1) Recursive Fibonacci program

```cpp
#include<iostream>
using namespace std;
void printFibonacci(int n){
   static int n1=0, n2=1, n3;
   if(n>0){
        n3 = n1 + n2;
        n1 = n2;
        n2 = n3;
cout<<n3<<" ";
        printFibonacci(n-1);
   }
}
int main(){
   int n;
   cout<<"Enter the number of elements: ";
   cin>>n;
   cout<<"Fibonacci Series: ";
   cout<<"0 "<<"1 ";
   printFibonacci(n-2);  //n-2 because 2 numbers are already printed
    return 0;
}
```

1) Non Recursive Fibonacci program

```
#include <iostream>
using namespace std;
int main() {
 int n1=0,n2=1,n3,i,number;
cout<<"Enter the number of elements: ";
cin>>number;
cout<<n1<<" "<<n2<<" "; //printing 0 and 1
for(i=2;i<number;++i) //loop starts from 2 because 0 and 1 are already printd

{
 n3=n1+n2;
 cout<<n3<<" ";
 n1=n2;
 n2=n3;
}
  return 0;
  }
```

2. program for Huffman Coding with STL

```
#include <bits/stdc++.h>
using namespace std;

// A Huffman tree node
struct MinHeapNode {

        // One of the input characters
        char data;

        // Frequency of the character
        unsigned freq;

        // Left and right child
        MinHeapNode *left, *right;

        MinHeapNode(char data, unsigned freq)

        {

                left = right = NULL;
                this->data = data;
                this->freq = freq;
        }
};

// For comparison of
// two heap nodes (needed in min heap)
struct compare {
        bool operator()(MinHeapNode* l, MinHeapNode* r)

        {
                return (l->freq > r->freq);
        }
};

// Prints huffman codes from
// the root of Huffman Tree.
void printCodes(struct MinHeapNode* root, string str)
{

        if (!root)
                return;

        if (root->data != '$')
                cout << root->data << ": " << str << "\n";

        printCodes(root->left, str + "0");
        printCodes(root->right, str + "1");
}

// The main function that builds a Huffman Tree and
// print codes by traversing the built Huffman Tree
void HuffmanCodes(char data[], int freq[], int size)
{
        struct MinHeapNode *left, *right, *top;

        // Create a min heap & inserts all characters of data[]
        priority_queue<MinHeapNode*, vector<MinHeapNode*>, compare> minHeap;
```

```
for (int i = 0; i < size; ++i)
        minHeap.push(new MinHeapNode(data[i], freq[i]));

// Iterate while size of heap doesn't become 1
while (minHeap.size() != 1) {

        // Extract the two minimum
        // freq items from min heap
        left = minHeap.top();
        minHeap.pop();

        right = minHeap.top();
        minHeap.pop();

        // Create a new internal node with
        // frequency equal to the sum of the
        // two nodes frequencies. Make the
        // two extracted node as left and right children
        // of this new node. Add this node
        // to the min heap '$' is a special value
        // for internal nodes, not used
        top = new MinHeapNode('$', left->freq + right->freq);

        top->left = left;
        top->right = right;

        minHeap.push(top);
}

// Print Huffman codes using
// the Huffman tree built above
```

```
        printCodes(minHeap.top(), "");
}

// Driver Code
int main()
{

        char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
        int freq[] = { 5, 9, 12, 13, 16, 45 };

        int size = sizeof(arr) / sizeof(arr[0]);

        HuffmanCodes(arr, freq, size);

        return 0;
}
```

3. **Write a program to solve a fractional knapsack problem using a greedy method.**

```cpp
#include<bits/stdc++.h>
using namespace std;
#define int long long

struct item {
    double value, weight, valuePerWeight;
};

bool compare(item i1, item i2) {
    return i1.valuePerWeight > i2.valuePerWeight;
}

signed main() {
    int n; cin >> n;

    vector<item> items;
    for(int i=0; i<n; i++) {
        double v,w;
        cin >> v >> w;

        items.push_back({v,w,v/w});
    }

    double W; cin >> W;

    sort(items.begin(), items.end(), compare);

    int ans = 0;
    for(int i=0; i<n; i++) {
        if(W >= items[i].weight) {
            W -= items[i].weight;
            ans += items[i].value;
        }
        else {
            ans += W * items[i].valuePerWeight;
            W = 0;
            break;
        }
    }
    cout << ans << endl;
    return 0;
}
```

```
 sort(items.begin(), items.end(), compare);

    int ans = 0;
    for(int i=0; i<n; i++) {
        if(W >= items[i].weight) {
            W -= items[i].weight;
            ans += items[i].value;
        }
        else {
            ans += W * items[i].valuePerWeight;
            W = 0;
            break;
        }
    }
    cout << ans << endl;
    return 0;
}
```

4. Write a program to solve a 0-1 knapsack problem using dynamic program or branch and bound

0-1 Knapsack

<u>Problem</u>
Given an array of items with their {weight, value}, and a knapsack (bori) with weight W. Find the maximum value of items that can be stolen and put into a knapsack.
Note: We either have to pick full item or no item, we cannot take partial items.

Example:

value	4	1	3
weight	4	1	3

W = 50 kg

Possible combinations that we can steal:

{15, 30}	{40}
V = 60 + 100 = 160	V = 150

It is important to note that we cannot apply greedy technique here as items are indivisible.

<u>Way of thinking:</u>
We iterate from left to right in the items array. For each item we have 2 choices

1. Take it ➤ Remaining capacity of the knapsack decreases.
2. Don't take it ➤ Capacity of knapsack remains the same.

Let f(n, W) = denotes the maximum value of items that we can pick till item n and current capacity of knapsack W.

Therefore f(n, W) = max(*f(n-1, W), f(n-1, W-weight[n]) + value[n]*)
Since we can represent it as a recurrence relation, it follows optimal substructure property.

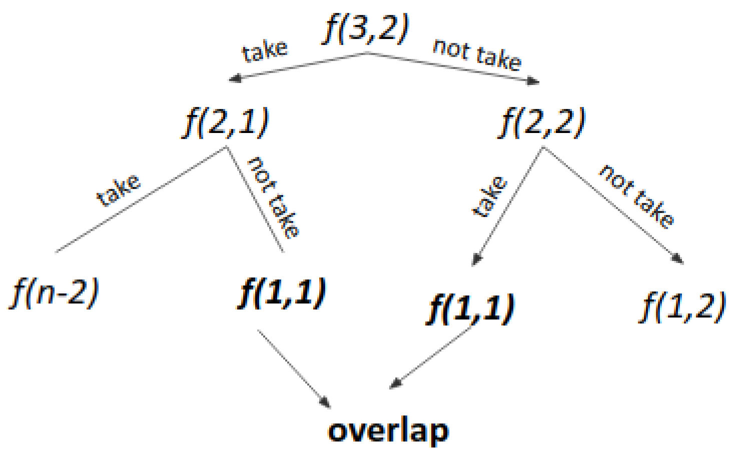
Let us check if it follows the overlapping subproblem property also?

Example

value	1	1	1
weight	5	10	20

W = 2kg

Recursion tree for the above example



Since *f(1,1)* repeats ➤ It follows overlapping subproblem property also.

Hence it can be solved using dynamic programming.

Brute force Approach

1. Make all subsequences and pick the one with maximum value, which fits the constraints of knapsack.

Time Complexity: O(2^n)
where n is the number of items

Optimal Solution (Using dynamic programming)

Approach 1 (Memoization)

1. Write the recursive solution.
2. Memoize it.

Approach 2 (Tabulation)

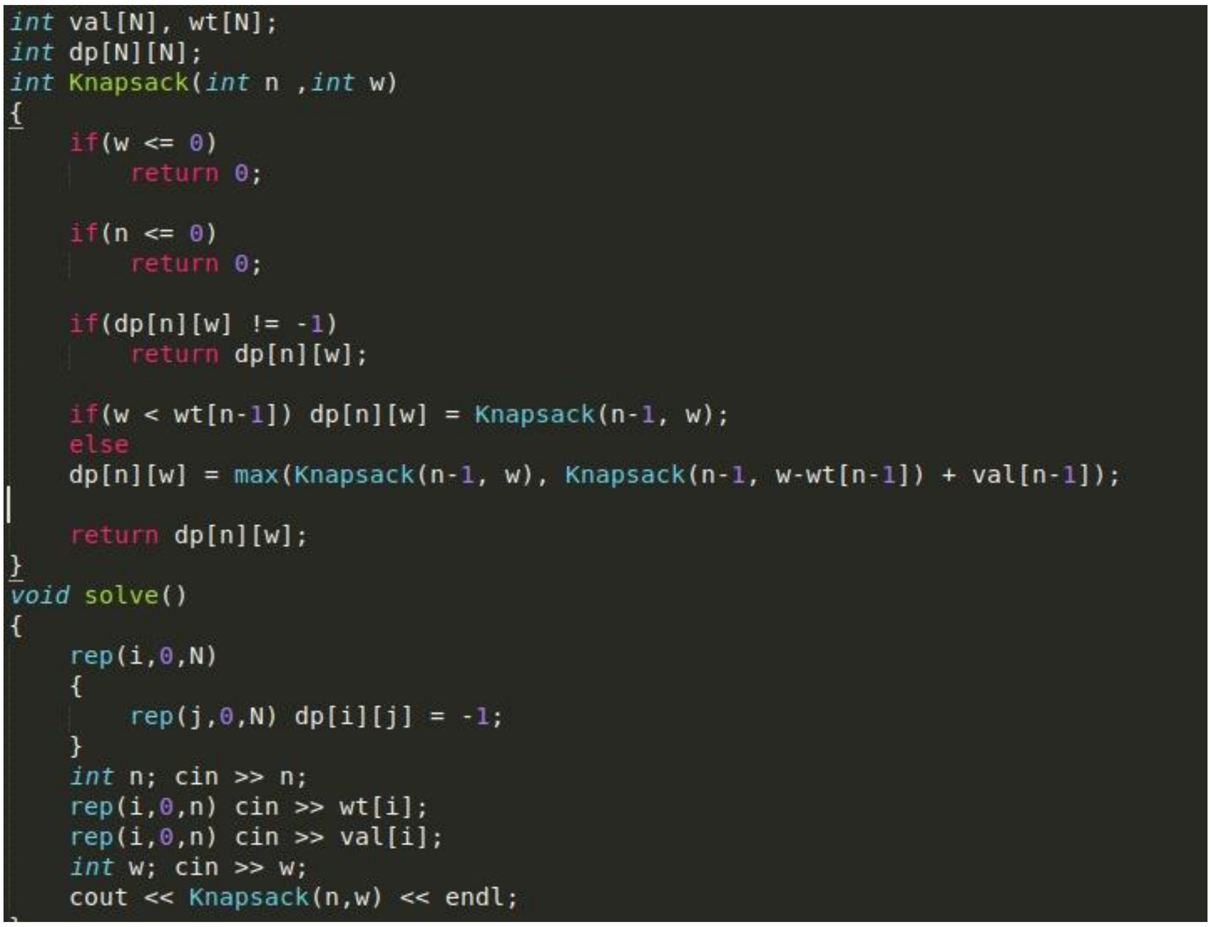
1. For each item, compute the answer for every weight from 0 to W.
2. Use recurrence of taking and not taking

dp[n][w] = max(dp[n-1][w], dp[n-1][w-wt[n]] + val[n])
provided w-wt[n] is non negative

3. Output the answer (*dp[n][w]*)

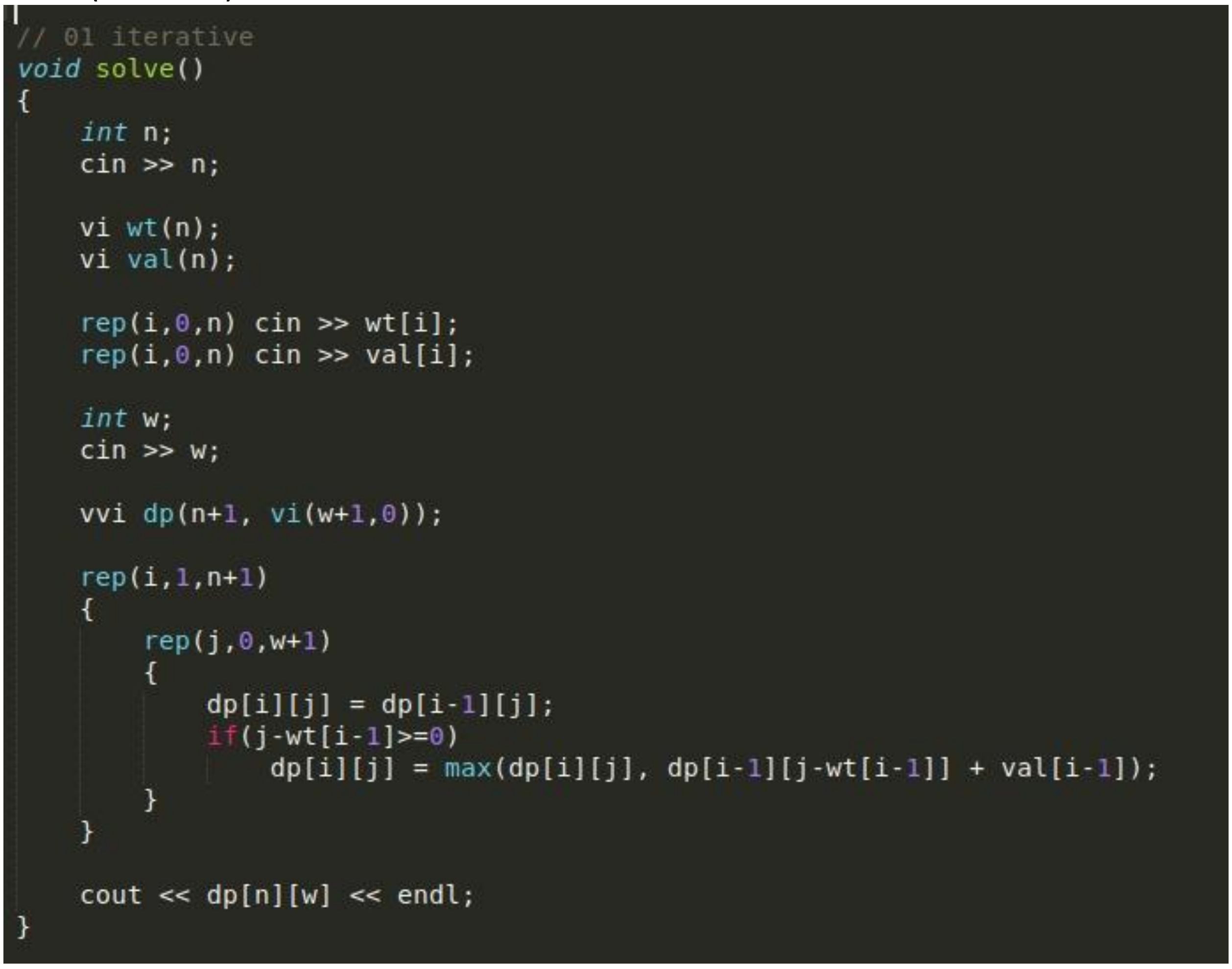
Time Complexity: 0(n*W)

Code (Memoization)



```
int val[N], wt[N];
int dp[N][N];
int Knapsack(int n ,int w)
{
    if(w <= 0)
        return 0;

    if(n <= 0)
        return 0;

    if(dp[n][w] != -1)
        return dp[n][w];

    if(w < wt[n-1]) dp[n][w] = Knapsack(n-1, w);
    else
    dp[n][w] = max(Knapsack(n-1, w), Knapsack(n-1, w-wt[n-1]) + val[n-1]);

    return dp[n][w];
}
void solve()
{
    rep(i,0,N)
    {
        rep(j,0,N) dp[i][j] = -1;
    }
    int n; cin >> n;
    rep(i,0,n) cin >> wt[i];
    rep(i,0,n) cin >> val[i];
    int w; cin >> w;
    cout << Knapsack(n,w) << endl;
```

Code (Iterative)



```
// 01 iterative
void solve()
{
    int n;
    cin >> n;

    vi wt(n);
    vi val(n);

    rep(i,0,n) cin >> wt[i];
    rep(i,0,n) cin >> val[i];

    int w;
    cin >> w;

    vvi dp(n+1, vi(w+1,0));

    rep(i,1,n+1)
    {
        rep(j,0,w+1)
        {
            dp[i][j] = dp[i-1][j];
            if(j-wt[i-1]>=0)
                dp[i][j] = max(dp[i][j], dp[i-1][j-wt[i-1]] + val[i-1]);
        }
    }

    cout << dp[n][w] << endl;
}
```

5.Design n-queen matrix having first queen placed use backtracking to place remaining queens to n-queen

```
#include <bits/stdc++.h>
#define N 4
using namespace std;

/* A utility function to print solution */
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                    cout << " " << board[i][j] << " ";
            printf("\n");
    }
}
```

/* A utility function to check if a queen can be placed on board[row][col]. Note that this function is called when "col" queens are already placed in columns from 0 to col -1. So we need to check only left side for attacking queens */

```
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    /* Check this row on left side */
    for (i = 0; i < col; i++)

            if (board[row][i])

                    return false;


    /* Check upper diagonal on left side */

    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)

            if (board[i][j])

                    return false;


    /* Check lower diagonal on left side */

    for (i = row, j = col; j >= 0 && i < N; i++, j--)

            if (board[i][j])

                    return false;


    return true;

}


/* A recursive utility function to solve N Queen problem */


bool solveNQUtil(int board[N][N], int col)

{

    /* base case: If all queens are placed then return true */

    if (col >= N)

            return true;


    /* Consider this column and try placing this queen in all rows one by
one */
```

```
    for (int i = 0; i < N; i++) {
        /* Check if the queen can be placed on
        board[i][col] */
        if (isSafe(board, i, col)) {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;

            /* recur to place rest of the queens */
            if (solveNQUtil(board, col + 1))
                return true;

            /* If placing queen in board[i][col]
            doesn't lead to a solution, then
            remove queen from board[i][col] */
            board[i][col] = 0; // BACKTRACK
        }
    }

    /* If the queen cannot be placed in any row in
        this column col then return false */
    return false;
}
```

/* This function solves the N Queen problem using Backtracking. It mainly uses solveNQUtil() to solve the problem. It returns false if queens cannot be placed, otherwise, return true and prints placement of queens in the

```
form of 1s. Please note that there may be more than one solutions, this
function prints one of the feasible solutions.*/

bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                                        { 0, 0, 0, 0 },
                                        { 0, 0, 0, 0 },
                                        { 0, 0, 0, 0 } };

    if (solveNQUtil(board, 0) == false) {
            cout << "Solution does not exist";
            return false;
    }

    printSolution(board);
    return true;
}

// driver program to test above function
int main()
{
    solveNQ();
    return 0;
}
```