Assignment No : 1.

Title :- Calculate Fibonacci number and its Step Count

Problem Statement :- Develop a Program in C++ to create a Fibonacci Series.

Prerequisites :- Design and analysis Algorithm.

objectives :- Implement recursive and non-recursive program to calculate. Fibonacci number find Step count.

outcome :- To understand the implementation of Fibonacci and analysis and find its Step count.

Theory Fibonacci Series :-

Fibonacci Series is defined as a Sequence of numbers. in Which the first two numbers are 1 and 1 or 0 and 1 depending on the selected beginning point of Sequence and each subsequence number is Sum of previous two. So, in this Series the n^{th} term is the Sum of $(n-1)^{th}$ term and $(n-2)^{nd}$ term.

Fibonacci Series in C++ - In Case of Fibonacci Series, next no. is the Sum of previous two numbers for example 0, 1, 1, 2, 3, 5, 8, 13, 21 etc.

The first two no. of Fibonacci Series are 0 and 1.

There are two ways to which Write Fibonacci Series program.

1) without Recursion.
2) with Recursion.

Generation of Fibonacci Series:-

mathematically, the n^{th} term of the Fibonacci Series can be
$t_n = t_{n-1} + t_{n-2}$

$2e_5$	$2e_6$	$2e_7$	$2e_8$	$2e_9$	$2e_{10}$
5	8	13	21	34	55

$2e_8 = 2e_7 + 2e_6$.

facilities : linux operating system, etc

Algorithm :-

Step1 : Start
Step2 : Declare Variable i, a, b, show
Step3 : Initialize the Variable, a=0, b=1 and show=0.
Step4 : Enter the no. of terms of Series to be printed, p
two no. of Series.
Step5 : use the following steps.
=> show = a+b
=> a=b
=> b=show
=> increase value of i each time by 1.
=> print the value of show.

Step 6 : End.
I/P :- Take the term no. as an input. Say it is 10.
O/P :- Enter the no. of elements : 14.
Conclusion : Hence we studied the concept of Fibona

Assignment No:2.

Title:- Solve a fractional knapsack problem using a greedy method.

Problem Statement:- Given a set of n items, each having value with weight w and the total capacity of a knapsack. The task is to find the maximum value of fractions of items that can fit into the knapsack.

Objective:- The objective is to fill the knapsack to some given volume with different such that value of selected item is maximized.

Outcome:- To understand the implementation of greedy method using fraction knapsack problem.

Theory:- Given a set of items, each with weight and value, determine a subset of volume to include in a collection so that the total weight is less or equal to a given limit and the total value is large as possible. The knapsack problem is in combinational optimization problem. It appears as a subproblem in many more complex mathematical model of real world problems.

Greedy Approach:- In this, we calculate the profit/weight and accordingly, we will select the item with the highest ratio would be selected first.

Fractional knapsack problem bajn in c++ and Java.

- We have taken and array of structure named item.
- Each item has value and weight
- We are calculating density = value/weight for each in order of decreasing density.

Find Approach:-

objective	profit	weight	Remaining Weight
3	15	5	15-5 = 10
2	10	3	10-3 = 7
6	9	3	7-3 = 4
5	8	1	4-1 = 3
7	7*3/4 = 5.25	3	3-3 = 0.

The total profit would be equal to
(15+10+9+8+5.25) = 47.25.

Second Approach:

objective	profit	weight	Remaining weight.
1	5	1	15-1 = 14
5	7	1	14-1 = 13
7	4	2	13-2 = 11
2	10	3	11-3 = 8.

(5+7+4+10+9+7+3) = 46.

Third approch: In this case we will calculate the ratio profit/weight:

object :	1	2	3	4	5	6	7
profit(p):	5	10	10	15	7	8	4
weight :	1	3	5	4	1	3	2

object 1 :: 5/1 : 5
object 2 :: 10/3 : 3.3
object 3 :: 15/5 : 3
object 4 :: 7/4 : 1.7
object 5 :: 8/1 : 8
object 6 :: 9/3 : 3
object 7 :: 4/2 : 2

p·w : 5 3.3 3 1.7 8 3 2.

In this approch we still select object based on maximum profit/weight ratio since the p/w of object 5 is maximum so we select object 5.

Algorithm:

(W[1--n], P[1--n], W)

for i:=1 to n
do x[i] := 0
weight := 0
for i:=1
if weight + w[i] ≤ W then
x[i] := 1.

```
weight = weight + w[i]
else
x[i] = [w - weight] / w[i]
  weight = w
  break
  return x
```

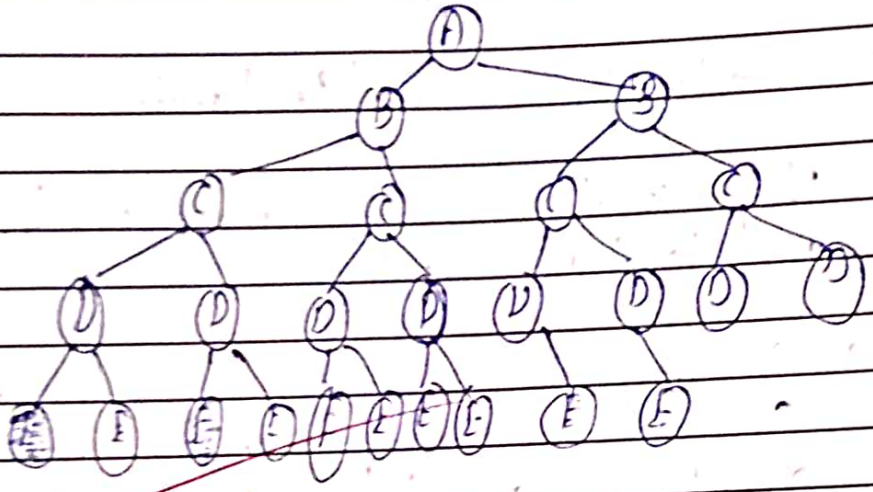
I/p: A[] = { {60, 15}, {100, 20}, {120, 30} }
Total capacity = 50

O/p: Enter total & item value and weight.
Enter 1 value 60
Enter 1 weight 10
enter 2 value 100
enter 2 weight 20
enter 3 value 120
enter 3 weight 30.

Enter data: value 60 100 120
weight 10 20 30.
Enter knapsack weight 50
total value in bag 50
... max value for 50 weight is 240
Process returned 0 (0x0) execution time : 17.376 s.

Conclusion: Hence, we have successfully studied concept problem using greedy method.

The backtracking based soln. works better than brute force by ignoring infeasible soln.



(E) - Ignored because best soln is infeasible

(D) - Ignored because of best soln is worst than current soln.

Branch and Bound is very useful technique for searching a soln but in worst case.

Algorithm

Step 1: Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy algorithm.

Step 2: Initialize max profit, max profit: 0.

Step 3: Creat an empty queue, Q.

Step 4: Creat a dummy node of decision tree and enqueue it to Q. profit, weight of dummy node are 0.

Step5:- Do following while Q is not empty.

Step6:- Extract an item type Q. Let the extracted

Step7:- Compute profit of next level node.

Step8:- If the profit is more than max. profit, then next level node to Q.

Step9:- Consider the case where next level node Considered as part of Soln and add a no with level as next, but weight and profit with Considering next level nodes.

Ip:- Input the number of items 4.

Enter the weight and profit of the item 1: 2
enter the weight and profit of the item 2: 1
enter the weight and profit of the item 3: 3
enter the weight and profit of the item 4: 2

enter the capacity of knapsack: 5.

Op:- The table is.

0	0	0	0	0	0
0	0	12	12	12	12
0	10	12	22	22	22
0	10	12	22	30	32
0	10	15	25	30	37

Assignment No: 3.

Title: Write a program to solve a 0-1 knapsack problem using dynamic programming or branch and bound strategy.

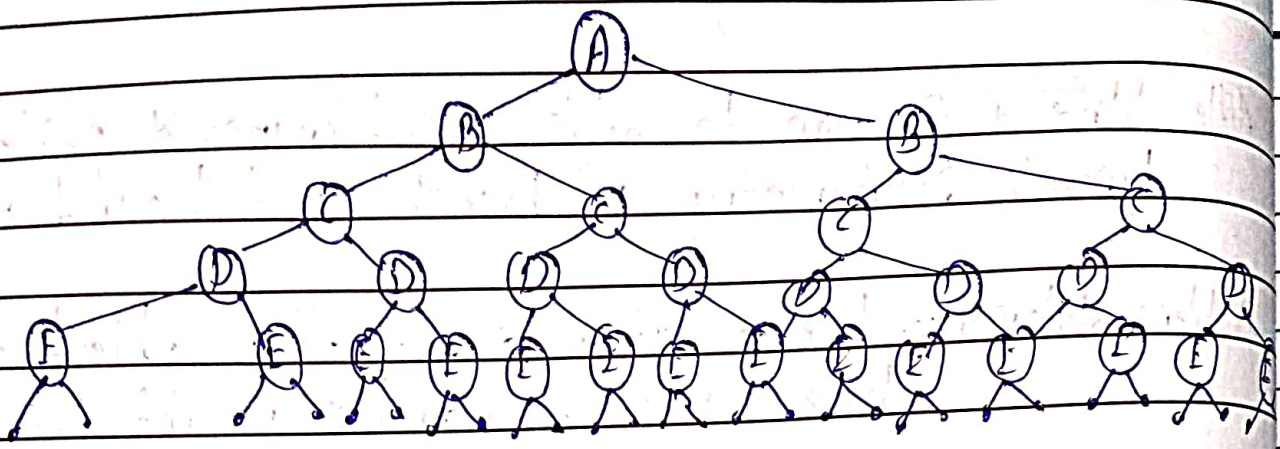
Problem statement: We are given a set of n object which have a value v_i and a weight w_i.

objective: Implement 0/1 knapsack problem using dynamic programming or branch and bound strategy.

Theory: Branch and bound is an algorithm design paradigm which is generally used for solving combinational optimization problem. These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.

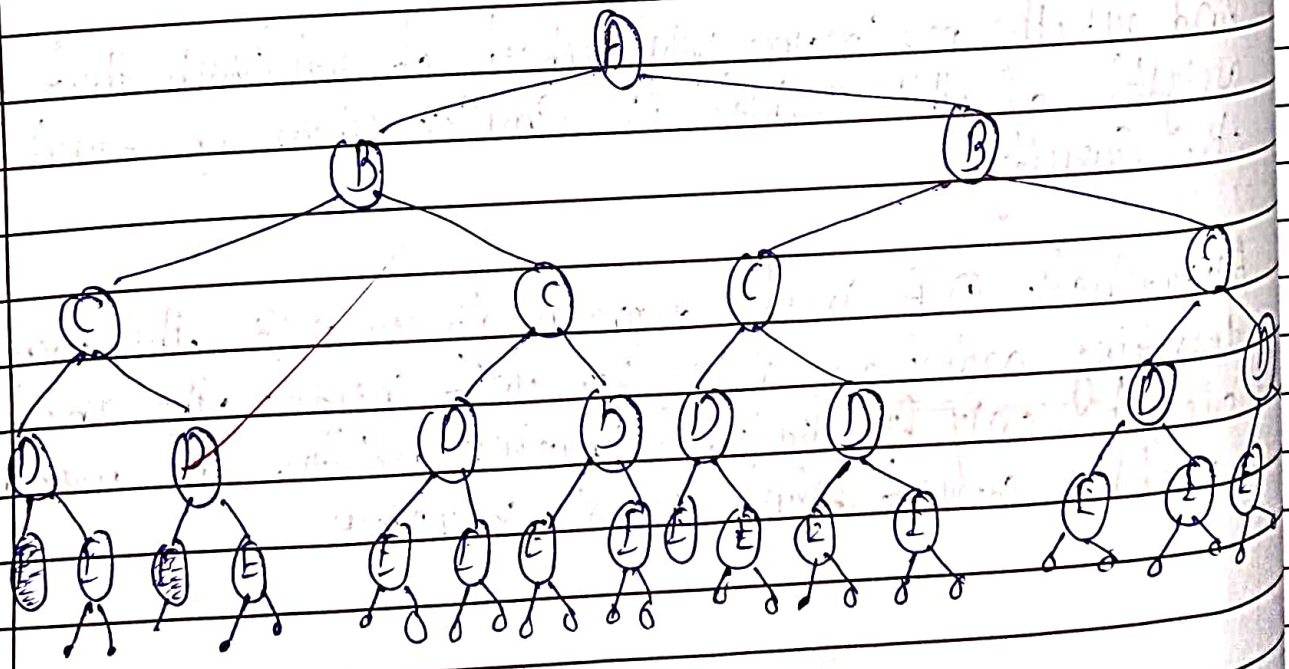
Find out the maximum value subset of w[] such that sum of weights of this subset is smaller than or equal to knapsack capacity.

Approaches: ① A Greedy approach is to pick the items in decreasing order of value per unit weight the greedy approach weight only for fractional knapsack problem and may not produce correct result for 0/1 knapsack.



We can use backtracking to optimize the brute force ap In the tree representation, we can do DFS tree. If we a point when a soln no longer is feasible, there is no to continue exploring. In the given example backtracking be much more effectively if we had even more items smaller knapsack capacity.

Backtracking:



The maximum profit is :37

The most valuable subset is:
{item:4: item 2: item 1: }

Conclusion:- We can see that branch and bound method will give the soln to the problem by exploring just 1 Path in the best case.

Assignment No: 5

Title: Design & queque matrix having first queen placed. use backtracking to place remaining queens to generate the final 8 queen matrix.

Problem Statement: Find an arrangement of 8 queen on a chess board, such that no queen can attack any other queen on the board.

objective: Implement 8 queen using backbacking.

outcomes: To understand the implementation of 8 queen using backbacking.

Theory: In backbacking we start with one possible move out of many available moves. We then try to solve the problem with selective moves and print the soln.

This problem is commonly seen for n=4 and n=8. let us look at an example where N=8 before solving the problem you need to know about the movement of the queen in chess. A queen can move any no. of steps in any direction. The only constraint is that we can't change its direction while it is moving.

$$i-j = k-l \quad \text{---} (1)$$
$$\text{or} \quad i+j = k+l \quad \text{---} (2)$$

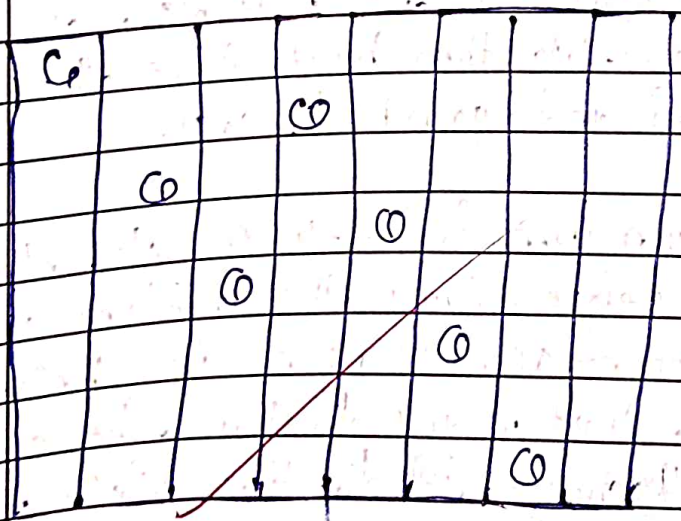
from (1) and (2) implies $j-l = i-k$ and $j-l = k-i$

- Two queens is on the same diagonal if $|(j-1) = (j-k)|$. Soln of 8 queen problem can be obtained to soln of 6 queen problem.

$x_1 = 3$ $x_2 = 6$ $x_3 = 2$ $x_4 = 7$ $x_5 = 1$ $x_6 = 4$ $x_7 = 8$ $x_8 = 5$.

- place (k, j) return a boolean value that is true if kth queen placed in column if tells both whether or i is distinct for all previous call $x_1, x_2 \ldots x_{k-1}$ and whether there is no other queen on same diagonal using place we give precise soln to the n-queen problem.

The soln can be shown as -



Time Complexity analysis.

- The possible method take o(n) time.
- For each innovation of loop in n-queen helper it runs o(n) time.

$T(n) = O(n!) + n^2 T(n-1)$

Algorithm:-

START.

1) Begin from the leftmost column.
2) If all queen are placed return the true/print configuration.
3) check for all row in current column.

a) If queen placed mark row and column and recursively check if the approch in the currently configuration do we obtain soln or not.

b) If recursion yeild a soln return true.

c) If all row tried and soln not obtain return false and backtrack.

END.

Conclusion:- Hence we learnt about 8-queen algorithm.

Assignment No: 5.

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

Problem Statement:: Write a fun^n that takes two parameters n and k: and return the value of Binomial Coefficient C(n,k).

Objative:: Implement deterministic : Quick Sort

Theory::

In Deterministic Quick Sort is applied to an array of length n whose entries are already sorted, then this algorithm uses $\Omega(n^2)$ step.
It comes down to how the pivotes are chosen in deterministic quickSort, the pivot are chosen by either always choosing the pivot at same relative index. For instance, a common method is to choose the median of first, last and middle element. as pivot.

array [1,2,3,4,5,6,7,8,9,10,9,8,6,5,4,3,2,1].

partition the array then Recursively sort the piece before and after the pivot element.

```
Quick Sort (A, p, r)
if p < r then.
q = Partion (A, p, r); // Rearrange A[p..r] in place:
```

QuickSort (A, p, q-1)
QuickSort (A, p+1, r)

Divide and Conquer.

a) Divide - partition the array A[p..r] into two subarrays A[p...q-1] & A[q+1, r] such that.
- A[x] <= A[q] for all x in [p...q-1].
- A[x] >= A[q] for all x in [q+1]r]

b) Conquer: Recursively sort A[p...q-1] + A[q+1r].

c) Combine: nothing to do there.

Randomized Quick Sort:

We expect good average Case behaviour if all i/p perm are equally likely, but what if it is not? - we tend the algorithm to get the same effect as if the were random. Instead of explicitely permuting d randomization can be accomplished tirelily by ra sampling of one of array element as pivot.

Randomized-QuickSort (A, p, r).

i) if: p < r.
ii) q := Randomized partition (A, p, r).

iii) Randomized - QuickSort (A, p, q-1)
iv) Randomized - QuickSort (A, q+1, r).

Randomized - Partition (A, p, r).
i) i = random (p, r)
ii) exchange A[r] with A[i].
iii) return PARTITION (A, p, r).

Randomized QuickSort analysis-

The analysis assumes that all element are unique but with some work can be generalized to remove this assumption.

$T(n)$: max $T(q) + T(n-q-1) + \Theta(n)$
$0 \le q \le n-1$.

Based on informal analysis we guess $T(n) \le n^2$ for some. Substitute this guess into the recurrence.

$T(n) \le \max (q^2 + (n-q-1)^2) + \Theta(n)$
$0 \le q \le n-1$.

$= c \cdot \max (q^2 + (n-q-1)^2) + \Theta(n)$.
$0 \le q \le n-1$

The maximum value of $q^2 + (n-q-1)^2$ occur when q is either 0 or $n-1$ and has value $(n-1)^2$ in either case.

max $(q^2 + (n-q-1)^2) \leq (n-1)^2$
$0 \leq q \leq n-1$ $= n^2 - 2n + 2$

Substituting this back into the recurrence.

$A(n) \leq cn^2 - (2n-1) + \Theta(n)$
$\leq cn^2$.

We can pick c so that $c(2n-1)$ dominated $\Theta(n)$. Thus the worst case running time is $O(n^2)$ one can also show that recurrence is $\Omega(n^2)$, so worst case is

Conclusion:
Here, we have studied and implemented the analysis of quick sorting deterministic and randomized variant.