01 KnapSack Dynamic

```c
#include<stdio.h>

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

int main(){

    int profits[] = {0,60,100,120};
    int weights[] = {0,10,20,30};

    /* IMPORTANT :  Add 0 as first entry in both the Arrays and then proceed as usual*/
    /* Number of objects = actual objects(inserted apart from 0 entry)*/

    int objects = 3;
    int capacity_constraint = 50;

    int Table[objects+1][capacity_constraint+1];

    for(int i=0 ;i<= objects;i++) {

        for(int w=0 ; w <= capacity_constraint ; w++){

            /*3 cases exist
            1) Fill row 0 and col 0 with Zero
            2) When weight of Object is less than a column use Formula
            3) Copy the entry from previous row
            */

             if(i==0 || w==0){
                 Table[i][w]=0;
             }
            else if(weights[i]<=w){
                 Table[i][w] = max(Table[i-1][w] , Table[i-1][w-weights[i]] + profits[i]);
             }

             else {
                 /* Weight of an Object is Greater*/
                 Table[i][w] = Table[i-1][w];
             }
        }
    }

               for(int i=0 ; i<objects+1;i++){
                for(int j=0;j<capacity_constraint+1 ; j++){


                    printf(" %d ",Table[i][j]);
                }
                printf("\n");
            }


      printf(" Max Profit : %d ",Table[objects+1][capacity_constraint+1]);



}
```

Matrix Multiplication

```c
#include <stdio.h>

int main()
{
    int a[10][10], b[10][10], result[10][10], r1, c1, r2, c2, i, j, k;

    printf("Enter rows and column for first matrix: ");
    scanf("%d %d", &r1, &c1);

    printf("Enter rows and column for second matrix: ");
    scanf("%d %d",&r2, &c2);

    // Column of first matrix should be equal to column of second matrix and
    while (c1 != r2)
    {
        printf("Error! column of first matrix not equal to row of
second.\n\n");
        printf("Enter rows and column for first matrix: ");
        scanf("%d %d", &r1, &c1);
        printf("Enter rows and column for second matrix: ");
        scanf("%d %d",&r2, &c2);
    }

    // Storing elements of first matrix.
    printf("\nEnter elements of matrix 1:\n");
    for(i=0; i<r1; ++i)
        for(j=0; j<c1; ++j)
        {
            printf("Enter elements a%d%d: ",i+1, j+1);
            scanf("%d", &a[i][j]);
        }

    // Storing elements of second matrix.
    printf("\nEnter elements of matrix 2:\n");
    for(i=0; i<r2; ++i)
        for(j=0; j<c2; ++j)
        {
            printf("Enter elements b%d%d: ",i+1, j+1);
            scanf("%d",&b[i][j]);
        }

    // Initializing all elements of result matrix to 0
    for(i=0; i<r1; ++i)
        for(j=0; j<c2; ++j)
        {
            result[i][j] = 0;
        }

    // Multiplying matrices a and b and
    // storing result in result matrix
    for(i=0; i<r1; ++i)
        for(j=0; j<c2; ++j)
            for(k=0; k<c1; ++k)
            {
                result[i][j]+=a[i][k]*b[k][j];
            }

    // Displaying the result
    printf("\nOutput Matrix:\n");
    for(i=0; i<r1; ++i)
```

```
        for(j=0; j<c2; ++j)
        {
            printf("%d  ", result[i][j]);
            if(j == c2-1)
                printf("\n\n");
        }
    return 0;
}
```

Fractional KnapSack : Greedy

```c
#include<stdio.h>

#define num_obj 7


void init_arrays(float* p_by_w,int n,float* profit,float* weight,float* solutions){
    for(int i=0;i<n;i++){
        p_by_w[i]=profit[i]/weight[i];
        solutions[i]=0;
    }
}

int main(){
    float profits[]={10,5,15,7,6,18,3};
    float weights[]={2,3,5,7,1,4,1};
    float profit_by_weight[num_obj];
    float solution[num_obj];

    init_arrays(profit_by_weight,num_obj,profits,weights,solution);


    int available_capacity=15;



    for(int i=0;i<num_obj && available_capacity>0 ;i++){

        int greatest=0;
        int index = 0;

        for(int j=0 ;j<num_obj;j++)
        {
            if(profit_by_weight[j]>greatest && solution[j]==0){
                index = j;
                greatest = profit_by_weight[j];
            }
        }

        if(available_capacity >= weights[index]){
            solution[index]=1;
            available_capacity = available_capacity - weights[index];
        }

        /* Hanlde Fractionals */

        else if(available_capacity < weights[index]){
            float temp =  available_capacity/weights[index];
            solution[index]=temp;
            available_capacity =- weights[index];
        }
    }

    for(int i=0;i<num_obj;i++){
        printf(" %f ",solution[i]);
    }

}
```

Matrix Chain Multiplication Dynamic Approach

```c
#include<stdio.h>

int main(){

    // D - Dimensions Array
    int D[] = {5,4,6,2,7};

    int Matrix[10][10]={0};
    int Split[10][10]={0};

    int n = 5; // Elements in Dimension Array
    /*
    Matrix - Matrix Cost Matrix
    Split  - Split Matrix stores where does the split occur
    */

   /* we need to find the cost for Matrix Multiplication in following Order
       we need to find diagonal elements for upper triangle diagonals
   */
    int j,min,q;

    for(int d=1 ; d < n-1 ; d++){

        for(int i=1 ; i < n-d ; i++){

            // i -> Rows , j is calculated using i and d

            j= i+d;
            min = 32767;
            Matrix[i][j]=min;


            for(int k=1 ; k<= j-1; k++){
                q = Matrix[i][k] + Matrix[k+1][j] + D[i-1] * D[k] * D[j];

                if(q<Matrix[i][j]){
                    min = q;
                    Split[i][j] = k;
                }

                Matrix [i][j] = min;
            }

        }
    }


    printf(" Answer : %d \n",Matrix[1][n-1]);

    for(int i=1 ; i<n ;i++){

        for(int j=1 ; j<n ;j++)
        {
            printf(" %d ",Matrix[i][j]);
        }

        printf("\n");
    }
}
```

Merge Sort : DAC

```c
#include<stdio.h>


void Merge(int * Array,int low,int mid,int high){
    int i=low;int j=mid+1;int k=0;
    int TempArray[25];

    while( (i<=mid) && (j<=high) ){
        if(Array[i]<Array[j]){
            TempArray[k]=Array[i];
            i++;k++;
        }
        else if(Array[i]>Array[j]){
            TempArray[k]=Array[j];
            j++;k++;
        }

        else {
            TempArray[k]=Array[i];
            i++;j++;k++;
        }
    }

    while(i<=mid){
        TempArray[k]=Array[i];
        i++;k++;
    }

    while(j<=high){
        TempArray[k]=Array[j];
        j++;k++;
    }

    /* Copy Back */
    k=0;
    for(int i=low;i<=high;i++,k++){
        Array[i]=TempArray[k];
    }

}
void MergeSort(int* Array,int low,int high){
    if(low<high){
        int mid = (low+high)/2;
        MergeSort(Array,low,mid);
        MergeSort(Array,mid+1,high);
        Merge(Array,low,mid,high);
    }
}

int main(){
    int trial[]={50,30,40,10,20,0};
    MergeSort(trial,0,5);

    for(int i=0;i<6;i++){
        printf(" %d ",trial[i]);
    }
}
```

Naive String Matching

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

int main(){

     char String[20];
     char Pattern[20];

     printf("Enter the String ? \n");
     scanf("%s",String);

     printf("Enter the Pattern in String : ' %s ' \n",String);
     scanf("%s",Pattern);

        int flag=0;

        int string_len = strlen(String);
        int pattern_len = strlen(Pattern);

     for(int i=0;i<string_len;i++){

         int j; /* Note declared outside its loop */
         for(j=0;j<pattern_len;j++){
             if(String[i+j] != Pattern[j])
             break;
         }

         if(j == pattern_len){
             flag=1;
             printf("Pattern has been found at index : %d \n",i);
         }
     }

        if(!flag){
            printf("Pattern didnt match anywhere");
        }

}
```

Backtracking : N Queens Problem

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



bool isSafetoPlace(int board[][10],int row,int column,int n){
    /*Queen will be safe to place if and only if
        1) No Other Queen is Placed in same column -to find itterate over previous
        2) No Other Queen is Placed in Right Upper Diagonal
        3) No Other Queen is Placed in Left Upper Diagonal
    */
    // 1 - Same Column Check
    for(int i=0 ;i<row;i++){
        if(board[i][column]==1){
            /* Queen already placed in some previous row at the column we are checking for . Thus we cannot place so return false*/
            return false;
        }
    }



    // 2 - Right Upper Diagonal

        int x=row;
        int y=column;

        while(x>=0 && y<n){
            if(board[x][y]==1)
            {return false;}
            x--;
            y++;
        }

    //3 - Left Upper Diagonal

        x=row;
        y=column;

        while(x>=0 && y>=0){
            if(board[x][y]==1)
            {return false;}
            x--;
            y--;
        }
    /* if control reaches upto here it means , these conditions are not satisfied and Queen is safe to be placed*/

    return true;
}

void DisplayBoard(int board[][10],int n){
    for(int i=0;i<n;i++){

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

            if(board[i][j]==1){
                printf(" Q ");
            }
```

```
            else {
                printf(" _ ");
            }
        }

        printf("\n\n");
    }
}

bool solveNQueen(int board[][10],int current_row,int n){
    /*Base Case: successfully placed Queens in N rows */
    if(current_row == n){
        /* Print Board Config*/
        printf("\n\n ************* \n\n");
        DisplayBoard(board,n);
        return false;
    }

    /* Recursie Case: Try to find the the right column in current row*/
    for(int current_column=0;current_column<n;current_column++){

        if(isSafetoPlace(board,current_row,current_column,n)){
            // Place the Queen assuming this is correct position
            board[current_row][current_column]=1;

            /* Ask the Remaining Board if next Queen can be Placed in Next Row*/
            bool canNextQueenBePlaced = solveNQueen(board,current_row+1,n);
            if(canNextQueenBePlaced){
                /* it means we have placed Queen above is Right and need not be shifted*/
                return true;
            }

            /* if next queen cannot be placed , our above assumption is wrong and needs to be corrected , Queen needs to be shifted in the current row and possibiliites need to be recalculated*/
            // Backtrack
            else board[current_row][current_column]=0;
        }

    }

    // At this control point we have tried all possible positions in current row but have failed to place a Queen, hence return False

    return false;

}

int main(){

    /* Initialize board to 0 */
    int board[10][10]={0};


    solveNQueen(board,0,4);


}
```

QuickSort : DAC

```c
#include<stdio.h>

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

int Partition(int Array[],int low,int high){
    int pivot = Array[low];
    int i=low-1;
    int j=high+1;

    while(i<j){
        do{
            i++;
        }while(Array[i]<=pivot);

        do{
            j--;
        }while(Array[j]>pivot);

        if(i<j){
            swap(Array,i,j);
        }
    }

    swap(Array,low,j);
    return j;
}
void QuickSort(int Array[],int low,int high){

    if(low<high){

    int partition_index = Partition(Array,low,high);
    QuickSort(Array,low,partition_index-1);
    QuickSort(Array,partition_index+1,high);

    }
}

int main(){
    int Array[7]={50,20,30,10,5,60,70};
    QuickSort(Array,0,6);

    for(int i=0;i<7;i++){
        printf(" %d ",Array[i]);
    }
}
```