LP-III DAA (2023-24) **Assignment 1 :** Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity.	
Students Name:	**Roll No :**
Batch :	**BE Computer Div -**

Title: Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity.

Aim: To perform analysis of Time and Space Complexity of Fibonacci numbers

Prerequisites: Data Structures, Algorithms

Theory:

Iterative Fibonacci number algorithms

```
  Algorithm Fibo (n)
{
int n1=0,n2=1,n3,i,number;
IF (n<2)
printf(n1,n2)
Else
{
 for(i=2;i<number;++i)//loop starts from 2 because 0 and 1 are already printed
{  n3=n1+n2;
 printf(" %d",n3);
n1=n2;
n2=n3;  }
}
}
```

Two cases (1) n = 0 or 1 and (2) n > 1.

1) When n = 0 or 1, lines 3 and 4 get executed once each. Since each line has an s/e of 1, the total step count for this case is **2**.

2) When n > 1, then For loop will executes 2 to n-1 times and complexity will be O(n)

Iterative Program

```
1.  #include<stdio.h>
2.  int main()
3.  {
4.   int n1=0,n2=1,n3,i,number;
5.   printf("Enter the number of elements:");
6.   scanf("%d",&number);
7.   printf("\n%d %d",n1,n2);//printing 0 and 1
8.   for(i=2;i<number;++i)//loop starts from 2 because 0 and 1 are already printed
9.   {
10.   n3=n1+n2;
11.   printf(" %d",n3);
12.   n1=n2;
13.   n2=n3;
14. }
15.  return 0;
16. }
```

Recursive Algorithm to find nth term

```
Algorithm  rFibonacci(n)
{
    if (n <= 1)
        return n;
    else
        return rFibonacci(n - 1) + rFibonacci(n - 2); }
```

Analysis

$T(n) = T(n-1) + T(n-2) + c$

$= 2T(n-1) + c$ //from the approximation $T(n-1) \sim T(n-2)$

$= 2*(2T(n-2) + c) + c$

$= 4T(n-2) + 3c$
$= 8T(n-3) + 7c$
$= 2^k * T(n - k) + (2^k - 1)*c$
Let's find the value of k for which: $n - k = 0$
$k = n$
$T(n) = 2^n * T(0) + (2^n - 1)*c$
$= 2^n * (1 + c) - c$
$T(n) = 2^n$

Recursive Program

```
def fibonacci(n):
   if(n <= 1):
      return n
   else:
      return(fibonacci(n-1) + fibonacci(n-2))
n = int(input("Enter number of terms:"))
print("Fibonacci sequence:")
for i in range(n):
   print(fibonacci(i))
```

Questions:

1. Compare iterative Vs recursive Algorithms.
2. Discuss space complexity of Fibonacci seris
3. Explain logic of Fibonacci series iterative and recursive
4. Attach printout of programs

LP-III DAA (202-24) **Assignment 2** Implementation of Huffman Encoding using a greedy strategy	
Students Name:	**Roll No :**
Batch :	**BE Computer Div -**

Aim : Write a program to implement Huffman Encoding using a greedy strategy.

Theory :

Greedy Strategy : In an optimization problem, we are given an input and asked to compute a structure, subject to various constraints, in a manner that either minimizes cost or maximizes profit. Such problems arise in many applications of science and engineering. Given an optimization problem, we are often faced with the question of whether the problem can be solved efficiently (as opposed to a brute-force enumeration of all possible solutions), and if so, what approach should be used to compute the optimal solution? In many optimization algorithms a series of selections need to be made. A simple design technique for optimization problems is based on a greedy approach, that builds up a solution by selecting the best alternative in each step, until the entire solution is constructed. When applicable, this method can lead to very simple and efficient algorithms. (Unfortunately, it does not always lead to optimal solutions.) Today, we will consider one of the most well-known examples of a greedy algorithm, the construction of Huffman codes.

Huffman Codes: Huffman codes provide a method of encoding data efficiently. Normally when characters are coded using standard codes like ASCII or the Unicode, each character is represented by a fixed-length codeword of bits (e.g., 8 or 16 bits per character). Fixed-length codes are popular, because its is very easy to break a string up into its individual characters, and to access individual characters and substrings by direct indexing. However, fixed-length codes may not be the most efficient from the perspective of minimizing the total quantity of data.

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

See this for applications of Huffman Coding.

There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

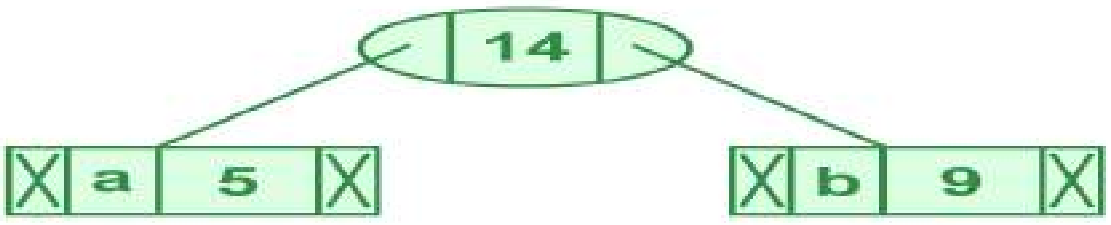
1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.
   Let us understand the algorithm with an example:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

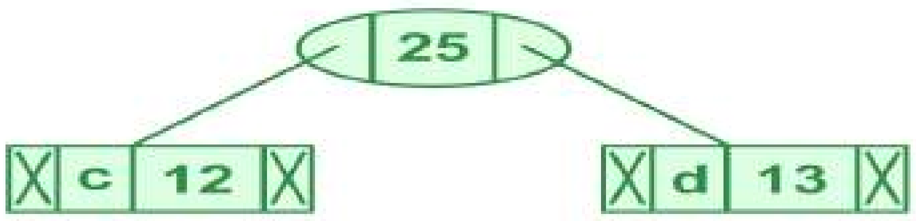
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

```
character            Frequency

       c                12

       d                13

 Internal Node          14

       e                16

       f                 45
```

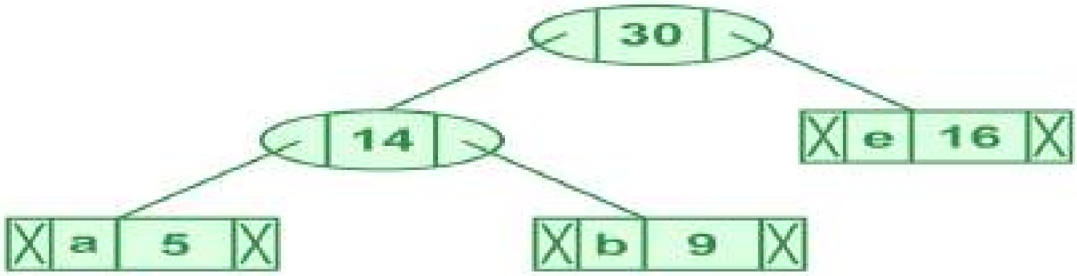
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

```
character           Frequency

Internal Node          14

       e               16

Internal Node          25

       f               45
```

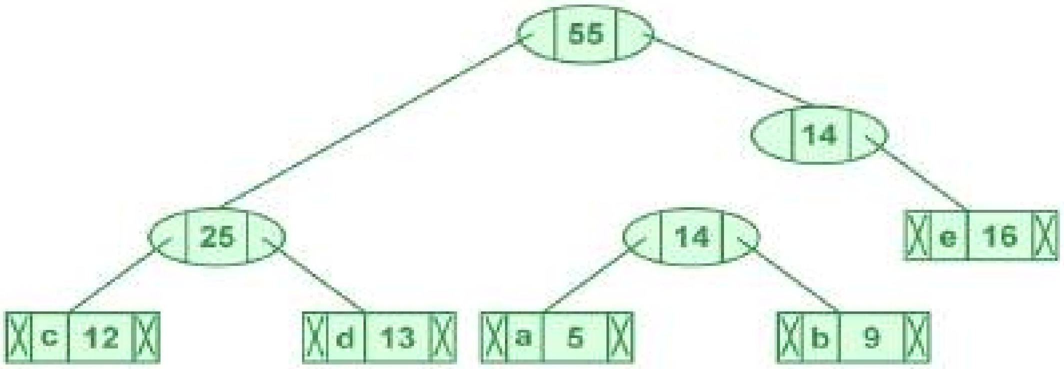
Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



Now min heap contains 3 nodes.

```
character          Frequency
Internal Node         25
Internal Node         30
      f               45
```

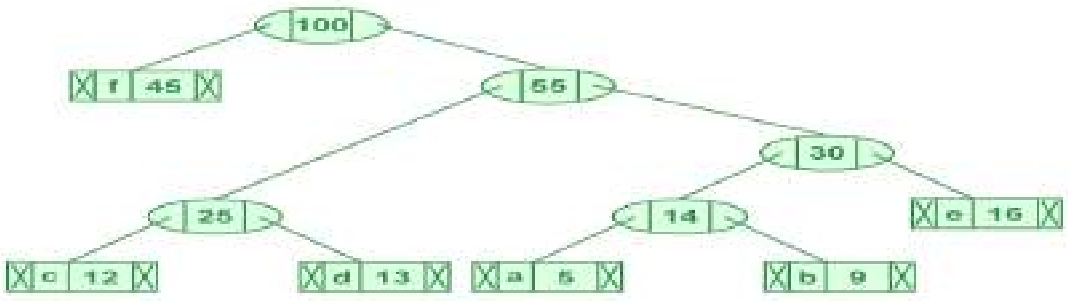
Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



Now min heap contains 2 nodes.

```
character     Frequency
       f         45
Internal Node    55
```

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100

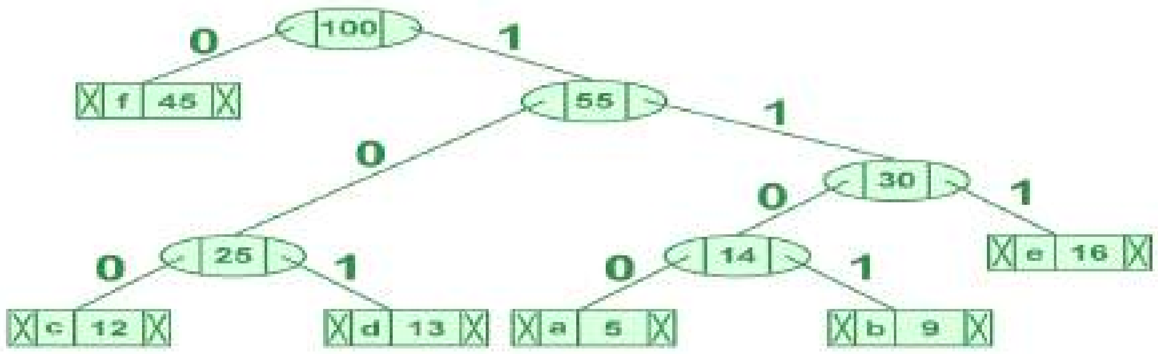


Now min heap contains only one node.

```
character      Frequency

Internal Node    100
```

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:
Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

```
character   code-word

    f          0

    c          100

    d          101

    a          1100

    b          1101

    e          111
```

Time complexity: O(nlogn) where n is the number of unique characters. If there are n nodes, extractMin() is called 2*(n – 1) times. extractMin() takes O(logn) time as it calles minHeapify(). So, overall complexity is O(nlogn).
If the input array is sorted, there exists a linear time algorithm. We will soon be discussing in our next post.

Applications of Huffman Coding:

1. They are used for transmitting fax and text.
2. They are used by conventional compression formats like PKZIP, GZIP, etc.
3. Multimedia codecs like JPEG, PNG, and MP3 use Huffman encoding(to be more precise the prefix codes).

Questions :

1. With another example explain Huffman coding.
2. Write an algorithm for Huffman coding
3. Derive the complexity of Huffman coding
4. Write control abstraction of Greedy strategy

LP-III DAA (2023-24) Assignment 3 Fractional Knapsack Problem Using Greedy Method	
Students Name:	**Roll No :**
Batch :	**BE Computer Div -**

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

Theory :

The Greedy algorithm could be understood very well with a well-known problem referred to as Knapsack problem. Although the same problem could be solved by employing other algorithmic approaches, Greedy approach solves Fractional Knapsack problem reasonably in a good time. Let us discuss the Knapsack problem in detail.

Knapsack Problem

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

Problem Scenario

A thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are n items available in the store and weight of i^{th} item is w_i and its profit is p_i. What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

Fractional Knapsack

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are **n** items in the store
- Weight of **i**th item wi>0wi>0
- Profit for **i**th item pi>0pi>0 and
- Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of **i**th item.

0⩽xi⩽10⩽xi⩽1

The **i**th item contributes the weight xi.wixi.wi to the total weight in the knapsack and profit xi.pixi.pi to the total profit.

Hence, the objective of this algorithm is to

maximize∑n=1n(xi.pi)maximize∑n=1n(xi.pi)

subject to constraint,

∑n=1n(xi.wi)⩽W∑n=1n(xi.wi)⩽W

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

∑n=1n(xi.wi)=W∑n=1n(xi.wi)=W

In this context, first we need to sort those items according to the value of piwipiwi, so that pi+1wi+1pi+1wi+1 ≤ piwipiwi . Here, ***x*** is an array to store the fraction of items.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
do x[i] = 0
weight = 0
for i = 1 to n
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
```

Analysis

If the provided items are already sorted into a decreasing order of piwipiwi, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

Example

Let us consider that the capacity of the knapsack ***W* = 60** and the list of provided items are shown in the following table –

Item	**A**	**B**	**C**	**D**
Profit	280	100	120	120
Weight	40	10	20	24
Ratio (piwi)/(piwi)	7	10	6	5

As the provided items are not sorted based on piwipiwi. After sorting, the items are as shown in the following table.

Item	**B**	**A**	**C**	**D**
Profit	100	280	120	120

Weight	10	40	20	24
Ratio (piwi)(piwi)	10	7	6	5

Solution

After sorting all the items according to piwipiwi. First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.

Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**

And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

Questions:

1. Discuss the term feasible solution
2. Solve any other example of fractional knapsack problem
3. Discuss time complexity of fractional knapsack problem
4. State applications of knapsack problem

LP-III DAA (2023-24) Assignment 4 **0-1 Knapsack Problem using Dynamic Programming OR Branch and bound**	
Students Name:	**Roll No :**
Batch :	**BE Computer Div -**

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

Theory : What is the 0/1 knapsack problem (Using Merging and purging of Pairs)

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

Consider the problem having weights and profits are:

Weights: {3, 4, 6, 5}

Profits: {2, 3, 1, 4}

The weight of the knapsack is 8 kg

The number of items is 4

The above problem can be solved by using the following method:

x_i = {1, 0, 0, 1}

= {0, 0, 0, 1}

= {0, 1, 0, 1}

The above are the possible combinations. 1 denotes that the item is completely picked and 0 means that no item is picked. Since there are 4 items so possible combinations will be:

2^4 = 16; So. There are 16 possible combinations that can be made by using the above problem. Once all the combinations are made, we have to select the combination that provides the maximum profit.

Another approach to solve the problem is dynamic programming approach. In dynamic programming approach, the complicated problem is divided into sub-problems, then we find the solution of a sub-problem and the solution of the sub-problem will be used to find the solution of a complex problem.

1. Set $S^0 = \{(0, 0)\}$
2. $S_1^i = \{(p, w) \mid (p – p_i) \in S^i, (w – w_i) \in S^i \}$

We can obtain S^{i+1} by invoking x_{i+1}

- If $x_i = 0$ (item x_i is excluded) then $S_1^i = S^i$
- If $x_i = 1$ (item x_i is included) then S_1^i is computed by adding (p_{i+1}, w_{i+1}) in each state of S^i.

1. S^{i+1} = MERGE_PURGE(S^i, S_1^i). MERGE_PURGE does following:

   For two pairs $(p_x, w_x) \in S^{i+1}$ and $(p_y, w_y) \in S^{i+1}$, if $p_x \le p_y$ and $w_x \ge w_y$, we say that (p_x, w_x) is dominated by (p_y, w_y). And the pair (p_x, w_x) is discarded.

   It also purges all the pairs (p, w) from S^{i+1} if $w > M$, i.e. it weight exceeds knapsack capacity.
2. Repeat step 1 n times
3. $f_n(M) = S^n$. This will find the solution of KNAPSACK(1, n, M).
4. for each pair $(p, w) \in S^n$

if $(p, w) \in S^{n-1}$, then set $x_n = 0$
if $(p, w) \notin S^{n-1}$, then set $x_n = 1$, update $p = p – x_n$ and $w = w – w_n$

Examples

Example: Solve the instance of 0/1 knapsack problem using dynamic Programming : n = 4, M = 25, $(P_1, P_2, P_3\ P_4)$ = (10, 12, 14, 16), (W_1, W_2, W_3, W_4) = (9, 8, 12, 14)

Solution:

Knapsack capacity is very large, i.e. 25, so tabular approach won't be suitable. We will use the set method to solve this problem

Initially, $S^0 = \{ (0, 0) \}$

Iteration 1:

Obtain S_1^0 by adding pair $(p_1, w_1) = (10, 9)$ to each pair of S^0

$S_1^0 = S^0 + (10, 9) = \{(10, 9)\}$

Obtain S^1 by merging and purging S^0 and S_1^0

S^1 = MERGE_PURGE (S^0, S_1^0)

= { (0, 0), (10, 9) }

Iteration 2:

Obtain S_1^1 by adding pair $(p_2, w_2) = (12, 8)$ to each pair of S^1

$S_1^1 = S^1 + (12, 8) = \{(12, 8), (22, 17)\}$
Obtain S^2 by merging and purging S^1 and S_1^1
S^2 = MERGE_PURGE(S^1, S_1^1)
= { (0, 0), (12, 8), (22, 17) }

Pair (10, 9) is discarded because pair (12, 8) dominates (10, 9)

Iteration 3:
Obtain S_1^2 by adding pair (p_3, w_3) = (14, 12) to each pair of S^2
$S_1^2 = S^2 + (14, 12)$
= { (14, 12), (26, 20), (36, 29) }

Obtain S^3 by merging and purging S^2 and S_1^2 .
S^3 = MERGE_PURGE (S^2, S_1^2)
= { (0, 0), (12, 8), (22, 17), (14, 12), (26, 20) }

Pair (36, 29) is discarded because its w > M

Iteration 4:
Obtain S_1^3 by adding pair (p_4, w_4) = (16, 14) to each pair of S^3
$S_1^3 = S^3 + (16, 14)$
= { (16, 14), (28, 22), (38, 31), (30, 26), (42, 34) }

Obtain S^4 by merging and purging S^3 and S_1^3.
S^4 = MERGE_PURGE (S^3, S_1^3)
= { (0, 0), (12, 8), (14, 12), (16, 14), (22, 17), (26, 20), (28, 22) }

Pair (38, 31), (30, 26) ,and (42, 34) are discarded because its w > M

Find optimal solution
Here, n = 4.

Start with the last pair in S^4, i.e. (28, 22)
(28, 22) $\in S^4$ but (28, 22) $\notin S^3$
So set $x_n = x_4 = 1$
Update,

$p = p – p_4 = 28 – 16 = 12$
$w = w – w_4 = 22 – 14 = 8$
$n = n – 1 = 4 – 1 = 3$

Now n = 3, pair (12, 8) $\in S^3$ and (12, 8) $\in S^2$
So set $x_n = x_3 = 0$
$n = n – 1 = 3 – 1 = 2$

Now n = 2, pair(12, 8) $\in S^2$ but (12, 8) $\notin S^1$
So set $x_n = x_2 = 1$
Update,

$p = p - p_2 = 12 - 12 = 0$
$w = w - w_2 = 8 - 8 = 0$
Problem size is 0, so stop.

Optimal solution vector is (x_1, x_2, x_3, x_3) = (0, 1, 0, 1) Thus, this approach selects pair (12, 8) and (16, 14) which gives profit of 28.

0-1 Knapsack problem using Branch and Bound

Branch and Bound Method

In solving this problem, we shall use the Least Cost- Branch and Bound method, since this shall help us eliminate exploring certain branches of the tree. We shall also be using the fixed-size solution here. Another thing to be noted here is that this problem is a maximization problem, whereas the Branch and Bound method is for minimization problems. Hence, the values will be multiplied by -1 so that this problem gets converted into a minimization problem.

Now, consider the 0/1 knapsack problem with *n* objects and total weight W. We define the upper bound(U) to be the summation of $v_i x_i$ (where v_i denotes the value of that objects, and x_i is a binary value, which indicates whether the object is to be included or not), such that the total weights of the included objects is less than *W*. The initial value of U is calculated at the initial position, where objects are added in order until the initial position is filled.
We define the cost function to be the summation of $v_i f_i$, such that the total value is the maximum that can be obtained which is less than or equal to *W*. Here f_i indicates the fraction of the object that is to be included. Although we use fractions here, it is not included in the final solution.
Here, the procedure to solve the problem is as follows are:

- Calculate the cost function and the Upper bound for the two children of each node. Here, the (i + 1)th level indicates whether the ith object is to be included or not.
- If the cost function for a given node is greater than the upper bound, then the node need not be explored further. Hence, we can kill this node. Otherwise, calculate the upper bound for this node. If this value is less than *U*, then replace the value of *U* with this value. Then, kill all unexplored nodes which have cost function greater than this value.
- The next node to be checked after reaching all nodes in a particular level will be the one with the least cost function value among the unexplored nodes.

- While including an object, one needs to check whether the adding the object crossed the threshold. If it does, one has reached the terminal point in that branch, and all the succeeding objects will not be included.

In this manner, we shall find a value of U at the end which eliminates all other possibilites. The path to this node will determine the solution to this problem.

We are a given a set of *n* objects which have each have a value v_i and a weight w_i. The objective of the 0/1 Knapsack problem is to find a subset of objects such that the total value is maximized, and the sum of weights of the objects does not exceed a given threshold *W*. An important condition here is that one can either take the entire object or leave it. It is not possible to take a fraction of the object.
Consider an example where *n* = 4, and the values are given by {10, 12, 12, 18}and the weights given by {2, 4, 6, 9}. The maximum weight is given by *W = 15*. Here, the solution to the problem will be including the first, third and the fourth objects.

Approaches to solve this problem

The first idea that comes to mind as soon as we look at the problem would be to look at all possible combinations of objects, calculate their total weight, and if the total weight is less than the threshold, to calculate the total value (This approach is known as the *Brute Force*. Although this approach would give us the solution, it is of exponential time complexity. Hence, we look at the other possible methods.
We can use the *Dynamic Programming* approach to solve this problem as well. Although this method is far more efficient than the Brute Force method, it does not work in scenarios where the item weights are non-integer values.
Backtracking can also be used to solve this problem. However, this would mean exploring all possible branches until the solution is invalid, then going back a step and exploring other possibilities. As was in the case of the brute force method, this method also has exponential time complexity.
Since this is a combinatorial problem, once can use the *Branch and Bound* method to solve this problem. We shall explore this way of solving this problem in detail.

Time and Space Complexity

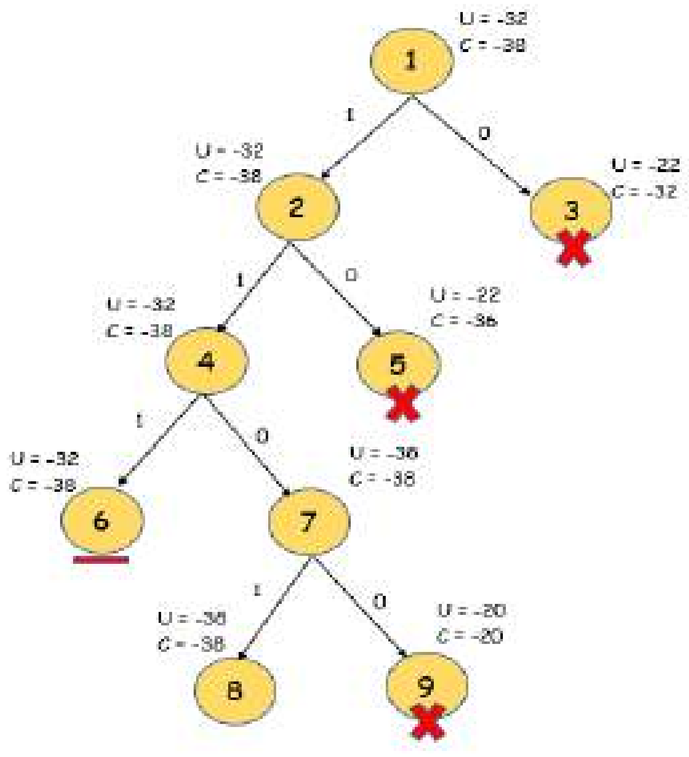
Even though this method is more efficient than the other solutions to this problem, its worst case time complexity is still given by $O(2^n)$, in cases where the entire tree has to be explored. However, in its best case, only one path through the tree will have to explored, and hence its best case time complexity is given by O(n). Since this method requires the creation of the state space tree, itsspace complexity will also be exponential.

Solving an Example

Consider the problem with n =4, V = {10, 10, 12, 18}, w = {2, 4, 6, 9} and W = 15. Here, we calculate the initital upper bound to be U = 10 + 10 + 12 = 32. Note that the 4th object cannot be included here, since that would exceed W. For the cost, we add 3/9 th of the final

value, and hence the cost function is 38. Remember to negate the values after calculation before comparison.
After calculating the cost at each node, kill nodes that do not need exploring. Hence, the final state space tree will be as follows (Here, the number of the node denotes the order in which the state space tree was explored):

Note here that node 3 and node 5 have been killed after updating U at node 7. Also, node 6 is not explored further, since adding any more weight exceeds the threshold. At the end, only nodes 6 and 8 remain. Since the value of U is less for node 8, we select this node. Hence the solution is {1, 1, 0, 1}, and we can see ere that the total weight is exactly equal to the threshold value in this case.



Conclusion

We can see that the branch and bound method will give the solution to the problem by exploring just 1 path in the best case. Although the worst case will be of exponential time complexity, this method will perform better than the other methods in most scenarios, since multiple branches get eliminated in each iteration.
With this article at OpenGenus, you must have the complete idea of solving 0-1 Knapsack problem using Branch and Bound.

Questions:

1. Explain Control abstraction of Dynamic Programming
2. Explain Control abstraction of Branch and Bound
3. Write an algorithm for 0/1 knapsack problem using dynamic programming
4. Write an algorithm for 0/1 knapsack problem using Branch and Bound
5. Generate the sets S^i, $0 \le i \le 3$ for following knapsack instance. N = 3, $(w_1, w_2, w_3) = (2, 3, 4)$ and $(p_1, p_2, p_3) = (1, 2, 5)$ with M = 6. Find optimal. Use dynamic programming

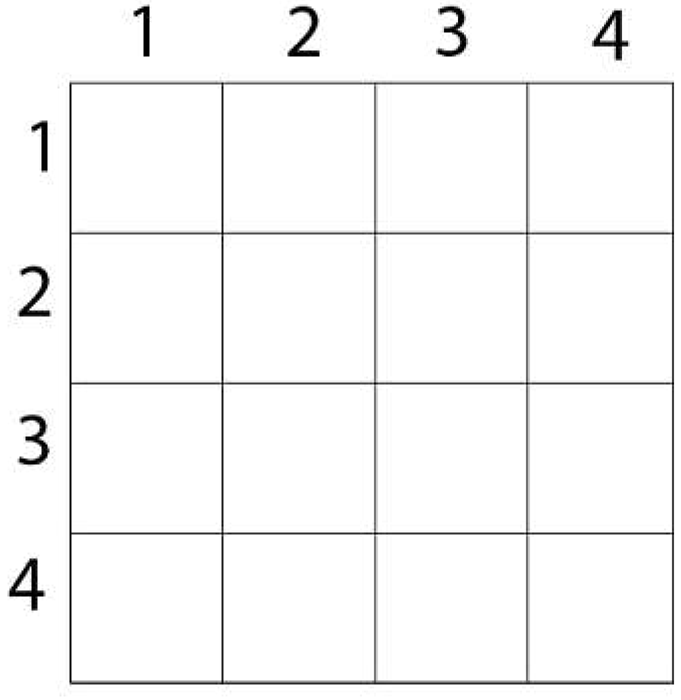
LP-III DAA (2023-24) Assignment 5 N-Queens Matrix Using Backtracking	
Students Name:	**Roll No :**
Batch :	**BE Computer Div -**

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

Theory : N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem.

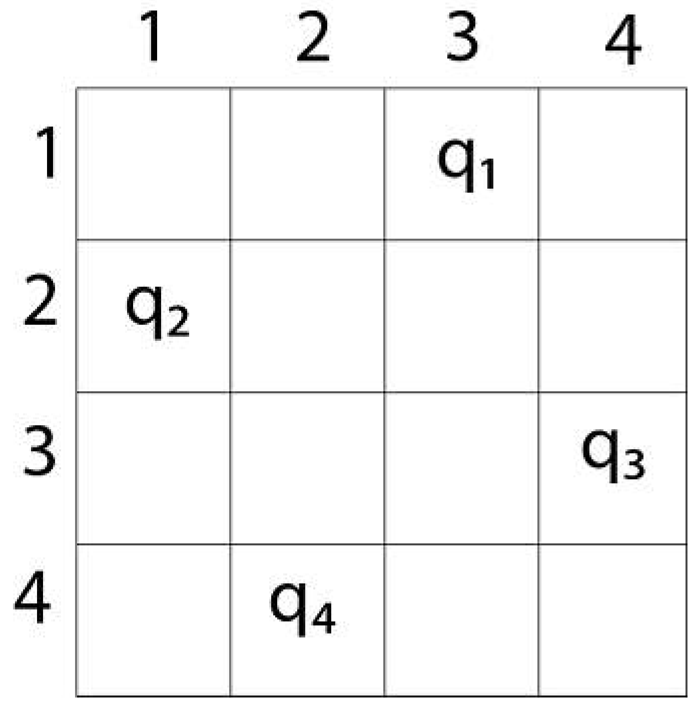
Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.



4x4 chessboard

Since, we have to place 4 queens such as q_1 q_2 q_3 and q_4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."

Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we put queen q_2 so that both these queens do not attack each other. We find that if we place q_2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q_2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q_3' safely. So we backtrack one step and place the queen 'q_2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q_3' which is (3, 2). But later this position also leads to a dead end, and no place is found where 'q_4' can be placed safely. Then we have to backtrack till 'q_1' and place it to (1, 2) and then all other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.



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

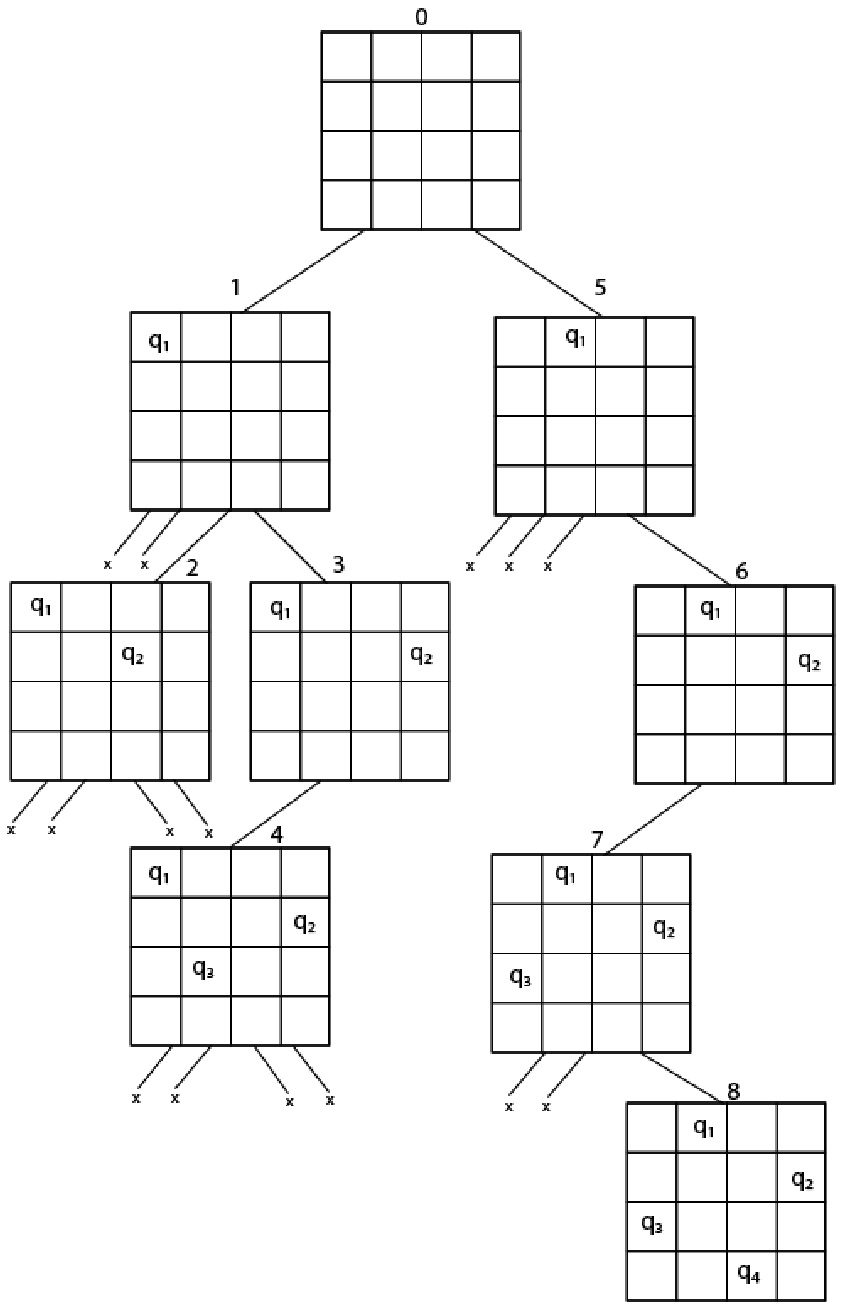
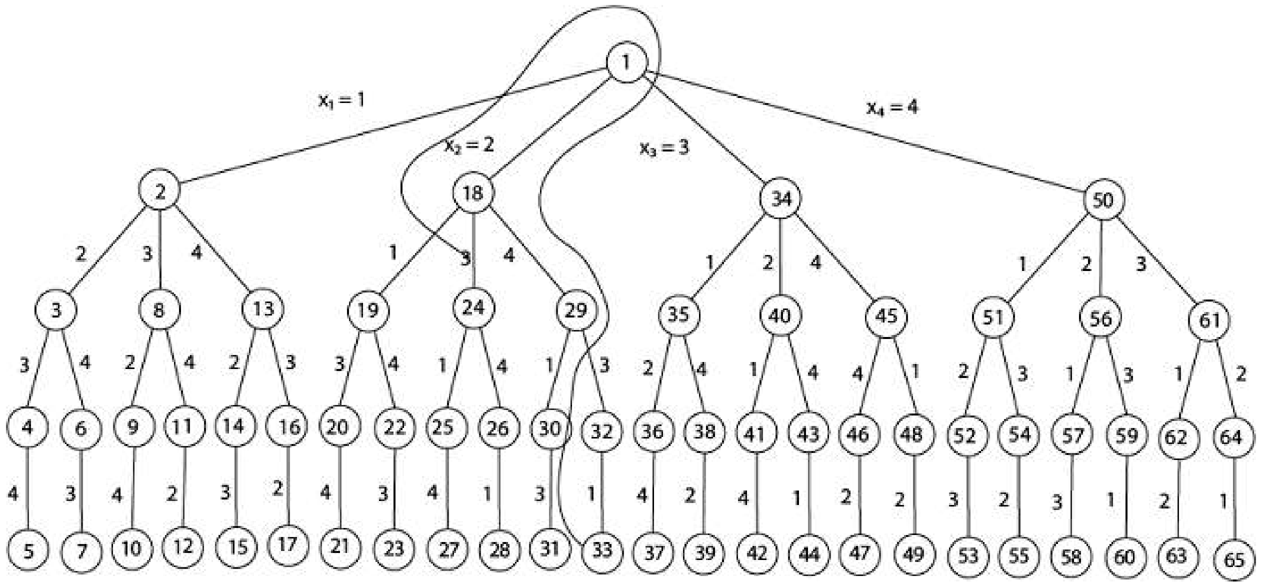
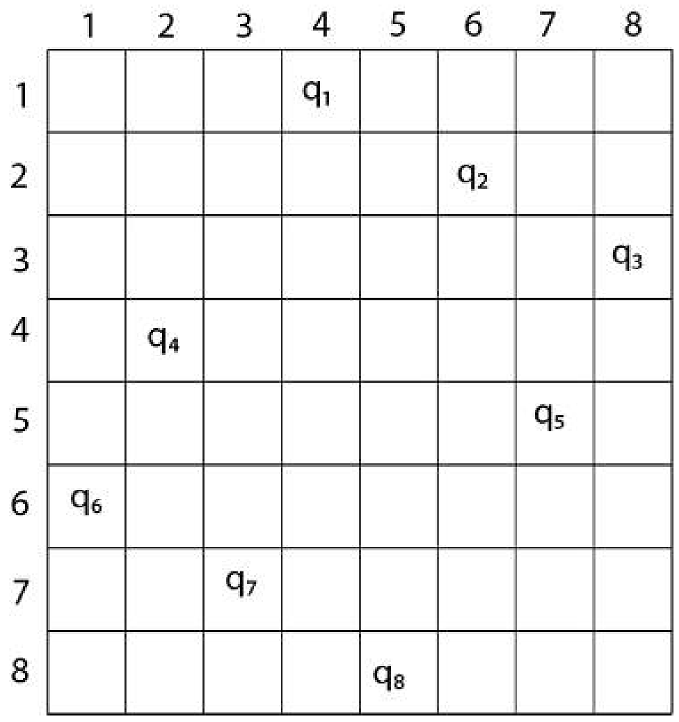


Fig shows the complete state space for 4 - queens problem. But we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.



It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen "q_i" is placed.

One possible solution for 8 queens problem is shown in fig:



1. Thus, the solution **for** 8 -queen problem **for** (4, 6, 8, 2, 7, 1, 3, 5).
2. If two queens are placed at position (i, j) and (k, l).
3. Then they are on same diagonal only **if** (i - j) = k - l or i + j = k + l.
4. The first equation implies that j - l = i - k.

5. The second equation implies that j - l = k - i.
6. Therefore, two queens lie on the duplicate diagonal **if** and only **if** |j-l|=|i-k|

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x_1, x_2,....x_{k-1} and whether there is no other queen on the same diagonal.

```
1. Place (k, i)
2. {
3.   For j ← 1 to k - 1
4.    do if (x [j] = i)
5.     or (Abs x [j]) - i) = (Abs (j - k))
6.   then return false;
7.   return true;
8. }
```

Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false. x [] is a global array whose final k - 1 values have been set. Abs (r) returns the absolute value of r.

```
1.  N - Queens (k, n)
2.  {
3.    For i ← 1 to n
4.        do if Place (k, i) then
5.    {
6.      x [k] ← i;
7.      if (k ==n) then
8.        write (x [1....n));
9.      else
10.     N - Queens (k + 1, n);
11.   }
12. }
```

Questions:

1. Discuss the complexity 0f 4 queen problem
2. Discuss the logic of attack of two queens
3. Write algorithm of n-queen problem
4. Can we apply Branch and Bound strategy to n-queen problem, discuss.