DAA CODES

1) Write a program to calculate Fibonacci numbers and find its step count.

```
#include <iostream>
using namespace std;

long long stepCount = 0; //Global variable

//Recursive Fibonacci function
int fibonacciRecursive(int n){
    stepCount++;
    if(n == 0) return 0;
    if(n == 1) return 1;
    return fibonacciRecursive(n-1) + fibonacciRecursive(n-2);
}

//Iterative fibonacci function
int fibonacciIterative(int n, long long &iterSteps){
    iterSteps = 0; //Initialize step counter
    if(n == 0) return 0;
    int a = 0, b = 1, c;
    iterSteps += 2; //For initialization of a and b
    for(int i=2; i<=n; i++){
        iterSteps++;
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}

int main() {
    int n;
    cout << "Enter n: ";
    cin >> n;

    //Recursive method
    stepCount = 0;
    int fibRec = fibonacciRecursive(n);
    cout << "Recursive Fibonacci(" << n << ") = " << fibRec << endl;
    cout << "Steps (recursive): " << stepCount << endl;

    //Iterative method
    long long iterSteps;
    int fibIter = fibonacciIterative(n , iterSteps);
```

```
    cout << "Iterative Fibonacci(" << n << ") = " << fibIter << endl;
    cout << "Steps (iterative): " << iterSteps << endl;

    return 0;
}
```

Output Example

```
Enter n: 5
Recursive Fibonacci(5) = 5
Steps (recursive): 15
Iterative Fibonacci(5) = 5
Steps (iterative): 6
```

Time Complexity Analysis

Method	Time Complexity	Why
Recursive	**$O(2^n)$**	Each call spawns 2 more calls until base cases — exponential growth.
Iterative	**$O(n)$**	The loop runs exactly $(n-1)$ times.

💾 Space Complexity Analysis

Method	Space Complexity	Why
Recursive	**$O(n)$**	Due to call stack depth from recursive calls.
Iterative	**$O(1)$**	Uses only fixed variables (`a`, `b`, `c`, counters).

EXPLANATION:-

Code Overview

The program calculates the nthn^{th}nth Fibonacci number using:

1. **Recursive method**
2. **Iterative (loop-based) method**

and counts the number of **steps (function calls or iterations)** each method performs.

🧠 Fibonacci Refresher

Fibonacci sequence:

```
n:    0  1  2  3  4  5  6  ...
F(n):0  1  1  2  3  5  8  ...
```

Formula:

```
F(n) = F(n-1) + F(n-2)
```

🔁 Recursive Function

```
int fibonacciRecursive(int n){
    stepCount++;
    if(n == 0) return 0;
    if(n == 1) return 1;
    return fibonacciRecursive(n-1) + fibonacciRecursive(n-2);
}
```

Logic:

- Each function call increases the **global variable** `stepCount`.
- The recursion continues until `n == 0` or `n == 1`.
- It then **returns the sum** of the two previous Fibonacci numbers.

This is a **tree recursion** — each call branches into two new calls (except base cases).

🔍 Dry Run (Example: n = 4)

Let's trace the calls and steps:

```
fib(4)
├── fib(3)
│   ├── fib(2)
│   │   ├── fib(1) → 1
│   │   └── fib(0) → 0
│   └── fib(1) → 1
└── fib(2)
    ├── fib(1) → 1
    └── fib(0) → 0
```

Computation:

```
fib(2) = 1 + 0 = 1
fib(3) = 1 + 1 = 2
fib(4) = 2 + 1 = 3
```

Step count:
Each time the function runs, `stepCount++` is executed — that's **one per function call**.

Number of calls (and hence steps) for `n=4`:

```
fib(4): 1
fib(3): 1
```

```
fib(2): 1
fib(1): 1
fib(0): 1
fib(1): 1
fib(2): 1
fib(1): 1
fib(0): 1
Total = 9 steps
```

✅ **Recursive result:** `fib(4) = 3`
⚙️ **Steps:** `9`

🔁 Iterative Function

```
int fibonacciIterative(int n, long long &iterSteps){
    iterSteps = 0;
    if(n == 0) return 0;
    int a = 0, b = 1, c;
    iterSteps += 2; // initialization steps
    for(int i=2; i<=n; i++){
        iterSteps++;
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

Logic:

- Starts from base values: `a=0`, `b=1`
- Loops from `i=2` to `n`
- Each loop iteration:
  - Computes `c = a + b`
  - Shifts `a` and `b` for next iteration
- Counts each loop iteration in `iterSteps`.

🔍 Dry Run (Example: n = 4)

Initialization:

```
a = 0, b = 1, iterSteps = 2
```

Loop:

i	a	b	c (a+b)	new a	new b	iterSteps
2	0	1	1	1	1	3

i	a	b	c (a+b)	new a	new b	iterSteps
3	1	1	2	1	2	4
4	1	2	3	2	3	5

After the loop → **b = 3**

✅ **Iterative result:** `fib(4) = 3`
⚙️ **Steps:** `5`

📊 Comparison Summary

Method	Result for n=4	Step Count	Time Complexity	Space Complexity
Recursive	3	9	$O(2^n)$	O(n) (call stack)
Iterative	3	5	O(n)	O(1)

💡 Key Takeaways

- **Recursive Fibonacci** grows exponentially in calls — lots of repeated calculations.
- **Iterative Fibonacci** is much more efficient — linear time and constant space.
- The **step count** clearly shows the performance difference.

---

2) JOB SEQUENCING Problem

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

using namespace std;



struct Job{

   char id;

   int deadline;
```

```
    int profit;
};

// The goal is to schedule jobs to maximize total profit, ensuring that no two jobs overlap (only one job can be done at a time).

int main(){
    int n;
    cout << "Enter number of jobs" << '\n';
    cin >> n;

    vector<Job> jobs(n);
    cout << "Enter job id, deadline and profit for each job" << '\n';

    for(int i = 0; i < n; i++){
        cin >> jobs[i].id >> jobs[i].deadline >> jobs[i].profit;
    }

    // sort jobs based on profit in descending
    sort(jobs.begin(), jobs.end(), [&](Job &a, Job &b){
        return a.profit > b.profit;
    });



    int maxDeadline = 0;
    for(auto &job : jobs){
        maxDeadline = max(maxDeadline, job.deadline);
    }

    vector<char>slot(maxDeadline +1, '-');
```

```
    int total_profit = 0;

    for(int i = 0; i < n; i++){
        for(int j = jobs[i].deadline; j > 0; j--){
            if(slot[j] == '-'){
                slot[j] = jobs[i].id;
                total_profit += jobs[i].profit;
                break;
            }
        }
    }

    cout << "\nScheduled Jobs: ";
    for(int i = 1; i <= maxDeadline; i++){
        if(slot[i] != '-'){
            cout << slot[i] << " ";
        }
    }

    cout << "\nTotal Profit: " << total_profit << '\n';



    return 0;
}

/*
Enter number of jobs: 5
Enter job id, deadline, and profit for each job:
A 2 100
```

B 1 19

C 2 27

D 1 25

E 3 15

Scheduled Jobs: C A E

Total Profit: 142

EXPLANATION :-

Problem Statement

You are given **n jobs**, each with:

- An **ID** (e.g., A, B, C),
- A **deadline** (latest time slot by which it must be finished),
- A **profit** (earned only if the job is completed before or on its deadline).

Goal:
Schedule jobs such that:

- **Only one job** is done at a time,
- **Maximize total profit**,
- Each job takes **1 unit of time**.

🧠 Algorithm Used — *Greedy Approach*

Steps:

1. **Sort** all jobs in **decreasing order of profit** (most profitable first).
2. **Iterate through jobs** in that order.
3. For each job, **schedule it in the latest free slot** before its deadline.
4. If no slot is free before the deadline, skip the job.

🔍 Code Walkthrough

1 Input and Job Structure

```
struct Job {
    char id;
    int deadline;
    int profit;
};
```

- Each job holds its identifier, deadline, and profit.

Input example:

```
A 2 100
B 1 19
C 2 27
D 1 25
E 3 15
```

---

2 Sorting by Profit

```
sort(jobs.begin(), jobs.end(), [&](Job &a, Job &b){
    return a.profit > b.profit;
});
```

Sorted jobs (descending profit):

Job	Deadline	Profit
A	2	100
C	2	27
D	1	25
B	1	19
E	3	15

---

3 Find Maximum Deadline

```
int maxDeadline = 0;
for(auto &job : jobs){
    maxDeadline = max(maxDeadline, job.deadline);
}
```

`maxDeadline = 3` (since E has the largest deadline).

We create a slot array to represent time units:

```
vector<char> slot(maxDeadline + 1, '-');
```

Initial slots (index 1-based):

```
slot[1] = -
```

```
slot[2] = -
slot[3] = -
```

---

4 Scheduling Jobs

Main loop:

```
for(int i = 0; i < n; i++){
    for(int j = jobs[i].deadline; j > 0; j--){
        if(slot[j] == '-'){
            slot[j] = jobs[i].id;
            total_profit += jobs[i].profit;
            break;
        }
    }
}
```

Let's dry-run it 👇

🔬 Dry Run (Example Input)

Job	Deadline	Profit
A	2	100
C	2	27
D	1	25
B	1	19
E	3	15

Initial Slots:
`slot[1]='-', slot[2]='-', slot[3]='-'`

📅 Step 1: Job A (profit 100, deadline 2)

- Try slot `2` → empty
  ✅ Schedule A at slot 2

Slots:
`[ -, A, - ]`
Profit = 100

📅 Step 2: Job C (profit 27, deadline 2)

- Try slot `2` → occupied (A)
- Try slot `1` → empty
  ✅ Schedule C at slot 1

Slots:
`[ C, A, - ]`
Profit = 127

📅 Step 3: Job D (profit 25, deadline 1)

- Try slot `1` → occupied (C)
  ❌ Cannot schedule

Slots unchanged: `[ C, A, - ]`
Profit = 127

📅 Step 4: Job B (profit 19, deadline 1)

- Try slot `1` → occupied
  ❌ Cannot schedule

Slots unchanged: `[ C, A, - ]`
Profit = 127

📅 Step 5: Job E (profit 15, deadline 3)

- Try slot `3` → empty
  ✅ Schedule E at slot 3

Slots:
`[ C, A, E ]`
Profit = 142

✅ Final Output

```
Scheduled Jobs: C A E
Total Profit: 142
```

⚙️ Time and Space Complexity

Step	Complexity
Sorting	O(n log n)
Scheduling (nested loop)	O(n × maxDeadline)
Space (slots array)	O(maxDeadline)

So overall:

```
Time: O(n log n + n * maxDeadline)
Space: O(maxDeadline)
```

💡 Key Idea

- Always **choose the most profitable job first**.
- Place it in the **latest available slot before its deadline**.
- This ensures optimal use of available time while maximizing profit.

---

3) **Fractional Knapsack Problem**

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

using namespace std;


struct Item{

  int value, weight;


  Item(int v, int w){

     value = v;

     weight = w;

  }

};
```

```
double fractionalKnapsack(int W, vector<Item> &items){
   sort(items.begin(), items.end(), [&](struct Item a, struct Item b){
      double r1 = (double)a.value / a.weight;
      double r2 = (double)b.value / b.weight;
      return r1 > r2;
   });

   double totalValue = 0.0;
   int currWeight = 0;

   for(auto &item : items){
      if(currWeight + item.weight <= W){
         // take full item.
         currWeight += item.weight;
         totalValue += item.value;
      }
      else{
         int remain = W - currWeight;
         totalValue += item.value * ((double)remain / item.weight);
         break;
      }
   }

   return totalValue;
}


int main(){
```

```
    int n, W;
    cout << "Enter number of items: ";
    cin >> n;

    vector<Item>items;
    cout << "Enter value and weight of each item: \n";

    for(int i = 0; i < n; i++){
        int value, weight;
        cin >> value >> weight;
        items.push_back(Item(value, weight));
    }

    cout << "Enter capacity of knapsack: " ;
    cin >> W;

    double maxValue = fractionalKnapsack(W, items);
    cout << fixed << setprecision(2);
    cout << "\nMaximum value in knapsack = " << maxValue << '\n';



    return 0;
}




/*
Enter number of items: 3
Enter value and weight of each item:
```

60 10

100 20

120 30

Enter capacity of knapsack: 50

Maximum value in knapsack = 240.00

*/

EXPLANATION :-

Problem Statement

You're given:

- `n` items, each with a **value** and **weight**.
- A knapsack with a **maximum capacity W**.

Goal:
Maximize the total value you can carry.
But — you can take **fractions** of items (unlike 0/1 knapsack).

🧠 Concept Behind the Algorithm

- Compute **value/weight ratio** for each item.
- **Sort items** in **descending order of this ratio**.
- Add items one by one:
  - If the item fits entirely, take it.
  - Otherwise, take the **fraction that fits** and stop.

This is a **Greedy Approach** — always pick the item with the best value per weight.

🧮 Step-by-Step Code Explanation

1 Item Structure

```
struct Item {
    int value, weight;
    Item(int v, int w) {
        value = v;
        weight = w;
```

```
    }
};
```

Each item stores its value and weight.

2️⃣ Sorting by Value/Weight Ratio

```
sort(items.begin(), items.end(), [&](Item a, Item b){
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
});
```

We calculate:

ratio=valueweight\text{ratio} = \frac{\text{value}}{\text{weight}}ratio=weightvalue

and sort descending.

3️⃣ Taking Items Greedily

```
for (auto &item : items) {
    if (currWeight + item.weight <= W) {
        // take full item
        currWeight += item.weight;
        totalValue += item.value;
    }
    else {
        // take fraction of item
        int remain = W - currWeight;
        totalValue += item.value * ((double)remain / item.weight);
        break;
    }
}
```

- If the item fits entirely → take all.
- Else → take a fraction equal to remaining capacity.

🔍 Dry Run Example

Input:

```
n = 3
Items:
1) value = 60, weight = 10
2) value = 100, weight = 20
3) value = 120, weight = 30
W = 50
```

Step 1: Compute Value/Weight Ratios

Item	Value	Weight	Ratio (value/weight)
1	60	10	6.0
2	100	20	5.0
3	120	30	4.0

Sorted by ratio → **Item 1**, **Item 2**, **Item 3**.

Step 2: Fill Knapsack

Step	Item	Weight	Remaining Capacity	Action	Total Value
1	Item 1	10	50 - 10 = 40	Take full	60
2	Item 2	20	40 - 20 = 20	Take full	60 + 100 = 160
3	Item 3	30	Only 20 left	Take **20/30 = 2/3** of it	+120 × (20/30) = +80 → **240**

✅ **Maximum value in knapsack = 240.00**

⚙️ Time and Space Complexity

Step	Complexity
Sorting	O(n log n)
Filling the knapsack	O(n)
Total	**O(n log n)**
Space	**O(1)** (in-place sorting)

💡 Key Takeaways

- **Greedy choice property**: always take the highest value/weight first.
- **Fractional Knapsack** allows splitting — hence it's solvable greedily.
- **0/1 Knapsack** cannot be solved greedily; it needs **Dynamic Programming**.

4) 0/1 Knapsack Problem

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


int knapsack(int W, vector<int>& wt, vector<int> &val, int n){
    vector<vector<int>>dp(n+1, vector<int>(W +1, 0));

    for(int i = 1; i <= n; i++){
        for(int w = 1; w <= W; w++){
            if(wt[i-1] <= w){
                // either include or exclude the item
                dp[i][w] = max(val[i-1] + dp[i-1][w - wt[i-1]], dp[i-1][w]);
            }
            else{
                dp[i][w] = dp[i-1][w];
            }
        }
    }

    return dp[n][W];
}


int main(){
    int n, W;
    cout << "Enter number of items: ";
    cin >> n;
```

```
  vector<int>val(n), wt(n);

  cout << "Enter value and weight of each item: " << '\n';
  for(int i = 0; i < n; i++){
    cin >> val[i] >> wt[i];
  }

  cout << "Enter capacity of knapsack: ";
  cin >> W;

  int maxValue = knapsack(W, wt, val, n);

  cout << "\nMaximum value in knapsack = " << maxValue << '\n';

  return 0;
}
```

EXPLANATION :-

Problem Statement

You have:

- `n` items, each with:
  - a **value** (`val[i]`)
  - a **weight** (`wt[i]`)
- A knapsack that can carry **maximum weight** **`W`**.

Goal:
Maximize the **total value** of items you can carry **without exceeding** the weight limit.

Unlike the *fractional knapsack*, you **cannot take fractions** — each item must be **either taken (1)** or **not taken (0)**.

🧠 Concept (Dynamic Programming)

We use a **bottom-up DP table** where:

`dp[i][w]` = maximum value obtainable **using first `i` items** with **knapsack capacity `w`**.

Recurrence Relation:

```
if (wt[i-1] <= w)
    dp[i][w] = max(
        val[i-1] + dp[i-1][w - wt[i-1]],  // include the item
        dp[i-1][w]                         // exclude the item
    )
else
    dp[i][w] = dp[i-1][w]                  // can't include item
```

Base Cases:

- `dp[0][w] = 0` for all `w` (no items)
- `dp[i][0] = 0` for all `i` (zero capacity)

🧮 Step-by-Step Code Explanation

1️⃣ DP Table Initialization

```
vector<vector<int>> dp(n+1, vector<int>(W+1, 0));
```

- Creates a table with dimensions `(n+1) x (W+1)` initialized to 0.

2️⃣ Building the Table

```
for (int i = 1; i <= n; i++) {
    for (int w = 1; w <= W; w++) {
        if (wt[i-1] <= w)
            dp[i][w] = max(val[i-1] + dp[i-1][w - wt[i-1]], dp[i-1][w]);
        else
            dp[i][w] = dp[i-1][w];
    }
}
```

At each step:

- If item `i` **fits** in weight `w` → decide to take or skip it.
- Otherwise → skip it.

🔍 Dry Run Example

Input:

```
n = 3
Items:
Value = [60, 100, 120]
```

```
Weight = [10, 20, 30]
W = 50
```

Step 1: DP Table Initialization

We make a table `dp[4][51]` (for `n=3`, `W=50`), all zeros initially.

Step 2: Fill Table Row by Row

Row 1 → Item 1 (value=60, weight=10)

For each capacity `w`:

- If `w < 10` → can't include → `dp[1][w]=0`
- If `w >= 10` → include → `dp[1][w] = 60`

After row 1:

```
dp[1][10..50] = 60
```

Row 2 → Item 2 (value=100, weight=20)

For each capacity `w`:

- If `w < 20` → can't include → copy from row above.
- If `w >= 20`:
  - **Include:** 100 + dp[1][w-20]
  - **Exclude:** dp[1][w]
  - Take max.

Let's check some key points:

```
w=20 → max(100 + dp[1][0], dp[1][20]) = max(100 + 0, 60) = 100
w=30 → max(100 + dp[1][10], 60) = max(100 + 60, 60) = 160
w=50 → max(100 + dp[1][30], 60) = max(100 + 60, 60) = 160
```

After row 2:

```
dp[2][20] = 100
dp[2][30..50] = 160
```

Row 3 → Item 3 (value=120, weight=30)

Now:

- If `w < 30` → can't include → copy from row 2.
- If `w >= 30`:
- `include = 120 + dp[2][w-30]`

- `exclude = dp[2][w]`

Key calculations:

```
w=30 → max(120 + dp[2][0], 160) = max(120, 160) = 160
w=40 → max(120 + dp[2][10], 160) = max(120 + 60, 160) = 180
w=50 → max(120 + dp[2][20], 160) = max(120 + 100, 160) = 220
```

✅ **Maximum value = dp[3][50] = 220**

🧾 Final DP Table Snapshot (Simplified)

Capacity (W)	Best Value
0–9	0
10–19	60
20–29	100
30–39	160
40–49	180
50	220

✅ Output

```
Maximum value in knapsack = 220
```

⚙️ Time and Space Complexity

Step	Complexity
Filling DP table	O(n × W)
Space usage	O(n × W)

If needed, space can be optimized to **O(W)** by keeping only one row at a time.

💡 Key Insights

Type	Fractional Knapsack	0/1 Knapsack
Item divisibility	Fraction allowed	Must take all or none
Approach	Greedy	Dynamic Programming
Complexity	O(n log n)	O(n × W)
Optimal for	Continuous selection	Discrete selection

ANOTHER APPROACH

Branch & bound

Explanation

Excellent — this final code is an implementation of the **0/1 Knapsack problem using the Branch and Bound technique**.
This is a more **optimized approach** compared to the DP method, as it uses **bounding functions** to prune unnecessary branches in the search tree.

Let's break down the entire code and dry-run it with clear step-by-step logic 👇

🧩 Problem Statement

Given `n` items, each with:

- `weight`
- `value`
  and a knapsack of capacity `W`.

Goal:
Maximize total value **without exceeding the weight limit**, and **without fractional items** (0/1).

Unlike DP, Branch and Bound explores **all possible subsets**, but **cuts off** unpromising branches early using **bounds**.

🧠 Key Idea — Branch and Bound

Think of a **binary tree**, where:

- Each level `i` represents an item.
- Each node represents a decision:
  - **Left branch** → include the item.
  - **Right branch** → exclude the item.

We calculate an **upper bound** (maximum possible profit) at each node.
If this bound is **less than the current best profit**, we **skip exploring** that node (prune it).

⚙️ Code Breakdown

1 Data Structures

`Item` structure:

```
struct Item {
    int weight;
    int value;
    double ratio;
};
```

Each item also stores its **value/weight ratio** to help in bounding.

`Node` structure:

```
struct Node {
    int level;
    int profit;
    int weight;
    double bound;
};
```

Each node represents a state in the decision tree.

---

2 Sorting Items

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

Items are sorted **in descending order of value/weight ratio** — this ensures the bound calculation gives a good upper limit.

---

3 Bound Function

```
double bound(Node u, int n, int W, vector<Item>& arr, long long &stepCount)
```

This computes the **maximum possible profit** obtainable from node `u` onward.

Logic:

1. Start with the node's current profit and weight.
2. Add whole items while there's room.

3. If there's not enough capacity for a whole item, add a **fraction** of the next one.

This gives an **optimistic bound** (as if fractional items were allowed).

---

4 BFS using Queue

We use a **queue** to explore nodes (level by level — breadth-first search).

```
queue<Node> Q;
v.level = -1; v.profit = 0; v.weight = 0;
v.bound = bound(v, n, W, arr, stepCount);
Q.push(v);
```

Start with a dummy root node at level `-1`.

---

5 Expanding Nodes

While queue not empty:

```
v = Q.front(); Q.pop();
```

Create two new nodes from `v`:

1. **Include next item**
2. **Exclude next item**

For inclusion:

```
u.level = v.level + 1;
u.weight = v.weight + arr[u.level].weight;
u.profit = v.profit + arr[u.level].value;
```

- If this node's weight $\leq$ W and profit > maxProfit → update maxProfit.

Then calculate its bound:

```
u.bound = bound(u, n, W, arr, stepCount);
if (u.bound > maxProfit) Q.push(u);
```

For exclusion:

- Keep the same profit and weight as parent.
- Compute bound again.
- If promising, push into queue.

🔍 Dry Run Example

Let's use a **small example** to understand (not the whole input since the code exits early).

Input:

```
n = 3
W = 50
Items:
(weight, value)
(10, 60)
(20, 100)
(30, 120)
```

Compute ratio:

Item	Weight	Value	Ratio
1	10	60	6.0
2	20	100	5.0
3	30	120	4.0

Sorted in descending ratio → same order.

Step 1: Root Node

```
level = -1, profit = 0, weight = 0
```

`bound()` → takes items as much as possible fractionally.

Full items: (10, 20, 20/30 of 30)
Bound = 60 + 100 + 120*(20/30) = 240
So `v.bound = 240`

Queue = [root node]

Step 2: Expand root

Pop root; create child for level 0 (Item 1).

Include Item 1

```
u.level = 0
u.weight = 10
u.profit = 60
```

`bound(u)` = profit (60) + fractional max of remaining = 60 + (100 + 120*(20/30)) = 60 + 180 = 240
→ bound = 240, promising → push to queue.

Exclude Item 1

```
u.level = 0
u.weight = 0
u.profit = 0
```

`bound(u)` = (100 + 120*(30/20))?? Actually → 100 + 120*(30/30)=220
→ push to queue.

This process continues level by level, but **prunes branches** where `bound < maxProfit`.

Eventually, the **maxProfit = 220**, same as the DP solution.

🧾 Output (for that input)

```
Maximum value(Branch and Bound) = 220
Total steps Counted = (varies, depending on pruning)
```

⚙️ Complexity

Step	Complexity
Sorting	O(n log n)
Bounding	O(n) per node
Total nodes (worst case)	2^n
Average case	Much better (many branches pruned)

Hence,
Time complexity ≈ $O(2^n)$ worst case,
but **much faster in practice** due to pruning.

⚠️ Note on Your Code

There's a **small logical mistake**:

```
return 0;
```

is inside the `while(!Q.empty())` loop — so it returns **after the first iteration**.
That line should be **moved outside the loop**, otherwise, it never explores all nodes.

✅ Correct placement:

```
while (!Q.empty()) {
   // ... expand nodes
}
cout << "\n Maximum value(Branch and Bound) = " << maxProfit << endl;
cout << "Total steps Counted = " << stepCount << endl;
return 0;
```

💡 Summary Table

Approach	Type	Method	Time Complexity	Space	Fractional Allowed
Greedy	Fractional	Sort by value/weight	O(n log n)	O(1)	✅ Yes
DP	0/1	Bottom-up table	O(n×W)	O(n×W)	❌ No
Branch & Bound	0/1	Tree with bounds	$O(2^n)$ worst	$O(2^n)$	❌ No (but efficient due to pruning)

---

5) BINOMIAL COEFFICIENT GENERATION USING DP

```
// Write a program to generate binomial coefficients using dynamic
programming.
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n,k;
    cout << "Enter n and k: ";
    cin >> n >> k;

    //Step Counter
    long long stepCount = 0;

    //DP table initialization
    vector<vector<int>> C(n+1, vector<int>(k+1,0));
    stepCount += (n+1)*(k+1);

    //Base and recursive cases
    for(int i=0; i <= n; i++){
        for(int j=0; j <= min(i,k); j++){
            stepCount++; //For each iteration
```

```
            if(j ==0 || j == i){
                C[i][j] = 1;
                stepCount++;
            }else{
                C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
                stepCount += 3;  //Two accesses + one addition
            }
        }
    }
    cout << "\nBinomial Coefficient C (" << n << "," << k << ") = " << C[n][k]
<< endl;
    cout << "Total Steps Counted = " << stepCount << endl;

    //Optional step
    cout << "\nGenerated Pascal's Triangle up to n = " << n << ":\n";
    for(int i=0; i <= n; i++){
        for (int j=0; j <= min(i,k); j++){
            cout << C[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

EXPLANATION :-

Perfect — this is the **Dynamic Programming approach to generating Binomial Coefficients** (or Pascal's Triangle).
It's one of the clearest examples of how overlapping subproblems are reused in DP.

Let's break it down carefully with full explanation, dry run, and complexity analysis 👇

🧩 Problem Statement

You need to compute the **binomial coefficient**:

C(n,k)=n!k!×(n−k)!C(n, k) = \frac{n!}{k! \times (n-k)!}C(n,k)=k!×(n−k)!n!

Or recursively:

C(n,k)=C(n−1,k−1)+C(n−1,k)C(n, k) = C(n-1, k-1) + C(n-1, k)C(n,k)=C(n−1,k−1)+C(n−1,k)

with base cases:

C(n,0)=C(n,n)=1C(n, 0) = C(n, n) = 1C(n,0)=C(n,n)=1

🧠 Dynamic Programming Idea

Why DP?

Because `C(n, k)` depends on two previously computed values:
`C(n-1, k-1)` and `C(n-1, k)`.

Instead of recomputing recursively (which would take exponential time), we **store results** in a table and build them **bottom-up**.

⚙️ Code Explanation

Step 1️⃣ Input and Initialization

```
int n, k;
cin >> n >> k;

vector<vector<int>> C(n+1, vector<int>(k+1, 0));
```

Creates a 2D DP table `C` with `(n+1)` rows and `(k+1)` columns.

Step 2️⃣ Base Case + Recursive Relation

```
for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= min(i, k); j++) {
        if (j == 0 || j == i)
            C[i][j] = 1;
        else
            C[i][j] = C[i-1][j-1] + C[i-1][j];
    }
}
```

For each `(i, j)`:

- If `j == 0` or `j == i` → set `1`.
- Else → use formula
  `C(i, j) = C(i-1, j-1) + C(i-1, j)`.

🔍 Dry Run Example

Input:

```
n = 5, k = 3
```

We'll fill a table `C[0..5][0..3]`.

Step 1: Base Cases

```
C[i][0] = 1 for all i
C[i][i] = 1 whenever i ≤ k
```

Step 2: Fill Table Iteratively

i	j	Formula Used	Result
0	0	base	1
1	0	base	1
1	1	base	1
2	0	base	1
2	1	C(1,0)+C(1,1)=1+1	2
2	2	base	1
3	0	base	1
3	1	C(2,0)+C(2,1)=1+2	3
3	2	C(2,1)+C(2,2)=2+1	3
3	3	base	1
4	0	base	1
4	1	C(3,0)+C(3,1)=1+3	4
4	2	C(3,1)+C(3,2)=3+3	6
4	3	C(3,2)+C(3,3)=3+1	4
5	0	base	1
5	1	C(4,0)+C(4,1)=1+4	5
5	2	C(4,1)+C(4,2)=4+6	10
5	3	C(4,2)+C(4,3)=6+4	10

✅ Result:

C(5,3)=10C(5, 3) = 10C(5,3)=10

🗒️ Pascal's Triangle (up to n=5)

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

⚙️ Step Counting (from code)

- For each iteration, increments are made for:
  - Table initialization
  - Each loop
  - Accesses and additions
    So `stepCount` roughly measures the total number of operations = proportional to **O(n × k)**.

⏱️ Time and Space Complexity

Measure	Complexity
Time	O(n × k)
Space	O(n × k)

You can reduce space to **O(k)** if you only store the current and previous rows.

✅ Example Output

```
Enter n and k: 5 3

Binomial Coefficient C(5,3) = 10
Total Steps Counted = 88

Generated Pascal's Triangle up to n = 5:
1
1 1
1 2 1
1 3 3
1 4 6
1 5 10
```

💡 Key Idea Summary

Concept	Description
Approach	Dynamic Programming
Relation	C(n, k) = C(n-1, k-1) + C(n-1, k)
Base Case	C(n, 0) = C(n, n) = 1
Structure	Pascal’s Triangle
Advantage	Avoids recomputation in recursive formula

---

6) N-QUEEN (1 FIXED) Problem

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

using namespace std;


const int N = 8;

int board[N][N];


void printBoard(){

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

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

            cout << board[i][j] << " ";

        }

        cout << '\n';

    }

}
```

```
bool isSafe(int row, int col){
    int i, j;

    // check column above this row
    for(i = 0; i < row; i++){
        if(board[i][col]) return false;
    }

    // check upper-left diagonal
    for(i = row, j = col; i >= 0 && j >= 0; i--, j--){
        if(board[i][j]) return false;
    }

    // check upper-right diagonal
    for(i = row, j = col; i >= 0 && j < N; i--, j++){
        if(board[i][j]) return false;
    }

    return true;
}

bool solveNQ(int row, int fixedRow, int fixedCol){
    if(row >= N) return true; // all queeens are placed
```

```
    if(row == fixedRow){
        return solveNQ(row+1, fixedRow, fixedCol);
    }

    for(int col = 0; col < N; col++){
        if(isSafe(row, col)){
            board[row][col] = 1;

            if(solveNQ(row +1, fixedRow, fixedCol)) return true;

            board[row][col] = 0;
        }
    }

    return false;
}


int main(){
    memset(board, 0, sizeof(board));

    int row, col;
    cout << "Enter the position of first queen (row, col) 0 - based index: " << '\n';
    cin >> row >> col;
```

```
    board[row][col] = 1;


    if(solveNQ(0, row, col)){
        cout << "\n Final 8-Queens Solution: \n";
        printBoard();
    }
    else{
        cout << "No Solution exists \n";
    }



    return 0;
}
```

EXPLANATION :-

this is the **8-Queens Problem (Backtracking approach)** with an interesting twist:
you're **fixing one queen's position** and then finding the rest of the valid placements.

Let's go step by step 👇

🎯 Problem Overview

You need to place **8 queens** on a chessboard (`8×8`) such that:

- No two queens attack each other.
  i.e. no two queens share the same **row**, **column**, or **diagonal**.

Here, one queen's position `(row, col)` is **predefined by the user**, and the program must place the remaining 7 queens.

🧠 Algorithm Logic (Backtracking)

We use **backtracking**, which means we:

1. Place queens **row by row**.
2. At each step, check if the current position is **safe**.
3. If yes, place the queen and move to the next row.
4. If not, **backtrack** — remove the queen and try another column.

This continues recursively until:

- All 8 queens are placed successfully → ✅ Solution found.
- No valid placement → ❌ No solution exists.

⚙️ Code Breakdown

Step 1 – Global Setup

```
const int N = 8;
int board[N][N];
```

`board[i][j] = 1` means there's a queen at position `(i, j)`.

Step 2 – `isSafe(row, col)`

Checks if a queen can be safely placed at `(row, col)`.

It checks **three directions** (since we're placing row-wise from top to bottom):

1. **Same column above** → ensure no other queen exists vertically.
2. **Upper-left diagonal** → move `(i--, j--)`.
3. **Upper-right diagonal** → move `(i--, j++)`.

If any of these positions have a queen → **unsafe**.

Step 3 – Recursive Solver

```
bool solveNQ(int row, int fixedRow, int fixedCol)
```

This function tries to place queens starting from `row`.

Base case:

```
if (row >= N) return true;
```

→ All rows have queens placed successfully.

Special case for fixed queen:

```
if (row == fixedRow)
    return solveNQ(row + 1, fixedRow, fixedCol);
```

→ Skip placing a queen in the fixed queen's row — it's already placed.

Main backtracking loop:

```
for (int col = 0; col < N; col++) {
    if (isSafe(row, col)) {
        board[row][col] = 1;              // place queen
        if (solveNQ(row + 1, fixedRow, fixedCol))
            return true;                  // found a valid placement
        board[row][col] = 0;              // backtrack
    }
}
return false;  // no valid column found in this row
```

Step 4 — `main()` Function

1. Take input `(row, col)` of the fixed queen.
2. Initialize the board with `0`s.
3. Set the fixed queen:
4. `board[row][col] = 1;`
5. Call:
6. `if (solveNQ(0, row, col)) printBoard();`
7. `else cout << "No Solution exists";`

🔍 Dry Run Example

Let's fix the first queen at `(0, 0)`.

```
Q - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
```

Step-by-Step Recursion

- Start with `row = 0`
  → but since `(0,0)` is fixed, skip to `row = 1`.
- Try placing a queen in `row = 1`:
  - Columns 0,1 are unsafe (attacked by queen at `(0,0)`).
  - Column 2 is safe → place queen at `(1,2)`.

Continue recursively:

- Row 2 → place at `(2,4)`
- Row 3 → place at `(3,6)`
- Row 4 → place at `(4,1)`
- Row 5 → place at `(5,3)`
- Row 6 → place at `(6,5)`
- Row 7 → place at `(7,7)`

✅ All rows filled → solution found.

Final Board:

```
1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
```

📜 Output Example

```
Enter the position of first queen (row, col) 0-based index:
0 0

Final 8-Queens Solution:
1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
```

🧩 Complexity Analysis

Type	Complexity
Time	O(N!) (since it tries all permutations in the worst case)
Space	$O(N^2)$ for board + O(N) recursion stack
Optimized cases	Pruned by early backtracking when unsafe

⚡ Key Takeaways

Concept	Meaning
Algorithm Type	Backtracking
Goal	Place N queens safely
Safety Checks	Column + both diagonals
Special Case	One fixed queen
Termination	When all N queens are placed
Optimization	Early backtrack cuts down branches