1. Write a C program to compute Ranksort of array element	

Input
`//Given an unsorted array you need to fetch the indices in ascending order.`
`//example: 5,3,9, 8,2`
`#include <stdio.h>`
`#define INT_MAX 1000`
 	
`void sortIndex(int arr[], int n){`
 	
`int arr2[n],min, k;`
 	
`for(int i = 0; i<n; i++)`
`arr2[i] = 0;`
 	
`for(int i = 0; i<n; i++){`
`for(k=0;arr2[k]!=0;k++);`
`min = k;`
`for(int j = 0; j < n; j++){`
`if(arr[j]<arr[min] && arr2[j]==0){`
`min = j;`
`}`
`}`
`arr2[min]=1;`
`printf("%d\t", min);`
`}`
 	
`}`
`int main(){`
`int arr[] = {5,3,9,8,2};`
`int n = sizeof(arr)/sizeof(arr[0]);`
`sortIndex(arr,n);`
`return 0;`
`}`
	Output
	
	4 1 0 3 2
	
	=== Code Execution Successful ===

2. Write a C program to compute Greedy Knapsack problem	

Input
`#include <stdio.h>`
`#include <stdlib.h>`
 	
`typedef struct {`
`float weight;`
`float value;`
`float ratio;`
`} Item;`
 	
`int compare(const void *a, const void *b) {`
`Item *itemA = (Item *)a;`
`Item *itemB = (Item *)b;`
`if (itemB->ratio > itemA->ratio) return 1;`
`else if (itemB->ratio < itemA->ratio) return -1;`
`else return 0;`
`}`
 	
`float greedyKnapsack(Item items[], int n, float W) {`
 	
`qsort(items, n, sizeof(Item), compare);`
 	
`float currentWeight = 0.0, maxValue = 0.0;`
	Output
	
	Enter the number of items: 5
	Enter the maximum weight of the knapsack: 80
	Enter the weights and values of the items:
	Item 1 (Weight Value): 10
	30
	Item 2 (Weight Value):
	10 30
	Item 3 (Weight Value): 50 70
	Item 4 (Weight Value): 8 90
	Item 5 (Weight Value): 10 20
	The maximum value that can be obtained is: 228.80
	
	=== Code Execution Successful ===

```
    for (int i = 0; i < n; i++) {
        if (currentWeight + items[i].weight <= W) {

            currentWeight += items[i].weight;
            maxValue += items[i].value;
        } else {

            float remainingWeight = W - currentWeight;
            maxValue += items[i].value * (remainingWeight /
items[i].weight);
            break;
        }
    }

    return maxValue;
}

int main() {
    int n;
    float W;

    printf("Enter the number of items: ");
    scanf("%d", &n);
    printf("Enter the maximum weight of the knapsack: ");
    scanf("%f", &W);


    Item items[n];
    printf("Enter the weights and values of the items:\n");
    for (int i = 0; i < n; i++) {
        printf("Item %d (Weight Value): ", i + 1);
        scanf("%f %f", &items[i].weight, &items[i].value);
        items[i].ratio = items[i].value / items[i].weight; // Calculate
ratio
    }

    float maxValue = greedyKnapsack(items, n, W);
    printf("The maximum value that can be obtained is: %.2f\n",
maxValue);

    return 0;
}
```

3. Write a C program to compute Greedy Job sequencing.

Input

```
#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 Jobss according to
```

Output

Following is maximum profit sequence of Jobs:
c a d

```c
// 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;
}
```

4. Write a C program to compute Ranksort of array element

Input	**Output**
// The longest common subsequence in C #include <stdio.h>	S1 : ACADB S2 : CBDA LCS: CB

```c
#include <string.h>

int i, j, m, n, LCS_table[20][20];
char S1[20] = "ACADB", S2[20] = "CBDA", b[20][20];

void lcsAlgo() {
  m = strlen(S1);
  n = strlen(S2);

  // Filling 0's in the matrix
  for (i = 0; i <= m; i++)
    LCS_table[i][0] = 0;
  for (i = 0; i <= n; i++)
    LCS_table[0][i] = 0;

  // Building the mtrix in bottom-up way
  for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++) {
      if (S1[i - 1] == S2[j - 1]) {
        LCS_table[i][j] = LCS_table[i - 1][j - 1] + 1;
      } else if (LCS_table[i - 1][j] >= LCS_table[i][j - 1]) {
        LCS_table[i][j] = LCS_table[i - 1][j];
      } else {
        LCS_table[i][j] = LCS_table[i][j - 1];
      }
    }

  int index = LCS_table[m][n];
  char lcsAlgo[index + 1];
  lcsAlgo[index] = '\0';

  int i = m, j = n;
  while (i > 0 && j > 0) {
    if (S1[i - 1] == S2[j - 1]) {
      lcsAlgo[index - 1] = S1[i - 1];
      i--;
      j--;
      index--;
    }

    else if (LCS_table[i - 1][j] > LCS_table[i][j - 1])
      i--;
    else
      j--;
  }

  // Printing the sub sequences
  printf("S1 : %s \nS2 : %s \n", S1, S2);
  printf("LCS: %s", lcsAlgo);
}

int main() {
  lcsAlgo();
  printf("\n");
}
```

5. Write a C program to compute Fibonacci series using dynamic programming.	
Input	**Output**

<pre>int Fibonacci(int N)
{
    int Fib[N+1],i;
 	
    //we know Fib[0] = 0, Fib[1]=1
    Fib[0] = 0;
    Fib[1] = 1;
 	
    for(i = 2; i <= N; i++)
        Fib[i] = Fib[i-1]+Fib[i-2];
 	
    //last index will have the result
    return Fib[N];
}
 	
int main()
{
    int n;
    scanf("%d",&n);
 	
    //if n == 0 or n == 1 the result is n
    if(n <= 1)
        printf("Fib(%d) = %d\n",n,n);
    else
        printf("Fib(%d) = %d\n",n,Fibonacci(n));
 	
    return 0;
}</pre>
	Fib(32764) = 777138227

6. Write a C program to compute Bread Fast Search Algorithm.

Input	Output

<pre>#include <stdio.h>
#include <stdlib.h>
 	
#define MAX 100
 	
int queue[MAX], front = -1, rear = -1;
int visited[MAX] = {0};
 	
void enqueue(int v) {
    if (rear == MAX - 1) return;
    if (front == -1) front = 0;
    queue[++rear] = v;
}
 	
int dequeue() {
    if (front == -1) return -1;
    int v = queue[front++];
    if (front > rear) front = rear = -1;
    return v;
}
 	
void BFS(int graph[MAX][MAX], int start, int n) {
    enqueue(start);
    visited[start] = 1;
    printf("BFS Traversal: ");</pre>
	Enter the number of vertices: 3
	Enter the adjacency matrix:
	1
	1 3 10 5 10
	
	0
	2
	2
	Enter the starting vertex: 1
	BFS Traversal: 1 0 2
	
	=== Code Execution Successful ===

<pre>   while (front != -1) {
        int v = dequeue();
        printf("%d ", v);
 	
        for (int i = 0; i < n; i++) {
            if (graph[v][i] && !visited[i]) {
                enqueue(i);
                visited[i] = 1;
            }
        }
    }
}
 	
int main() {
    int graph[MAX][MAX], n, start;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);
 	
    printf("Enter the starting vertex: ");
    scanf("%d", &start);
    BFS(graph, start, n);
    return 0;
}</pre>	

<table>
<tr><td colspan="2">7. Write a C program to compute Bread Fast Search Algorithm.</td></tr>
<tr><td>Input</td><td>Output</td></tr>
<tr><td><pre>// DFS algorithm in C

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

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

struct node* createNode(int v);

struct Graph {
  int numVertices;
  int* visited;

  // We need int** to store a two dimensional array.
  // Similary, we need struct node** to store an array of Linked
lists
  struct node** adjLists;
};

// DFS algo
void DFS(struct Graph* graph, int vertex) {
  struct node* adjList = graph->adjLists[vertex];
  struct node* temp = adjList;

  graph->visited[vertex] = 1;</pre></td><td><pre>Adjacency list of vertex 0
 2 -> 1 ->

 Adjacency list of vertex 1
 2 -> 0 ->

 Adjacency list of vertex 2
 3 -> 1 -> 0 ->

 Adjacency list of vertex 3
 2 ->
Visited 2
Visited 3
Visited 1
Visited 0

=== Code Execution Successful ===</pre></td></tr>
</table>

```
  printf("Visited %d \n", vertex);

  while (temp != NULL) {
    int connectedVertex = temp->vertex;

    if (graph->visited[connectedVertex] == 0) {
      DFS(graph, connectedVertex);
    }
    temp = temp->next;
  }
}

// Create a node
struct node* createNode(int v) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}

// Create graph
struct Graph* createGraph(int vertices) {
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices * sizeof(struct node*));

  graph->visited = malloc(vertices * sizeof(int));

  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }
  return graph;
}

// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
  // Add edge from src to dest
  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;

  // Add edge from dest to src
  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}

// Print the graph
void printGraph(struct Graph* graph) {
  int v;
  for (v = 0; v < graph->numVertices; v++) {
    struct node* temp = graph->adjLists[v];
    printf("\n Adjacency list of vertex %d\n ", v);
    while (temp) {
      printf("%d -> ", temp->vertex);
      temp = temp->next;
    }
```

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

int main() {
  struct Graph* graph = createGraph(4);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
  addEdge(graph, 2, 3);

  printGraph(graph);

  DFS(graph, 2);

  return 0;
}
```