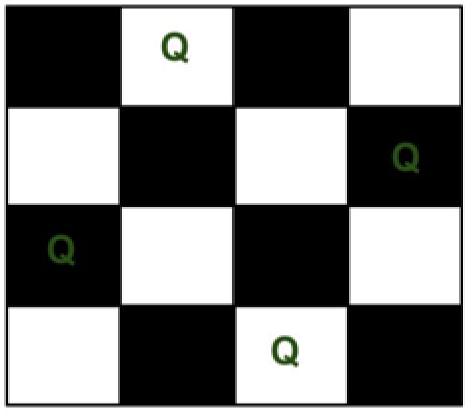
Practical:7 Backtracking and Branch & Bound

7.1 Write a C program to implement 4 Queen Problem. Is it possible to find the solution for 1 Queen, 2 Queen, 3 Queen?

Theory:

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen Problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.

{0, Q, 0, 0}

{0, 0, 0, Q}

{Q, 0, 0, 0}

{0, 0, Q, 0}

Another Solution is:

{0, 0, Q, 0}

{Q, 0, 0, 0}

{0, 0, 0, Q}

{0, Q, 0, 0}

The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

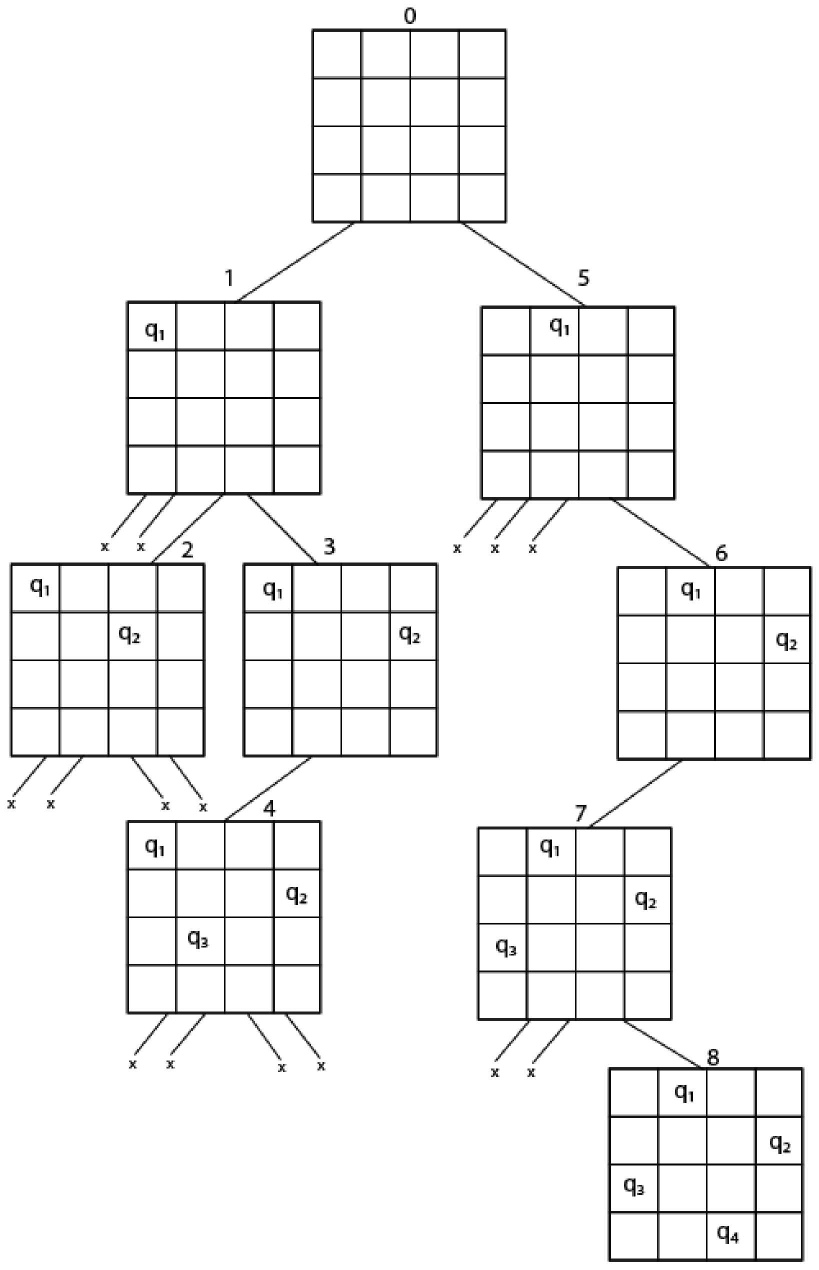


Fig shows the complete state space for 4-queens problem.

Algorithm:

Step 1: Start

Step 2: Given n queens, read n from user and let us denote the queen number by k. k=1,2,..,n.

Step 3: We start a loop for checking if the queen can be placed in the respective column of the row.

Step 4: For checking that whether the queen can be placed or not, we check if the previous queens are not in diagonal or in same row with it.

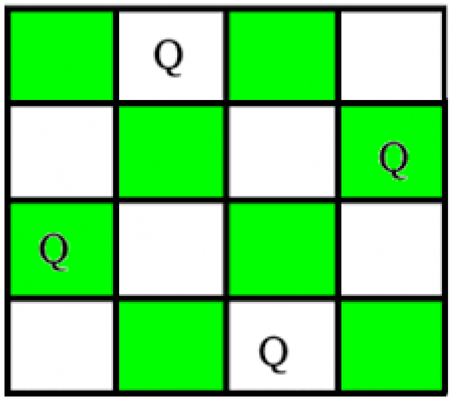
Step 5: If the queen cannot be placed backtracking is done to the previous queens until a feasible solution is not found.

Step 6: Repeat the steps 3-5 until all the queens are placed.

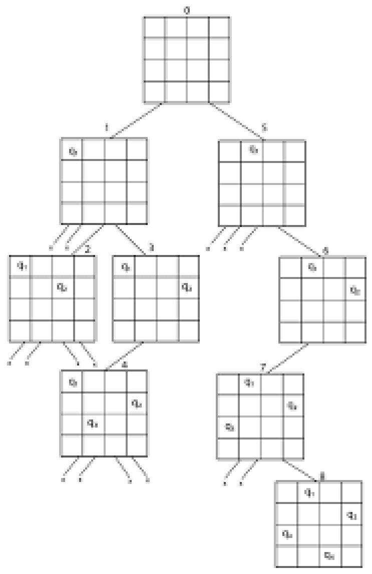
Step 7: The column numbers of the queens are stored in an array and printed as a n-tuple solution

Step 8: Stop

Example:



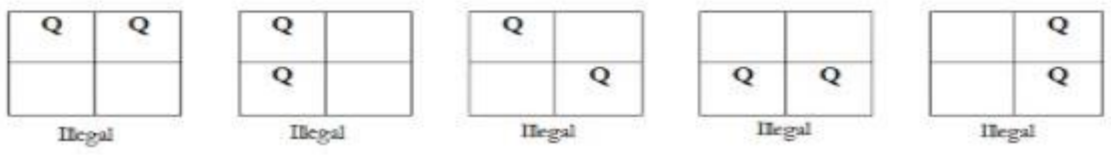
4 Queen Problem



N-Queen Problem

- Yes, It is possible to find **1 queen problem**, because we have only 1 queen and 1 place, so 1 queen can easily store that place.

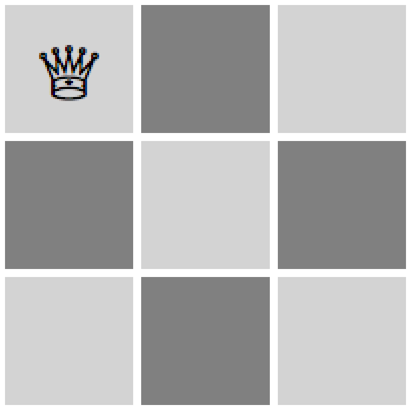
- **2 – Queen's problem** is not solvable because 2 – Queens can be placed on 2 x 2 chess board as shown in figure.

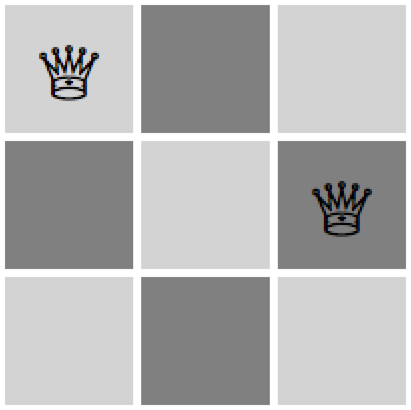


- So as shown in figure we can't put a queen in any way that itfollow the rules of N Queen Problem.

- **3 – Queen's problem** is not solvable because 3 – Queens can be placed on 3 x 3 chess board

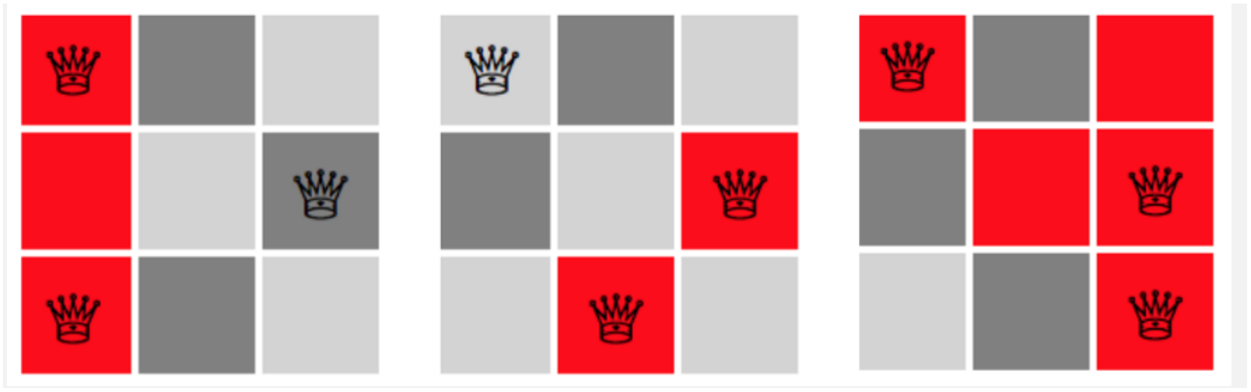
- let’s place the first queen to the top left.
- For the second row, we only have one spot — the far right — to place the queen without the two queens attacking each other.





Exploring permutations after placing the first piece in row 1, column 1

- But then: we are not able to fit a third queen in the third row.

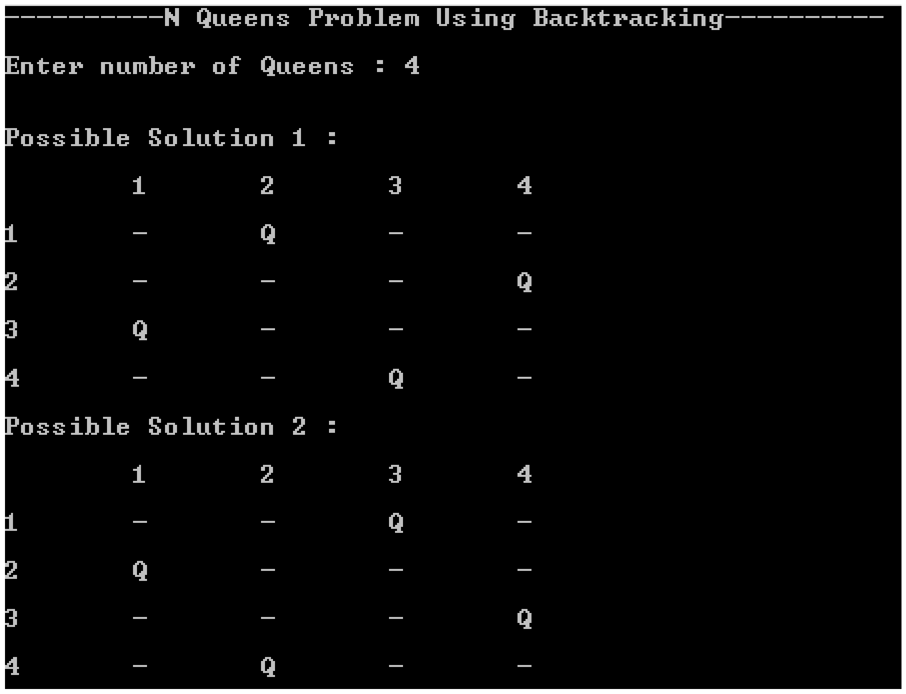


Exploring Possible Placement for Row 3
Placing a Queen in any of the columns in row 3 would create a conflict with other Queens. So we do not have a valid solution for 3 queen problem.

Code:

```
#include <stdio.h>

#include <math.h>


int board[20], count;

//function for printing the solution for N-queen Problem

void print(int n)

{

        int i, j;

        printf("\nPossible Solution %d :\n\n", ++count);


        for (i = 1; i <= n; ++i)

                printf("\t%d", i);


        for (i = 1; i <= n; ++i)
        {
                printf("\n\n%d", i);
                for (j = 1; j <= n; ++j) //for nxn board
                {
                        if (board[i] == j)
                                printf("\tQ"); //queen at ith,jth position
                        else
                                printf("\t-"); //empty position
                }
        }
}

/*funtion to check conflicts for row, column and diagonal
If no conflict for respective postion returns 1 otherwise returns 0*/
int place(int row, int column)
{
        int i;
        for (i = 1; i <= row - 1; ++i)
        {
                //checking column and diagonal conflicts for two queens are under
attack
                if (board[i] == column)
                        return 0;
                else if (abs(board[i] - column) == abs(i - row))
                return 0;
        }
        return 1; //no conflicts means no queens are under attack
}

//function to check for proper positioning of queen
void queen(int row, int n)
{
        int column;
        for (column = 1; column <= n; ++column)
        {
                if (place(row, column))
                {
                        board[row] = column; //no conflicts so place queen
                        if (row == n) //dead end
                                print(n); //printing the board config
                        else //try queen with next position
                                queen(row + 1, n);
                }
        }
}

int main()
{
        int n, i, j;
        void queen(int row, int n);

        printf("----------N Queens Problem Using Backtracking----------");
        printf("\n\nEnter number of Queens : ");
        scanf("%d", &n);
        queen(1, n);
        return 0;
}
```

OUTPUT:



```
----------N Queens Problem Using Backtracking----------

Enter number of Queens : 4

Possible Solution 1 :
        1       2       3       4
1       -       Q       -       -
2       -       -       -       Q
3       Q       -       -       -
4       -       -       Q       -
Possible Solution 2 :
        1       2       3       4
1       -       -       Q       -
2       Q       -       -       -
3       -       -       -       Q
4       -       Q       -       -
```

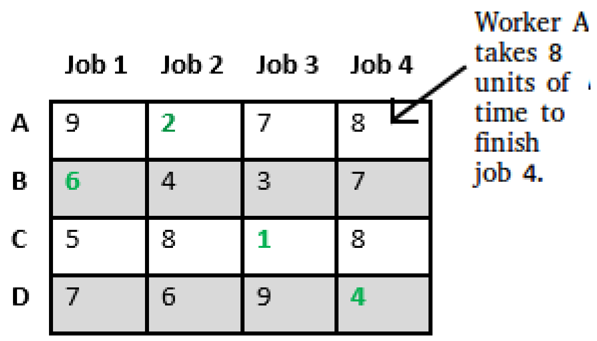
Conclusion:

I have successfully performed the given N-Queen problem using ‘Backtracking’ and had taken out the output of the given test cases. I learn about Backtracking is a general algorithm for finding all the solutions to some computational problems.

5.2 Amar takes 2, 6 and 7 hours of time to perform cooking, gardening and cleaning respectively. Akbar takes 4, 8 and 3 hours of time to perform cooking, gardening and cleaning respectively. Anthony takes 9, 5 and 1 hours of time to perform cooking, gardening and cleaning respectively. Find out optimal job assignment for Amar, Akbar and Anthony.

Theory:

Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.



	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Worker A takes 8 units of time to finish job 4.

An example job assignment problem. Green values show optimal job assignment that is A-Job4, B-Job1 C-Job3 and D-Job4

- The selection rule for the next node in BFS and DFS is “blind”. i.e. the selection rule does not give any preference to a node that has a very good chance of getting the search to an answer node quickly. The search for an optimal solution can often be speeded by using an “intelligent” ranking function, also called an approximate cost function to avoid searching in sub-trees that do not contain an optimal solution. It is similar to BFS-like search but with one major optimization. Instead of following

FIFO order, we choose a live node with least cost. We may not get optimal solution by following node with least promising cost, but it will provide very good chance of getting the search to an answer node quickly.

There are two approaches to calculate the cost function:

1. For each worker, we choose job with minimum cost from list of unassigned jobs (take minimum entry from each row).
2. For each job, we choose a worker with lowest cost for that job from list of unassigned workers (take minimum entry from each column).

Algorithm:

Let's take below example and try to calculate promising cost when Job 2 is assigned to worker A.

	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Since Job 2 is assigned to worker A (marked in green), cost becomes 2 and Job 2 and worker A becomes unavailable (marked in red).

	Job 1	Job 2	Job 3	Job 4
A	9	**2**	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

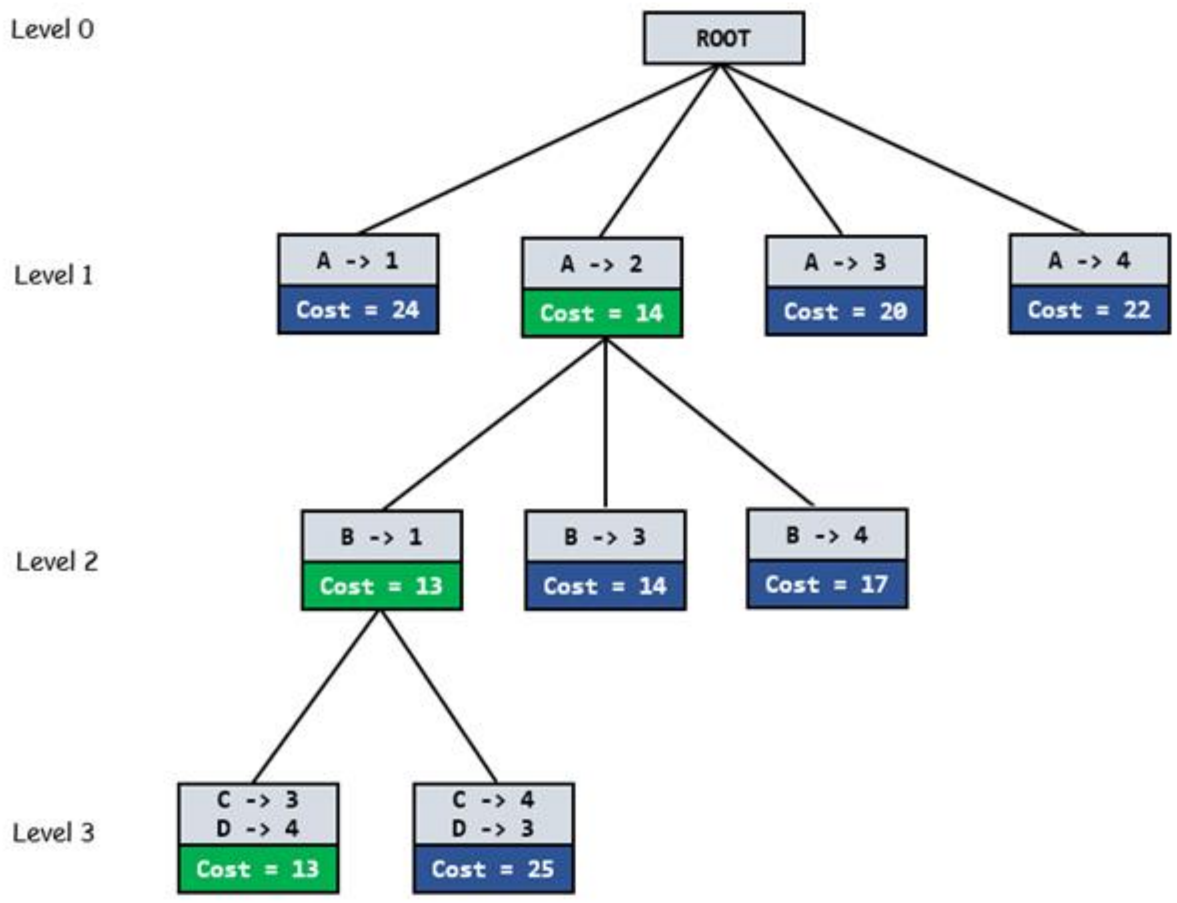
Now we assign job 3 to worker B as it has minimum cost from list of unassigned jobs. Cost becomes 2 + 3 = 5 and Job 3 and worker B also becomes unavailable.

	Job 1	Job 2	Job 3	Job 4
A	9	**2**	7	8
B	6	4	**3**	7
C	5	8	1	8
D	7	6	9	4

Finally, job 1 gets assigned to worker C as it has minimum cost among unassigned jobs and job 4 gets assigned to worker C as it is only Job left. Total cost becomes 2 + 3 + 5 + 4 = 14.

	Job 1	Job 2	Job 3	Job 4
A	9	**2**	7	8
B	6	4	**3**	7
C	**5**	8	1	8
D	7	6	9	**4**

Below diagram shows complete search space diagram showing optimal solution path in green.



<u>Example:</u>

```
node
{
  int job_number;
  int worker_number;
  node parent;
  int cost;
}
```

```
// Input: Cost Matrix of Job Assignment problem
// Output: Optimal cost and Assignment of Jobs
algorithm findMinCost (costMatrix mat[][])
{
  // Initialize list of live nodes(min-Heap)
  // with root of search tree i.e. a Dummy node
  while (true)
  {
    // Find a live node with least estimated cost
    E = Least();

    // The found node is deleted from the list
    // of live nodes
    if (E is a leaf node)
    {
      printSolution();
      return;
    }

   for each child x of E
   {
      Add(x); // Add x to list of live nodes;
      x->parent = E; // Pointer for path to root
   }
  }
}
```

Code:

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

#define N 3

struct Node
{
	// stores parent node of current node
	// helps in tracing path when answer is found
	Node* parent;

	// contains cost for ancestors nodes
	// including current node
	int pathCost;

	// contains least promising cost
	int cost;

	// contain worker number
	int workerID;

	// contains Job ID
	int jobID;
```

```
        // Boolean array assigned will contains
        // info about available jobs
        bool assigned[N];
};

// Function to allocate a new search tree node
// Here Person x is assigned to job y
Node* newNode(int x, int y, bool assigned[], Node* parent)
{
        Node* node = new Node;

        for (int j = 0; j < N; j++)
                node->assigned[j] = assigned[j];
        node->assigned[y] = true;

        node->parent = parent;
        node->workerID = x;
        node->jobID = y;

        return node;
}

// Function to calculate the least promising cost
// of node after worker x is assigned to job y.
```

```
int calculateCost(int costMatrix[N][N], int x, int y, bool assigned[])
{
        int cost = 0;
        // to store unavailable jobs
        bool available[N] = {true};

        // start from next worker
        for (int i = x + 1; i < N; i++)
        {
                int min = INT_MAX, minIndex = -1;

                // do for each job
                for (int j = 0; j < N; j++)
                {
                        // if job is unassigned
                        if (!assigned[j] && available[j] &&
                        costMatrix[i][j] < min)
                        {
                                // store job number
                                minIndex = j;
                                // store cost
                                min = costMatrix[i][j];
                        }
                }
                // add cost of next worker
                cost += min;

                // job becomes unavailable
                available[minIndex] = false;
        }
        return cost;
}

// Comparison object to be used to order the heap
struct comp
{
        bool operator()(const Node* lhs, const Node* rhs) const
        {
                return lhs->cost > rhs->cost;
        }
};

// print Assignments
void printAssignments(Node *min)
{
        if(min->parent==NULL)
                return;
```

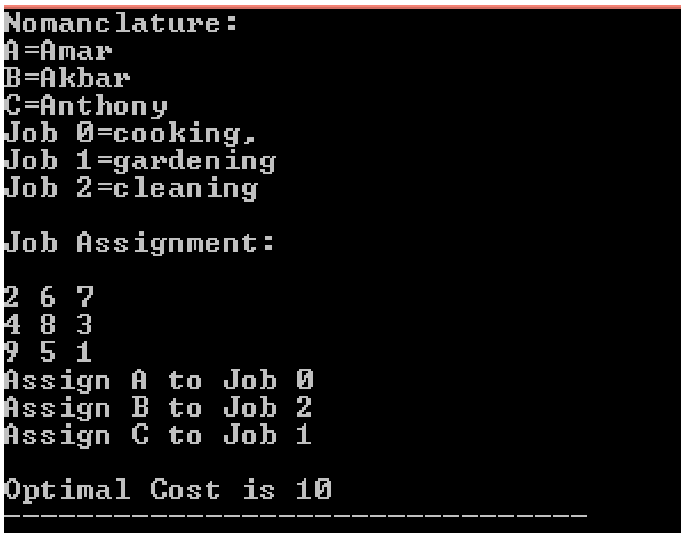
```
    printAssignments(min->parent);
    cout << "Assign " << char(min->workerID + 'A')
    << " to Job " << min->jobID << endl;
}

// Finds minimum cost using Branch and Bound.
int findMinCost(int costMatrix[N][N])
{
    // Create a priority queue to store live nodes of
    // search tree;
    priority_queue<Node*, std::vector<Node*>, comp> pq;

    // initialize heap to dummy node with cost 0
    bool assigned[N] = {false};
    Node* root = newNode(-1, -1, assigned, NULL);
    root->pathCost = root->cost = 0;
    root->workerID = -1;

    // Add dummy node to list of live nodes;
    pq.push(root);

    // Finds a live node with least cost,
    // add its childrens to list of live nodes and
    // finally deletes it from the list.
```

```
while (!pq.empty())
{
    // Find a live node with least estimated cost
    Node* min = pq.top();

    // The found node is deleted from the list of
    // live nodes
    pq.pop();

    // i stores next worker
    int i = min->workerID + 1;

    // if all workers are assigned a job
    if (i == N)
    {
        printAssignments(min);
        return min->cost;
    }

    // do for each job
    for (int j = 0; j < N; j++)
    {
        // If unassigned
        if (!min->assigned[j])
```

```
                    {
                              // create a new tree node
                              Node* child = newNode(i, j, min->assigned, min);

                              // cost for ancestors nodes including current node
                              child->pathCost = min->pathCost + costMatrix[i][j];

                              // calculate its lower bound
                              child->cost = child->pathCost +
                              calculateCost(costMatrix, i, j, child->assigned);

                              // Add child to list of live nodes;
                              pq.push(child);
                    }
          }
     }
}

// Driver code
int main()
{
     int costMatrix[N][N];
     int i,j;
     cout << "Nomanclature:\n";
```

```
        cout << "A=Amar\nB=Akbar\nC=Anthony\n";
        cout << "Job 0=cooking,\nJob 1=gardening\nJob 2=cleaning\n\n";
        cout << "Job Assignment:\n\n";
         for(i=0;i<N;i++)
         {
                for(j=0;j<N;j++)
                {
                        cin>>costMatrix[i][j];
                }
        }
        cout << "\nOptimal Cost is "<< findMinCost(costMatrix);
        return 0;
}
```

OUTPUT:



```
Nomanclature:
A=Amar
B=Akbar
C=Anthony
Job 0=cooking,
Job 1=gardening
Job 2=cleaning

Job Assignment:

2 6 7
4 8 3
9 5 1
Assign A to Job 0
Assign B to Job 2
Assign C to Job 1

Optimal Cost is 10
--------------------------------
```

Conclusion:

I have successfully performed the job assignment problem using ‘Branch & Bound’ and had taken out the output of the given test cases. I have learn about Branch and bound is an algorithm design paradigm for discrete and combinatory optimization problems, as well as mathematical optimization.