- **Dynamic Programming Approaches**

1. Bottom-Up approach

Algorithm

1. set Fib[0] = 0

2. set Fib[1] = 1

3. From index 2 to n compute result using the below formula

   Fib[index] = Fib[index - 1] + Fib[index - 2]

4. The final result will be stored in Fib[n].

Code

```
#include<stdio.h>
int Fibonacci(int N)
{
    int Fib[N+1],i;
    Fib[0] = 0;
    Fib[1] = 1;

    for(i = 2; i <= N; i++)
        Fib[i] = Fib[i-1]+Fib[i-2];

return Fib[N];
}

int main()
{
    int n;
    scanf("%d",&n);

    if(n <= 1)
        printf("Fib(%d) = %d\n",n,n);
    else
        printf("Fib(%d) = %d\n",n,Fibonacci(n));

    return 0;}
```

2. Top-Down approach

Algorithm

3. Fib(n)
4. If n == 0 || n == 1 return n;
5. Otherwise, compute subproblem results recursively.
6. return Fib(n-1) + Fib(n-2);

Code

```c
#include<stdio.h>

int Fibonacci(int N)
{
    if(N <= 1)
        return N;
    return Fibonacci(N-1) + Fibonacci(N-2);
}

int main()
{
    int n;
    scanf("%d",&n);
    printf("Fib(%d) = %d\n",n,Fibonacci(n));

    return 0;
}
```

- ## 0/1 Knapsack

```
-   if wt[i] > w then
-   V[i,w] = V[i-1,w]
-
-   else if wt[i] <= w then
-   V[i,w] = max( V[i-1,w], val[i] + V[i-1, w - wt[i]] )
```

- After calculation, the value table V

V[i,w]	w = 0	1	2	3	4	5
i = 0	0	0	0	0	0	0
1	0	0	0	100	100	100
2	0	0	20	100	100	120
3	0	0	20	100	100	120
4	0	40	40	100	140	140

- Maximum value earned
  Max Value = V[n,W]
  = V[4,5]
  = 140

```
void knapSack(int W, int n, int val[], int wt[]) {
  int i, w;
  int V[n+1][W+1];

  for(w = 0; w <= W; w++) {
    V[0][w] = 0;
  }
```

```
  for(i = 0; i <= n; i++)
  {
    V[i][0] = 0;
  }

  for(i = 1; i <= n; i++) {
    for(w = 1; w <= W; w++) {
      if(wt[i] <= w) {
        V[i][w] = getMax(V[i-1][w], val[i] + V[i-1][w - wt[i]]);
      } else {
        V[i][w] = V[i-1][w];
      }
    }
  }

  printf("Max Value: %d\n", V[n][W]);
}
```

- LCS

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

- Kruskal

```
MST-KRUSKAL(G, w)
A = Ø
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

- Prims

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

- Bellman Ford

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

Greedy vs Divide & Conquer vs Dynamic Programming

Greedy	Divide & Conquer	Dynamic Programming
Optimises by making the best choice at the moment.	Optimises by breaking down a subproblem into simpler versions of itself and using multi-threading & recursion to solve.	Same as Divide and Conquer, but optimises by caching the answers to each subproblem as not to repeat the calculation twice.
Doesn't always find the optimal solution, but is very fast.	Always finds the optimal solution, but is slower than Greedy.	Always finds the optimal solution, but may be pointless on small datasets.
Requires almost no memory.	Requires some memory to remember recursive calls.	Requires a lot of memory for memoisation / tabulation

- ## Dijkstra

```c
#include <stdio.h>
#define INFINITY 9999
#define MAX 4

void Dijkstra(int Graph[MAX][MAX], int n, int start);

void Dijkstra(int Graph[MAX][MAX], int n, int start) {
  int cost[MAX][MAX], distance[MAX], pred[MAX];
  int visited[MAX], count, mindistance, nextnode, i, j;

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (Graph[i][j] == 0)
        cost[i][j] = INFINITY;
      else
        cost[i][j] = Graph[i][j];

  for (j = 0; j < n; j++) {
    distance[j] = cost[start][j];
    pred[j] = start;
    visited[j] = 0;
  }

  distance[start] = 0;
```

```c
  visited[start] = 1;
  count = 1;

  while (count < n - 1) {
    mindistance = INFINITY;

    for (i = 0; i < n; i++)
      if (distance[i] < mindistance && !visited[i]) {
        mindistance = distance[i];
        nextnode = i;
      }

    visited[nextnode] = 1;
    for (i = 0; i < n; i++)
      if (!visited[i])
        if (mindistance + cost[nextnode][i] < distance[i]) {
          distance[i] = mindistance + cost[nextnode][i];
          pred[i] = nextnode;
        }
    count++;
  }

  for (i = 0; i < n; i++)
    if (i != start) {
      printf("\nDistance from source to %d: %d", i, distance[i]);
    }
}
```