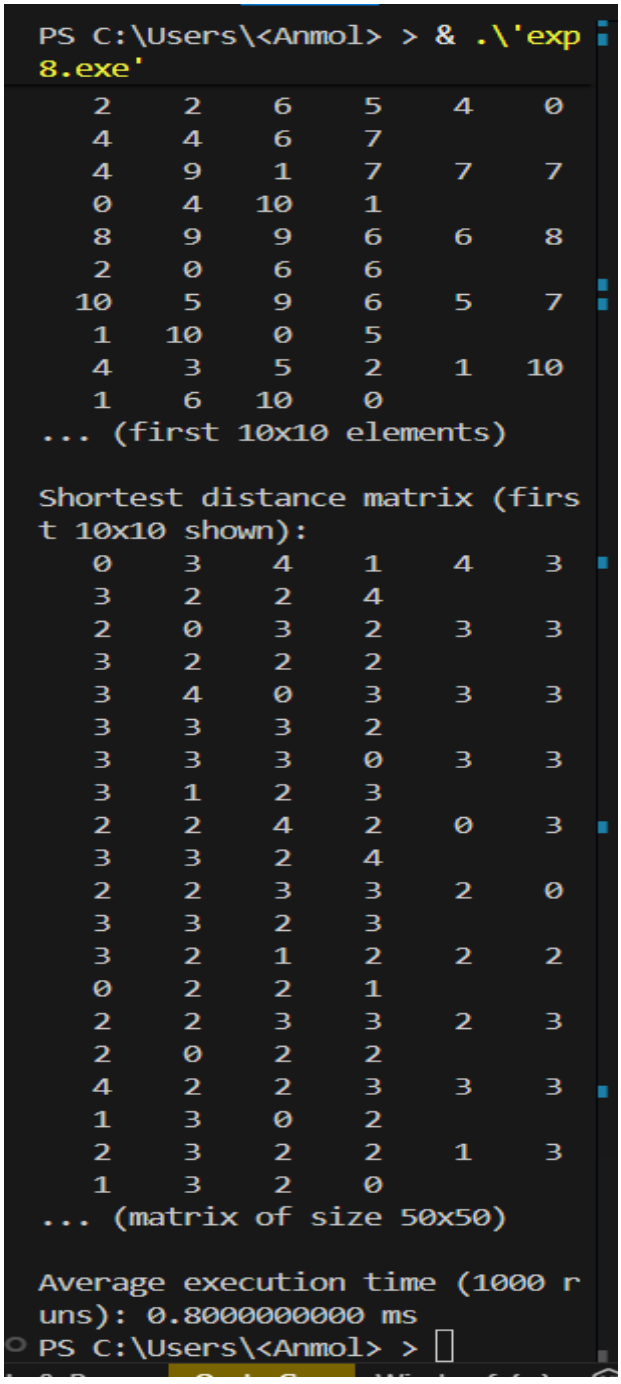
```
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<n;j++)
            {
                if(G[i][k]+G[k][j]<G[i][j])
                {
                    G[i][j]=G[i][k]+G[k][j];
                }
            }
        }
    }
}
int main()
{
    int n;
    printf("Enter number of vertices:");
    scanf("%d",&n);
    int **G=(int**)malloc(n*sizeof(int*));
    int **T=(int**)malloc(n*sizeof(int*));
    for(int i=0;i<n;i++)
    {
        G[i]=(int*)malloc(n*sizeof(int));
        T[i]=(int*)malloc(n*sizeof(int));
    }
    srand(time(NULL));
    printf("Adjacency matrix (fully connected, weights 1-10):\n");
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            if(i==j)
            {
                G[i][j]=0;
            }
            else
            {
                G[i][j]=(rand()%10)+1;
            }
            T[i][j]=G[i][j];
            if(i<10 && j<10)
            {
                printf("%4d ",G[i][j]);
            }
        }
        if(i<10)
        {
            printf("\n");
        }
```

```
    }
    if(n>10)
    {
        printf("... (first 10x10 elements)\n");
    }
    clock_t st=clock();
    for(int r=0;r<1000;r++)
    {
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<n;j++)
            {
                G[i][j]=T[i][j];
            }
        }
        floyd(n,G);
    }
    clock_t en=clock();

    double t=(double)(en-st)*1000.0/CLOCKS_PER_SEC/1000.0;

    printf("\nShortest distance matrix (first 10x10 shown):\n");
    for(int i=0;i<n && i<10;i++)
    {
        for(int j=0;j<n && j<10;j++)
        {
            printf("%4d ",G[i][j]);
        }
        printf("\n");
    }
    if(n>10)
    {
        printf("... (matrix of size %dx%d)\n",n,n);
    }
    printf("\nAverage execution time (1000 runs): %.10f ms\n",t);
    for(int i=0;i<n;i++)
    {
        free(G[i]);
        free(T[i]);
    }
    free(G);
    free(T);
    return 0;
}
```

Output:



Observation Table:

Vertices	Avg.Time(ms)
10	0.004
20	0.046

30	0.182
40	0.378
50	0.800

Complexity:

Algorithm	Time Complexity	Space Complexity
Floyd-Warshall	O(V^3)	O(V^2)

Result:

The program successfully computed the shortest paths between all vertex pairs using the Floyd–Warshall algorithm. The execution time increased rapidly from 0.004 ms at 10 vertices to 0.800 ms at 50 vertices, confirming its O(V^3) time complexity. Compared to Dijkstra's algorithm, Floyd–Warshall is slower but provides all-pairs shortest paths in one execution, making it more suitable for dense graphs where every vertex is connected.

Conclusion:

Floyd–Warshall and Dijkstra's algorithms both determine shortest paths but serve different purposes. Dijkstra's is faster and preferred for single-source shortest paths in large or sparse graphs, while Floyd–Warshall is simpler and ideal for dense or smaller graphs where all-pairs shortest path data is required. Thus, Floyd–Warshall offers completeness at the cost of speed, whereas Dijkstra's focuses on efficiency for specific sources.

DAA Experiment No: 9

Experiment Name:

0/1 Knapsack Problem using Dynamic Programming

Aim:

To determine the maximum profit possible for a set of items that can either be taken or left (0/1) using the Dynamic Programming approach to the Knapsack problem.

Theory Involved:

- The 0/1 Knapsack problem is a classic problem in combinatorial optimization.
- Each item has a value and a weight, and we must select items to maximize the total value without exceeding the bag's capacity.
- The Dynamic Programming approach divides the problem into smaller subproblems and stores their solutions to avoid recomputation.
- The recurrence relation used is:
  k[i][j] = max(val[i-1] + k[i-1][j - wt[i-1]], k[i-1][j])
  if the item fits in the capacity; otherwise:
  k[i][j] = k[i-1][j]
- The final answer (maximum profit) is stored in k[n][w].

Algorithm Steps:

- Create a DP table k[n+1][w+1].
- Initialize first row and column as 0 (base condition).
- Fill the table row by row using the above relation.
- The value in k[n][w] gives the maximum profit achievable.

Code:

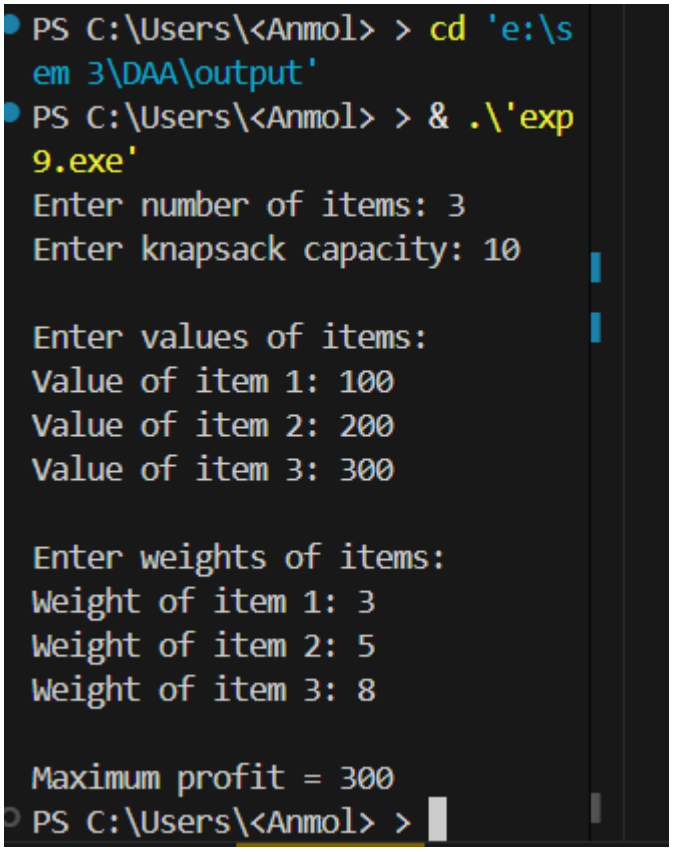
```c
// 0/1 Knapsack using Dynamic Programming
#include <stdio.h>
#include <stdlib.h>

int max(int a,int b)
{
    if(a>b)
        return a;
    else
        return b;
}
int knap(int n,int w,int val[],int wt[])
{
    int **k=(int**)malloc((n+1)*sizeof(int*));
```

```
    for(int i=0;i<=n;i++)
    {
        k[i]=(int*)malloc((w+1)*sizeof(int));
    }
    for(int i=0;i<=n;i++)
    {
        for(int j=0;j<=w;j++)
        {
            if(i==0||j==0)
            {
                k[i][j]=0;
            }
            else if(wt[i-1]<=j)
                {
                    k[i][j]=max(val[i-1]+k[i-1][j-wt[i-1]],k[i-1][j]);
                }
            else
                {
                    k[i][j]=k[i-1][j];
                }
        }
    }
    int res=k[n][w];
    for(int i=0;i<=n;i++)
    {
        free(k[i]);
    }
    free(k);
    return res;
}
int main()
{
    int n,w;
    printf("Enter number of items: ");
    scanf("%d",&n);
    printf("Enter knapsack capacity: ");
    scanf("%d",&w);
    int *val=(int*)malloc(n*sizeof(int));
    int *wt=(int*)malloc(n*sizeof(int));
    printf("\nEnter values of items:\n");
    for(int i=0;i<n;i++)
    {
        printf("Value of item %d: ",i+1);
        scanf("%d",&val[i]);
    }
    printf("\nEnter weights of items:\n");
    for(int i=0;i<n;i++)
    {
```

```
        printf("Weight of item %d: ",i+1);
        scanf("%d",&wt[i]);
    }
    int ans=knap(n,w,val,wt);
    printf("\nMaximum profit = %d\n",ans);
    free(val);
    free(wt);
    return 0;
}
```

Output:



Observation Table:

No. of Items	Capacity	Values (v)	Weights (w)	Maximum Profit
3	10	100,200,300	3,5,8	300
5	10	20,30,40,50,60	4,2,1,6,7	140
8	20	11,12,13,14,15,66,77,88	9,4,6,1,7,3,2,8	258

Complexity:

Parameter	Complexity
Time	O(n x W)
Space	O(n x W)

Result:

The program successfully calculated the maximum profit for different numbers of items and knapsack capacities using the Dynamic Programming approach**.**
As seen in the observations, the profit values increased with the number of items and capacity, confirming that the algorithm efficiently selects the best combination of items to maximize profit without exceeding the weight limit.

Conclusion:

The 0/1 Knapsack problem was successfully implemented using the Dynamic Programming approach.
The results clearly show that the algorithm efficiently computes the maximum profit for different item sets and bag capacities by storing and reusing subproblem results.
It eliminates redundant calculations and guarantees the optimal solution.
Thus, the Dynamic Programming method proves to be both accurate and efficient for solving the 0/1 Knapsack problem, especially for small to medium-sized inputs.

DAA Experiment No: 10

Experiment Name:
Naïve vs Rabin–Karp String Matching Using Rolling Hash

Aim:
To implement both Naïve and Rabin–Karp string matching algorithms, compare their working, observe how each method reports matches, and record their execution time for user-provided text and pattern.

Theory Involved:

Naïve String Matching:
The Naïve method checks the pattern at every possible position in the text.
It compares characters one by one until a mismatch or full match occurs.
Its worst-case time complexity is O(mn), where m is the pattern length and n is the text length.It is a straightforward approach and does not use any preprocessing or hashing.

Rabin–Karp String Matching:
Rabin–Karp uses a rolling hash function to quickly compare hash values of the pattern and windows of the text.If the hash values match, then a direct character comparison is done to confirm the match.It computes the hash of the first text window and updates the hash for each next window in O(1) time using a rolling hash formula.Average time complexity is O(n + m), but worst-case remains O(mn) when too many collisions occur.It is more efficient when searching multiple patterns at once.

Rolling Hash Details:
The implementation uses base D=256 and a prime modulus Q=101 for generating hash values.
This keeps the hash value small and helps reduce collisions.
The hash update formula for sliding the window is:
newHash=(D*(oldHash-text[s]*h)+text[s+m]) % Q
If the computed hash becomes negative, modulus is added to correct it.

Algorithm Steps:

Naïve Algorithm:

1. Shift pattern from position 0 to n–m.
2. Compare all characters of pattern with text window.
3. If all characters match, report the match location.

Rabin–Karp Algorithm:

1. Compute hash of the pattern and the first text window.
2. For each shift, compare the two hash values.
3. If hashes match, verify characters to confirm.

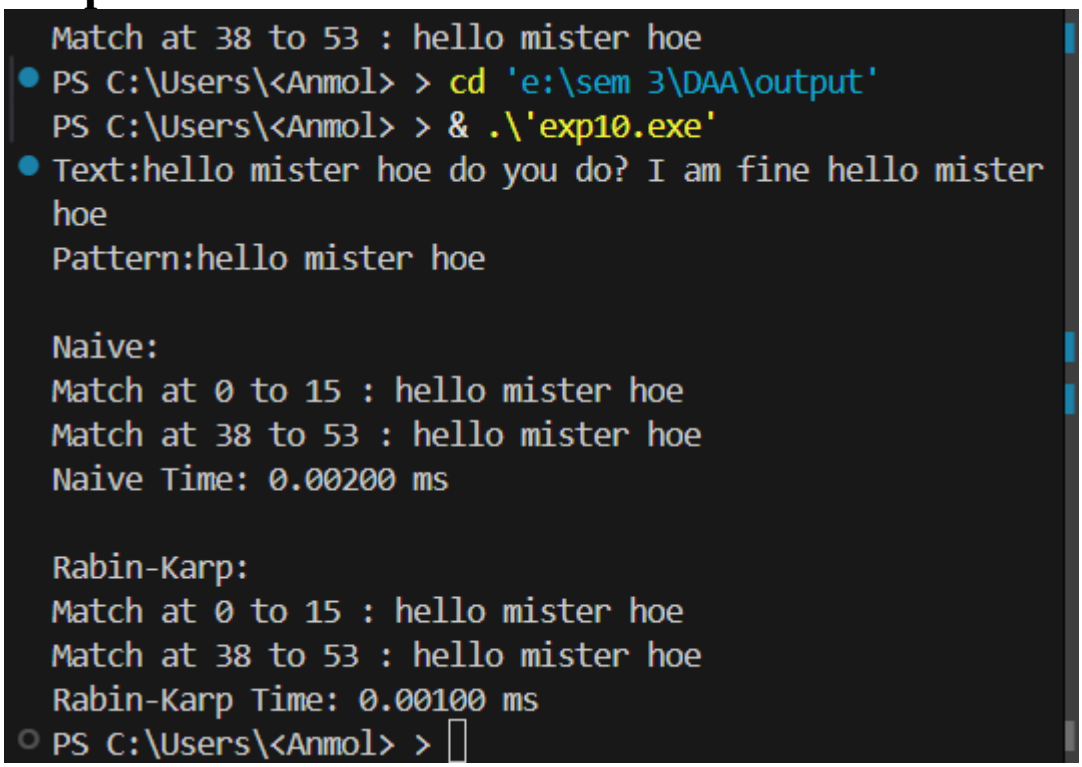
4. Slide window using the rolling hash formula.
5. If hash becomes negative, add modulus to fix it.

Code:

```c
// String Match Naive vs Rabin-Karp
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#define D 256
#define Q 101
char* getln()
{
    int c,sz=0,cap=16;
    char *s=malloc(cap);
    while((c=getchar())!='\n' && c!=EOF)
    {
        if(sz+1>=cap)
        {
            cap*=2;
            s=realloc(s,cap);
        }
        s[sz++]=c;
    }
    s[sz]=0;
    return s;
}
void prt(const char *t,int st,int m)
{
    printf("Match at %d to %d : ",st,st+m-1);
    for(int i=0;i<m;i++) printf("%c",t[st+i]);
    printf("\n");
}
void naive(const char *t,const char *p)
{
    int n=strlen(t),m=strlen(p);
    for(int s=0;s<=n-m;s++)
    {
        int j=0;
        while(j<m && t[s+j]==p[j]) j++;
        if(j==m) prt(t,s,m);
    }
}
void rk(const char *t,const char *p)
{
    int n=strlen(t),m=strlen(p);
    if(m>n) return;
    int h=1;
```

```
    for(int i=0;i<m-1;i++) h=(h*D)%Q;
    int ph=0,th=0;
    for(int i=0;i<m;i++)
    {
        ph=(ph*D+p[i])%Q;
        th=(th*D+t[i])%Q;
    }
    for(int s=0;s<=n-m;s++)
    {
        if(ph==th)
        {
            int ok=1;
            for(int i=0;i<m;i++)
                if(t[s+i]!=p[i]){ok=0;break;}
            if(ok) prt(t,s,m);
        }
        if(s<n-m)
        {
            th=(D*(th-t[s]*h)+t[s+m])%Q;
            if(th<0) th+=Q;
        }
    }
}
int main()
{
    printf("Text:");
    char *t=getln();
    printf("Pattern:");
    char *p=getln();
    clock_t a,b;
    double tn,tr;
    printf("\nNaive:\n");
    a=clock();
    naive(t,p);
    b=clock();
    tn=(double)(b-a)/1000.0;
    printf("Naive Time: %.5f ms\n",tn);
    printf("\nRabin-Karp:\n");
    a=clock();
    rk(t,p);
    b=clock();
    tr=(double)(b-a)/1000.0;
    printf("Rabin-Karp Time: %.5f ms\n",tr);
    free(t);
    free(p);
    return 0;
}
```

Output:



Observation Table:

Text Length n	Pattern Length m	Naïve (ms)	Rabin–Karp (ms)
100 Words	8 Letters	0.004	0.002
200 Words	4 Letters	0.004	0.003
300 Words	9 Letters	0.013	0.008

Complexity:

Algorithm	Time Complexity	Space Complexity
Naïve	O(n*m)	O(1)
Rabin–Karp	O(n+m)	O(1)

Result:

Both Naïve and Rabin–Karp algorithms were executed on text inputs of 100, 200, and 300 words with different pattern lengths.For smaller inputs (100–200 words) both algorithms showed almost equal time.For larger input (300 words) Rabin–Karp performed faster due to its rolling-hash mechanism.Naïve took 0.004 ms, 0.004 ms, and 0.013 ms respectively, while Rabin–Karp took 0.002 ms, 0.003 ms, and 0.008 ms.This matches the theoretical behaviour:

Naïve grows with O(n·m) while Rabin–Karp grows with O(n+m).Both algorithms used O(1) extra space.

Conclusion:

Naïve gives correct results but becomes slower as text size increases because it compares characters at every shift.Rabin–Karp remains consistently faster for bigger inputs because of its efficient rolling hash, reducing unnecessary comparisons.For small text sizes, the difference is minor, but for larger inputs the improvement in Rabin–Karp is clearly visible.Thus, the experiment confirms that Naïve is simple but inefficient for large data, while Rabin–Karp is more scalable and performs better as n increases.