2. Fractional knapsack

```c
#include <stdio.h>
int n = 5;
int p[10] = {3, 3, 2, 5, 1}; //weight
int w[10] = {10, 15, 10, 12, 8};//profit
int W = 10;
int main(){
   int cur_w;
   float tot_v;
   int i, maxi;
   int used[10];
   for (i = 0; i < n; ++i)
      used[i] = 0;
   cur_w = W;
   while (cur_w > 0) {
      maxi = -1;
      for (i = 0; i < n; ++i)
         if ((used[i] == 0) &&
               ((maxi == -1) || ((float)w[i]/p[i] > (float)w[maxi]/p[maxi])))
            maxi = i;
      used[maxi] = 1;
      cur_w -= p[maxi];
      tot_v += w[maxi];
      if (cur_w >= 0)
         printf("Added object %d (%d, %d) completely in the bag. Space left:
%d.\n", maxi + 1, w[maxi], p[maxi], cur_w);
      else {
         printf("Added %d%% (%d, %d) of object %d in the bag.\n", (int)((1 +
(float)cur_w/p[maxi]) * 100), w[maxi], p[maxi], maxi + 1);
         tot_v -= w[maxi];
         tot_v += (1 + (float)cur_w/p[maxi]) * w[maxi];
      }
   }
   printf("Filled the bag with objects worth %.2f.\n", tot_v);
   return 0;
}
```

3. Job Sequencing

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

// A structure to represent a Jobs
typedef struct Jobs {
   char id; // Jobs Id
   int dead; // Deadline of Jobs
   int profit; // Profit if Jobs is over before or on deadline
} Jobs;

// This function is used for sorting all Jobs according to
// profit
int compare(const void* a, const void* b){
   Jobs* temp1 = (Jobs*)a;
   Jobs* temp2 = (Jobs*)b;
   return (temp2->profit - temp1->profit);
}

// Find minimum between two numbers.
int min(int num1, int num2){
   return (num1 > num2) ? num2 : num1;
}
int main(){
   Jobs arr[] = {
      { 'a', 2, 100 },
      { 'b', 2, 20 },
      { 'c', 1, 40 },
      { 'd', 3, 35 },
      { 'e', 1, 25 }
   };
   int n = sizeof(arr) / sizeof(arr[0]);
   printf("Following is maximum profit sequence of Jobs: \n");
   qsort(arr, n, sizeof(Jobs), compare);
   int result[n]; // To store result sequence of Jobs
   bool slot[n]; // To keep track of free time slots

   // Initialize all slots to be free
   for (int i = 0; i < n; i++)
      slot[i] = false;
```

```
   // Iterate through all given Jobs
   for (int i = 0; i < n; i++) {

      // Find a free slot for this Job
      for (int j = min(n, arr[i].dead) - 1; j >= 0; j--) {

         // Free slot found
         if (slot[j] == false) {
            result[j] = i;
            slot[j] = true;
            break;
         }
      }
   }

   // Print the result
   for (int i = 0; i < n; i++)
      if (slot[i])
         printf("%c ", arr[result[i]].id);
   return 0;
}
```

10. Floyd Warshall's strategy

```
// Floyd-Warshall Algorithm in C

#include <stdio.h>

// defining the number of vertices
#define nV 4

#define INF 999

void printMatrix(int matrix[][nV]);

// Implementing floyd warshall algorithm
void floydWarshall(int graph[][nV]) {
  int matrix[nV][nV], i, j, k;

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

```
    for (j = 0; j < nV; j++)
      matrix[i][j] = graph[i][j];

  // Adding vertices individually
  for (k = 0; k < nV; k++) {
    for (i = 0; i < nV; i++) {
      for (j = 0; j < nV; j++) {
        if (matrix[i][k] + matrix[k][j] < matrix[i][j])
          matrix[i][j] = matrix[i][k] + matrix[k][j];
      }
    }
  }
  printMatrix(matrix);
}

void printMatrix(int matrix[][nV]) {
  for (int i = 0; i < nV; i++) {
    for (int j = 0; j < nV; j++) {
      if (matrix[i][j] == INF)
        printf("%4s", "INF");
      else
        printf("%4d", matrix[i][j]);
    }
    printf("\n");
  }
}

int main() {
  int graph[nV][nV] = {{0, 3, INF, 5},
              {2, 0, INF, 4},
              {INF, 1, 0, INF},
              {INF, INF, 2, 0}};
  floydWarshall(graph);
}
```

KNAPSACK 0/1:

```
#include<stdio.h>
int max(int a, int b) {
   if(a>b){
      return a;
   } else {
      return b;
```

```
    }
}
int knapsack(int W, int wt[], int val[], int n) {
   int i, w;
   int knap[n+1][W+1];
   for (i = 0; i <= n; i++) {
      for (w = 0; w <= W; w++) {
         if (i==0 || w==0)
            knap[i][w] = 0;
         else if (wt[i-1] <= w)
            knap[i][w] = max(val[i-1] + knap[i-1][w-wt[i-1]],
knap[i-1][w]);
         else
            knap[i][w] = knap[i-1][w];
      }
   }
   return knap[n][W];
}
int main() {
   int val[] = {20, 25, 40};
   int wt[] = {25, 20, 30};
   int W = 50;
   int n = sizeof(val)/sizeof(val[0]);
   printf("The solution is : %d", knapsack(W, wt, val, n));
   return 0;
}
```

// C code to implement the
// matrix chain multiplication using recursion

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

// Matrix Ai has dimension p[i-1] x p[i]
// for i = 1 . . . n
int MatrixChainOrder(int p[], int i, int j)

```
{
	if (i == j)
		return 0;
	int k;
	int min = INT_MAX;
	int count;

	// Place parenthesis at different places
	// between first and last matrix,
	// recursively calculate count of multiplications
	// for each parenthesis placement
	// and return the minimum count
	for (k = i; k < j; k++)
	{
		count = MatrixChainOrder(p, i, k)
				+ MatrixChainOrder(p, k + 1, j)
				+ p[i - 1] * p[k] * p[j];

		if (count < min)
			min = count;
	}

	// Return minimum count
	return min;
}

// Driver code
int main()
{
	int arr[] = { 1, 2, 3, 4, 3 };
	int N = sizeof(arr) / sizeof(arr[0]);
```

```
        // Function call
        printf("Minimum number of multiplications is %d ",
               MatrixChainOrder(arr, 1, N - 1));
        getchar();
        return 0;
    }
```

Bellman ford

5.

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

#define MAX_VERTICES 1000
#define MAX_EDGES 1000

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    struct Edge edges[MAX_EDGES];
};

void bellmanFord(struct Graph* graph, int src) {
    int V = graph->V;
    int E = graph->E;
    int dist[MAX_VERTICES];

    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edges[j].src;
            int v = graph->edges[j].dest;
            int weight = graph->edges[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
```

```
            dist[v] = dist[u] + weight;
        }
    }

    for (int i = 0; i < E; i++) {
        int u = graph->edges[i].src;
        int v = graph->edges[i].dest;
        int weight = graph->edges[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle\n");
            return;
        }
    }

    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d\t\t%d\n", i, dist[i]);
}

int main() {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = 5;
    graph->E = 8;

    graph->edges[0].src = 0;
    graph->edges[0].dest = 1;
    graph->edges[0].weight = -1;

    graph->edges[1].src = 0;
    graph->edges[1].dest = 2;
    graph->edges[1].weight = 4;

    graph->edges[2].src = 1;
    graph->edges[2].dest = 2;
    graph->edges[2].weight = 3;

    graph->edges[3].src = 1;
    graph->edges[3].dest = 3;
    graph->edges[3].weight = 2;

    graph->edges[4].src = 1;
    graph->edges[4].dest = 4;
    graph->edges[4].weight = 2;

    graph->edges[5].src = 3;
    graph->edges[5].dest = 2;
    graph->edges[5].weight = 5;
```

```
    graph->edges[6].src = 3;
    graph->edges[6].dest = 1;
    graph->edges[6].weight = 1;

    graph->edges[7].src = 4;
    graph->edges[7].dest = 3;
    graph->edges[7].weight = -3;

    bellmanFord(graph, 0);

    free(graph);
    return 0;
}
```

1.

```
#include <stdio.h>
void heapify (int a[],int n, int i, int* steps)
{
    int max, left, right;
    max=i;
    left=2*i+1;
    right=2*i+2;
    if (left<n && a[left]>a[max]) max=left; (*steps)++;
    if (right<n && a[right]>a[max]) max=right;  (*steps)++;
    if (max==i) return;
    int temp=a[i];
    a[i]=a[max];
    a[max]=temp;
    (*steps)++;
    heapify(a,n,max, steps);
}
void heapsort (int a[], int n, int* steps)
{
    int i;
    for (i=n/2-1;i>=0;i--)
        heapify(a,n,i, steps);
    for (i=n-1;i>=0;i--)
    {
        int temp=a[i];
        a[i]=a[0];
        a[0]=temp;
        (*steps)++;
        heapify(a,i,0, steps);
    }
}

void merge(int a[], int begin, int mid, int end, int* steps)
{
```

```
    int i, j, k;
    int n1 = mid - begin + 1;
    int n2 = end - mid;
    int la[n1], ra[n2];
    for (i = 0; i < n1; i++)
    {
        (*steps)++;
        la[i] = a[begin + i];
    }
    for (j = 0; j < n2; j++)
    {
        (*steps)++;
        ra[j] = a[mid + 1 + j];
    }
    i = j = 0;
    k = begin;
    while (i < n1 && j < n2)
    {
        if (la[i] <= ra[j])
            a[k++] = la[i++];
        else
        {
            a[k++] = ra[j++];
            (*steps)++;
        }
    }
    while (i < n1)
    {
        a[k++] = la[i++];
        (*steps)++;
    }
    while (j < n2)
    {
        a[k++] = ra[j++];
        (*steps)++;
    }
}
void mergeSort(int a[], int begin, int end, int *steps)
{
    if (begin < end)
    {
        int mid = (begin + end) / 2;
        mergeSort(a, begin, mid, steps);
        mergeSort(a, mid + 1, end, steps);
        merge(a, begin, mid, end, steps);
    }
}
```

```
int partition(int a[], int begin, int end, int *steps)
{
    int pivot = a[end];
    int i, j = begin;
    for (i = begin; i < end; i++)
    {
        (*steps)++;
        if (a[i] < pivot) {
            int t = a[j];
            a[j] = a[i];
            a[i] = t;
            j++;
        }
    }
    (*steps)++;
    int t = a[j];
    a[j] = a[end];
    a[end] = t;
    return j;
}

int quickSort(int a[], int start, int end, int *steps)
{
    if (start < end)
    {
        int p = partition(a, start, end - 1, steps);
        quickSort(a, start, p, steps);
        quickSort(a, p + 1, end, steps);
    }
}

//Print for testing only
void printarray (int a[], int n)
{
    int i;
    for (i=0;i<n;i++)
    printf("%d \n",a[i]);
}

int main()
{
    int steps = 0, n = 1000, i, c;
    FILE *file;
    file = fopen("random.txt", "r");
    char temp[6];
    int a[n];
    fgets(temp, sizeof(temp), file);
    for (i = 0; i < n; i++)
```

```
    {
        fgets(temp, sizeof(temp), file);
        a[i] = atoi(temp);
    }
    fclose(file);
    printf("Enter which type of sorting you want to do\n");
    printf("1 for Merge Sort, 2 for Quick Sort, 3 for Heap Sort: ");
    scanf("%d", &c);
    if (c == 1)
    {
        mergeSort(a, 0, n - 1, &steps);
        printarray(a,n);
    }
    else if (c == 2)
    {
        quickSort(a, 0, n, &steps);
        printarray(a,n);
    }
    else if (c == 3)
    {
        heapsort(a, n, &steps);
        printarray(a,n);
    }
    else
        printf("Wrong Choice!");
    printf("Steps= %d", steps);
    return 0;
}
```

https://github.com/Abhiroop2004/Design-and-Analysis-of-Algorithms-Lab-Assignments/blob/main/sorting.c

```
#include <bits/stdc++.h>

using namespace std;

void lcs(string s1, string s2) {

  int n = s1.size();
  int m = s2.size();

  vector < vector < int >> dp(n + 1, vector < int > (m + 1, 0));
  for (int i = 0; i <= n; i++) {
    dp[i][0] = 0;
  }
  for (int i = 0; i <= m; i++) {
    dp[0][i] = 0;
  }

  for (int ind1 = 1; ind1 <= n; ind1++) {
    for (int ind2 = 1; ind2 <= m; ind2++) {
```

```
      if (s1[ind1 - 1] == s2[ind2 - 1])
        dp[ind1][ind2] = 1 + dp[ind1 - 1][ind2 - 1];
      else
        dp[ind1][ind2] = 0 + max(dp[ind1 - 1][ind2], dp[ind1][ind2 - 1]);
    }
  }

  int len = dp[n][m];
  int i = n;
  int j = m;

  int index = len - 1;
  string str = "";
  for (int k = 1; k <= len; k++) {
    str += "$"; // dummy string
  }

  while (i > 0 && j > 0) {
    if (s1[i - 1] == s2[j - 1]) {
      str[index] = s1[i - 1];
      index--;
      i--;
      j--;
    } else if (s1[i - 1] > s2[j - 1]) {
      i--;
    } else j--;
  }
  cout << str;
}

int main() {

  string s1 = "abcde";
  string s2 = "bdgek";

  cout << "The Longest Common Subsequence is ";
  lcs(s1, s2);
}
```