

# LP III Lab Manual

Name : Vaishnavi .R. Borse (A2)

Roll no: 20

1. Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyse their time and space complexity.

Code:

```
// C++ program to count Fibonacci numbers in given range
#include <bits/stdc++.h>
using namespace std;

// Returns count of fibonacci numbers in [low, high]
int countFibs(int low, int high)
{
    // Initialize first three Fibonacci Numbers
    int f1 = 0, f2 = 1, f3 = 1;

    // Count fibonacci numbers in given range
    int result = 0;

    while (f1 <= high)
    {
        if (f1 >= low)
            result++;
        f1 = f2;
        f2 = f3;
        f3 = f1 + f2;
    }

    return result;
}

// Driver program
```

```
int main()
{
    int low = 10, high = 100;
    cout << "Count of Fibonacci Numbers is "
        << countFibs(low, high);
    return 0;
}
```

Output: Count of Fibonacci Numbers is 5

# LP III Lab Manual

**Name :** Vaishnavi .R. Borse (A2)

**Roll no:** 20

2. Write a program to solve a fractional Knapsack problem using a greedy method

CODE:

```
// C++ program to solve fractional Knapsack Problem
```

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

```
using namespace std;
```

```
// Structure for an item which stores weight and
```

```
// corresponding value of Item
```

```
struct Item {
```

```
    int profit, weight;
```

```
    // Constructor
```

```
    Item(int profit, int weight)
```

```
    {
```

```
        this->profit = profit;
```

```
        this->weight = weight;
```

```
}
```

```
};
```

```
// Comparison function to sort Item  
// according to profit/weight ratio  
static bool cmp(struct Item a, struct Item b)  
{  
    double r1 = (double)a.profit / (double)a.weight;  
    double r2 = (double)b.profit / (double)b.weight;  
    return r1 > r2;
```

```
}
```

```
// Main greedy function to solve problem  
double fractionalKnapsack(int W, struct Item arr[], int N)
```

```
{
```

```
    // Sorting Item on basis of ratio  
    sort(arr, arr + N, cmp);
```

```
    double finalvalue = 0.0;
```

```
    // Looping through all items
```

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

```
        // If adding Item won't overflow,  
        // add it completely
```

```
if (arr[i].weight <= W) {  
    W -= arr[i].weight;  
    finalvalue += arr[i].profit;  
}  
  
// If we can't add current Item,  
// add fractional part of it  
else {  
    finalvalue  
    += arr[i].profit  
    * ((double)W / (double)arr[i].weight);  
    break;  
}  
  
}  
  
// Returning final value  
return finalvalue;  
}  
  
// Driver code  
int main()  
{  
    int W = 50;  
    Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } };
```

```
int N = sizeof(arr) / sizeof(arr[0]);  
  
// Function call  
cout << fractionalKnapsack(W, arr, N);  
return 0;  
}
```

Output:

240

# LP III Lab Manual

**Name :** Vaishnavi .R. Borse (A2)

**Roll no:** 20

3. Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

**Code:**

```
#include <iostream>
```

```
#include <algorithm>
```

```
#include <vector>
```

```
#include <queue>
```

```
using namespace std;
```

```
class Item {
```

```
public:
```

```
    int value;
```

```
    int weight;
```

```
    double ratio;
```

```
Item(int value, int weight) {
```

```
    this->value = value;
```

```
    this->weight = weight;  
    this->ratio = (double)value / weight;  
}  
};
```

```
class KnapsackNode {
```

```
public:  
    vector<int> items;  
    int value;  
    int weight;
```

```
KnapsackNode(vector<int> items, int value, int weight) {
```

```
    this->items = items;  
    this->value = value;  
    this->weight = weight;  
}  
};
```

```
class Knapsack {
```

```
public:  
    int maxWeight;  
    vector<Item> items;
```

```
Knapsack(int maxWeight, vector<Item> items) {
```

```
    this->maxWeight = maxWeight;  
    this->items = items;  
}  
  
  
int solve() {  
    sort(this->items.begin(), this->items.end(), [](const Item&  
a, const Item& b) {  
        return a.ratio > b.ratio;  
    });  
  
  
    int bestValue = 0;  
    queue<KnapsackNode> q;  
    q.push(KnapsackNode({}, 0, 0));  
  
  
    while (!q.empty()) {  
        KnapsackNode node = q.front();  
        q.pop();  
        int i = node.items.size();  
  
  
        if (i == this->items.size()) {  
            bestValue = max(bestValue, node.value);  
        } else {  
            Item item = this->items[i];  
    }
```

```

        KnapsackNode withItem(node.items,
node.value + item.value, node.weight + item.weight);

        if (isPromising(withItem, this->maxWeight,
bestValue)) {

            q.push(withItem);

        }

        KnapsackNode withoutItem(node.items,
node.value, node.weight);

        if (isPromising(withoutItem, this->maxWeight,
bestValue)) {

            q.push(withoutItem);

        }

    }

    return bestValue;
}

bool isPromising(KnapsackNode node, int maxWeight, int
bestValue) {

    return node.weight <= maxWeight && node.value +
getBound(node) > bestValue;

}

int getBound(KnapsackNode node) {

```

```
int remainingWeight = this->maxWeight - node.weight;
int bound = node.value;

for (int i = node.items.size(); i < this->items.size(); i++) {
    Item item = this->items[i];

    if (remainingWeight >= item.weight) {
        bound += item.value;
        remainingWeight -= item.weight;
    } else {
        bound += remainingWeight * item.ratio;
        break;
    }
}

return bound;
};

Output:
Best value: 220
```

# LP III Lab Manual

Name : Vaishnavi .R. Borse (A2)

Roll no: 20

4. Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen's matrix.

Code

```
// C++ program to solve N Queen Problem using backtracking
```

```
#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++)  
            if(board[i][j])  
                cout << "Q ";  
            else cout << ". ";  
            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;
}

```

## Output

```

. . Q .
Q . . .
. . . Q
. Q . .

```

# LP III Lab Manual

Name : Vaishnavi .R. Borse (A2)

Roll no: 20

5. Write a program for analysis of quick sort by using deterministic and randomized variant

Code

```
// C++ implementation QuickSort
// using Lomuto's partition Scheme.

#include <cstdlib>
#include <time.h>
#include <iostream>
using namespace std;

// This function takes last element
// as pivot, places
// the pivot element at its correct
// position in sorted array, and
// places all smaller (smaller than pivot)
// to left of pivot and all greater
// elements to right of pivot
int partition(int arr[], int low, int high)
{
    // pivot
    int pivot = arr[high];

    // Index of smaller element
    int i = (low - 1);
```

```

for (int j = low; j <= high - 1; j++)
{
    // If current element is smaller
    // than or equal to pivot
    if (arr[j] <= pivot) {

        // increment index of
        // smaller element
        i++;
        swap(arr[i], arr[j]);
    }
}
swap(arr[i + 1], arr[high]);
return (i + 1);
}

// Generates Random Pivot, swaps pivot with
// end element and calls the partition function
int partition_r(int arr[], int low, int high)
{
    // Generate a random number in between
    // low .. high
    srand(time(NULL));
    int random = low + rand() % (high - low);

    // Swap A[random] with A[high]
    swap(arr[random], arr[high]);

    return partition(arr, low, high);
}

/* The main function that implements
QuickSort

```

```
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high) {

        /* pi is partitioning index,
        arr[p] is now
        at right place */
        int pi = partition_r(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout<<arr[i]<<" ";
}

// Driver Code
int main()
{
    int arr[] = { 10, 7, 8, 9, 1, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
quickSort(arr, 0, n - 1);
printf("Sorted array: \n");
printArray(arr, n);

return 0;
}
```

Output

**Sorted array:**

1 5 7 8 9 10