Marathwada Mitra Mandal's
College of Engineering, Pune
Karve Nagar, Pune-411 052

Department of Computer Engineering

Lab Manual

410246: Laboratory Practice III

Design and Analysis of Algorithm

BE COMPUTER

Prepared by:

Dr. Swati Shekapure
Ms. Reshma Kapadi

Pattern 2019
Academic Year 23-24

410246: Laboratory Practice III

Design and Analysis of Algorithm

A. Course Outcome

Course Outcome	**Statement**
	At the end of the course, student will be able to
410246.1	Analyze performance of an algorithm in terms of time and space
410246.2	Implement an algorithm that follows one of the following algorithm design strategies: divide and conquer, greedy, dynamic programming, backtracking, branch and bound.

B. CO-PO Mapping (Levels :1-Low , 2-Medium, 3-High)

Course Outcome	**Program outcomes**											
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
410246.1	3	2	2	-	1	-	-	1	2	-	2	2
410246.2	3	2	3	-	1	-	-	1	2	-	-	2

CO-PSO mapping

Course Outcome	**Program Specific Outcomes**		
	1	**2**	
410246.1	2	2	
410246.2	2	2	

INDEX

Sr. No.	Group	Title of Assignment	CO	PO	PSO
1	A	Write a program to calculate Fibonacci numbers and find its step count.	1	High: 1 Medium:2,3,9,11,12 Low:5,8	Medium: 1,2
2	A	Write a program to implement Huffman Encoding using a greedy strategy.	2	High: 1,3 Medium:2,9,12 Low:5,8	Medium: 1,2
3	A	Write a program to solve a fractional Knapsack problem using a greedy method.	2	High: 1,3 Medium:2,9,12 Low:5,8	Medium: 1,2
4	A	Write a program to solve a 0-1 Knapsack problem using dynamic programming	2	High: 1,3 Medium:2,9,12 Low:5,8	Medium: 1,2
5	A	Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen's matrix.	2	High: 1,3 Medium:2,9,12 Low:5,8	Medium: 1,2
6	A	Write a program for analysis of quick sort by using deterministic and randomized variants.	2	High: 1,3 Medium:2,9,12 Low:5,8	Medium: 1,2
7A	A	**Mini Project:** Write a program to implement matrix multiplication. Also implement multithreaded matrix multiplication with either one thread per row or one thread per cell. Analyze and compare their performance.	2	High: 1,3 Medium:2,9,12 Low:5,8	Medium: 1,2
7B	A	**Mini Project:**Implement merge sort and multithreaded merge sort. Compare time required by both the algorithms. Also analyze the	2	High: 1,3 Medium:2,9,12 Low:5,8	Medium: 1,2

		performance of each algorithm for the best case and the worst case			
7C	A	**Mini Project:**Implement the Naive string matching algorithm and Rabin-Karp algorithm for string matching. Observe difference in working of both the algorithms for the same input.	2	High: 1,3 Medium:2,9,12 Low:5,8	Medium: 1,2
7D	A	**Mini Project -** Different exact and approximation algorithms for Travelling-Sales-Person Problem	2	High: 1,3 Medium:2,9,12 Low:5,8	Medium: 1,2
8	CBS	Design 8-Queens matrix having first Queen placed. Use Branch and Bound to place remaining Queens to generate the final 8-queen's matrix.	2	High: 1,3 Medium:2,9,12 Low:5,8	Medium: 1,2
9	(Virtual Lab)	Implement Deterministic Quick sort	2	High: 1,3 Medium:2,9,12 Low:5,8	Medium: 1,2

'येथे बहुतांचे हित ।'

Marathwada Mitra Mandal's

COLLEGE OF ENGINEERING

Karvenagar, Pune

Permanently affiliated to SPPU | Accredited with 'A' Grade by NAAC

Recipient of 'Best College' award in 2018-19 by SPPU

Vision

To aspire for the Welfare of Society
through excellence in Science and Technology.

Mission

Our Mission is to

- **M**ould young talent for higher endeavours.
- **M**eet the challenges of globalization.
- **C**ommit for social progress with values and ethics.
- **O**rient faculty and students for research and development.
- **E**mphasize excellence in all disciplines.

Marathwada Mitra Mandal's
COLLEGE OF ENGINEERING
Department of Computer Engineering

Vision:
To contribute to welfare of society by empowering students with latest skills, tools and technologies in the field of Computer Engineering through excellence in education and research

Mission:
1. To provide an excellent academic environment for continuous improvement in the domain knowledge of Computer Engineering to solve real world problems.
2. To impart value-based education to students, with innovative and research skills to make them responsible engineering professionals for societal upliftment.
3. To strengthen links with industries through partnerships and collaborative developmental works.

Program Educational Objectives (PEOs)
1. To develop globally competent graduates with strong fundamental knowledge and analytical capability in latest technological trends.
2. To prepare graduates as ethical and committed professionals with a sense of societal and environmental responsibilities.
3. To inculcate research attitude in multidisciplinary domains with experiential learning and developing entrepreneurship skills.
4. To groom graduates by incorporating investigative approaches among them to effectively deal with global challenges.

Program Specific Outcomes (PSOs)
A graduate of the Computer Engineering Program will be able to
1. Analyze the problems and design solutions in the areas of Artificial Intelligence & High Performance Computing.
2. Develop advanced digital solutions using standard software engineering practices.

Program Outcomes (POs)
Engineering Graduates will be able to:
1. **Engineering Knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.
2. **Problem Analysis:** Identify, formulates, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design / development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. Conduct investigations of complex problems: Use research – based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the

information to provide valid conclusions.

5. Modern tool usage: Create, select and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs)

1. A graduate of the Computer Engineering Program will be able to Attain proficiency in analyzing and applying Computer Engineering Fundamentals viz. algorithms, system softwares and networking for contemporary system development through advanced technologies like - Cyber Security, High Performance Computing and Data Analytics (Artificial Intelligence,Machine Learning) etc.
2. Identify, formulate and solve real world problems in societal and environmental contexts through lifelong learning, entrepreneurship and leadership skills.

Assignment No. 1

TITLE: Write a program to calculate Fibonacci numbers and find its step count

OBJECTIVE: Learn how to analyze the performance of an algorithm

OUTCOME: Analyze performance of an algorithm.

THEORY:

Fibonacci Series generates subsequent numbers by adding two previous numbers. The Fibonacci series starts from two numbers − F0 & F1. The initial values of F0 & F1 can be taken 0, 1 or 1, 1 respectively.

Fibonacci series satisfies the following conditions −

Fn = Fn-1 + Fn-2

So a Fibonacci series can look like this −

F8 = 0 1 1 2 3 5 8 13

or, this −

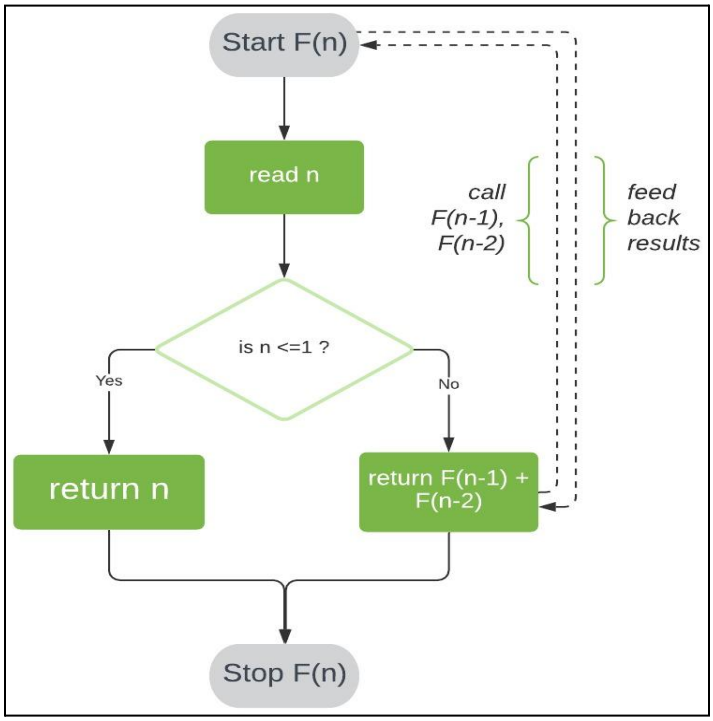
F8 = 1 1 2 3 5 8 13 21

If we denote the number at position n as Fn, we can formally define the Fibonacci Sequence as:
Fn = 0 for n = 0
Fn = 1 for n = 1
Fn = Fn-1 + Fn-2 for n > 1
Therefore, the start of the sequence is:
0, 1, 1, 2, 3, 5, 8, 13, …

Recursive Algorithm
Our first solution will implement recursion. This is probably the most intuitive approach, since the Fibonacci Sequence is, by definition, a recursive relation.

Method
Let's start by defining F(n) as the function that returns the value of Fn.
To evaluate F(n) for n > 1, we can reduce our problem into two smaller problems of the same kind: F(n-1) and F(n-2).
We can further reduce F(n-1) and F(n-2) to F((n-1)-1) and F((n-1)-2); and F((n-2)-1) and F((n-2)-2), respectively.
If we repeat this reduction, we'll eventually reach our known base cases and, thereby, obtain a solution to F(n).
Employing this logic, our algorithm for F(n) will have two steps:
Check if n ≤ 1. If so, return n.
Check if n > 1. If so, call our function F with inputs n-1 and n-2, and return the sum of the two results.

Here's a visual representation of this algorithm:



Pseudocode

Now that we understand how this algorithm works, let's implement some pseudocode:

Algorithm 1: F(n)

Input: Some non-negative integer n
Output: The nth number in the Fibonacci Sequence
if $n \leq 1$ **then**
  return n
else
  return $F(n-1) + F(n-2)$;

Analysis of Time Complexity

We can analyze the time complexity of F(n) by counting the number of times its most expensive operation will execute for n number of inputs. For this algorithm, the operation contributing the greatest runtime cost is addition.

Finding an Equation for Time Complexity

$F_0 = 0$

$F_n=1$

$F_n=F_{(n-1)}+ F_{(n-2)}$

Fig : Recursive calls during computation of Fibonacci number

Exercise 2: Solve the recurrence relation for *the Fibonacci sequence:*

The Fibonacci sequence is defined by the linear homogeneous, second-order recurrence relation $f_n - f_{n-1} - f_{n-2} = 0$ for $n \geq 3$

And the initial conditions $f_1 = 1$, $f_2 = 1$

Let $f_n = t^n$ for some t.

$t^n = t^{n-1} + t^{n-2}$

$t^n - t^{n-1} - t^{n-2} = 0$

$t^{n-2} (t^2 - t - 1) = 0.$

We must use the quadratic formula to solve: $t^2 - t - 1 = 0$

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{1 \pm \sqrt{5}}{2}$$

where a = 1, b = -1, c = -1

Let $S_n = \{ (1+\sqrt{5})/2 \}^n$ and $T_n = \{ (1-\sqrt{5})/2 \}^n$

Then $U_n = b S_n + d T_n$ *is a solution.*

$b S_0 + d T_0 = 0$ for $f_0 = 0$

$b S_1 + d T_1 = 1$ for $f_1 = 1$

For $f_0 = 0$ and $f_1 = 1$, We get $\mathbf{b = 1/\sqrt{5}}$

$\mathbf{d = -1/\sqrt{5}}$

So, $\mathbf{f_n = b S_n + d T_n}$

Iterative Algorithm

Let's move on to a much more efficient way of calculating the Fibonacci Sequence.

For this algorithm, we'll start at our known base cases and then evaluate each succeeding value until we finally reach the nth number. We'll store our sequence in an array F[].

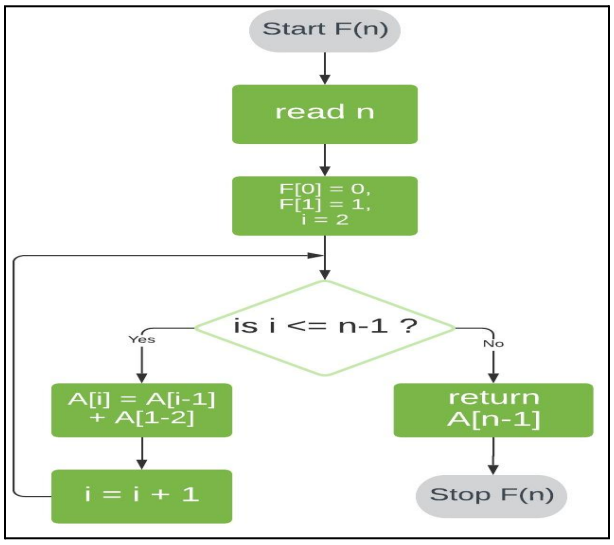
Method

First, we'll store 0 and 1 in F[0] and F[1], respectively.

Next, we'll iterate through array positions 2 to n-1. At each position i, we store the sum of the two preceding array values in F[i].

Finally, we return the value of F[n-1], giving us the number at position n in the sequence.

Here's a visual representation of this process:



Pseudocode

Let's take a look at the pseudocode for this approach:

Algorithm 2: $F(n)$

Input: Some non-negative integer n
Output: The nth number in the Fibonacci Sequence
$A[0] \leftarrow 0$;
$A[1] \leftarrow 1$;
for $i \leftarrow 2$ ***to*** $n-1$ **do**
  $A[i] \leftarrow A[i-1] + A[i-2]$;
return $A[n-1]$

Time Complexity

Analyzing the time complexity for our iterative algorithm is a lot more straightforward than its recursive counterpart.

In this case, our most costly operation is assignment. Firstly, our assignments of F[0] and F[1] cost O(1) each. Secondly, our loop performs one assignment per iteration and executes (n-1)-2 times, costing a total of O(n-3) = O(n).

Therefore, our iterative algorithm has a time complexity of O(n) + O(1) + O(1) = O(n).

This is a marked improvement from our recursive algorithm!

CONCLUSION: Hence, We have Successfully Implemented Fibonacci Algorithm recursively and Iteratively.

FAQS:

1. Why is the need for studying algorithms?

Answer: From a practical standpoint, a standard set of algorithms from different areas of computing must be known, in addition to being able to design them and analyze their efficiencies. From a theoretical standpoint the study of Algorithms is the cornerstone of computer science.

2. What are the characteristics of an algorithm?

Answer: Every algorithm should have the following five characteristics

(i) Input

(ii) Output

(iii) Definiteness

(iv) Effectiveness

(v) Termination

Therefore, an algorithm can be defined as a sequence of definite and effective instructions, which terminates with the production of correct output from the given input.In other words, viewed little more formally, an algorithm is a step by step formalization of a mapping function to map input set onto an output set.

3. What are the basic asymptotic efficiency classes?

The various basic efficiency classes are

- Constant:1
- Logarithmic: logn
- Linear:n
- N-log-n: n logn
- Quadratic:n2
- Cubic: n3
- Exponential:2^n
- Factorial:n!

4. List out algorithm design strategies and explain any one.

5. Analyze binary search algorithm

Assignment No. 2

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

OBJECTIVE: Learn how to implement algorithms that follow algorithm design strategies namely divide and conquer, greedy, dynamic programming, backtracking, branch and bound.

OUTCOME: Implement an algorithm that follows one of the following algorithm design strategies: divide and conquer, greedy, dynamic programming, backtracking, branch and bound.

THEORY:

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, meaning 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”.

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.

I. create a priority queue Q consisting of each unique character.
II. sort then in ascending order of their frequencies.

for all the unique characters:

create a newNode

extract minimum value from Q and assign it to leftChild of newNode

extract minimum value from Q and assign it to rightChild of newNode

calculate the sum of these two minimum values and assign it to the value of newNode

insert this newNode into the tree

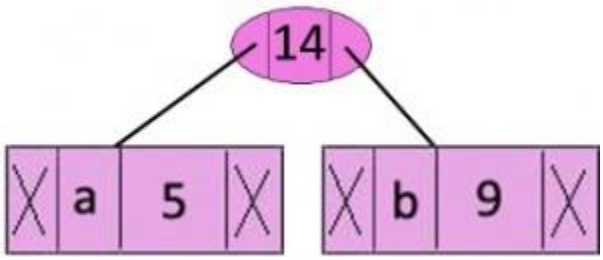
return rootNode

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 the root of a tree with a single node.

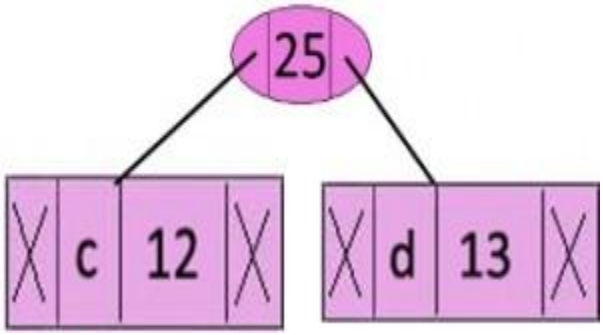
Step 2 Extract two minimum frequency nodes from the min heap. Add a new internal node with frequency 5 + 9 = 14.



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

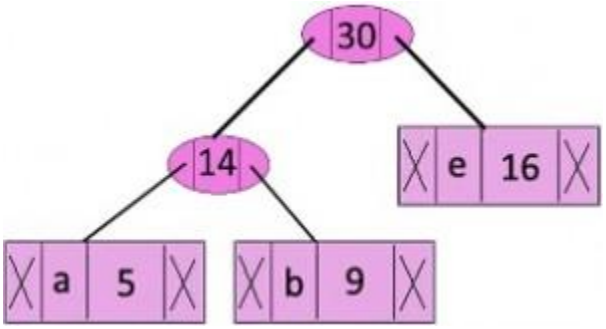
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



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

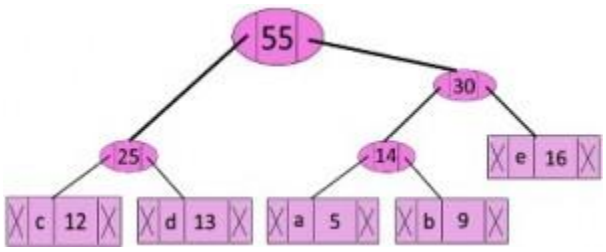
Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



Now the min heap contains 3 nodes.

Character	Frequency
Internal Node	25
Internal Node	30
f	45

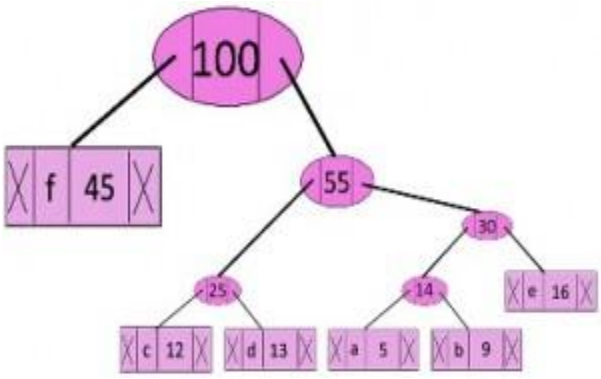
Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



Now the 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



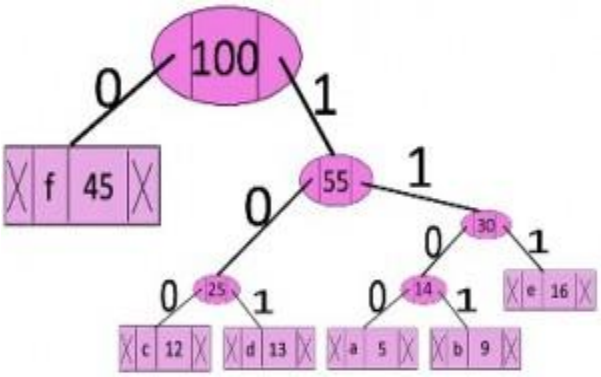
Now the min heap contains only one node.

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

Huffman Coding Complexity

The time complexity for encoding each unique character based on its frequency is O(nlog n).

Extracting minimum frequency from the priority queue takes place 2*(n-1) times and its complexity is O(log n). Thus the overall complexity is O(nlog n).

CONCLUSION: Hence , We have Successfully implemented Huffman Coding

FAQS:

1. **Explain the greedy method.**

2. Define feasible and optimal solutions.

3. Write the control abstraction for greedy methods.

```
Algorithm Greedy (a, n)
{
solution=0;
for i=1 to n do
{
x= select(a);
if feasible(solution ,x) then
solution=Union(solution ,x);
}
return solution;
}
```

4.What are the constraints of the knapsack problem?

To maximize $\Sigma p_i x_i$

$1 \le i \le n$

The constraint is : $\Sigma w_i x_i \ge m$ and $0 \le x_i \le 1$ $1 \le i \le n$

$1 \le i \le n$

where m is the bag capacity, n is the number of objects and for each object i w_i and p_i are the weight and profit of an object respectively.

5. Differentiate NP hard and NP complete problems

Assignment No. 3

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

OBJECTIVE: Learn how to implement algorithms that follow algorithm design strategies namely divide and conquer, greedy, dynamic programming, backtracking, branch and bound.

OUTCOME: Implement an algorithm that follows one of the following algorithm design strategies: divide and conquer, greedy, dynamic programming, backtracking, branch and bound.

THEORY:

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 a 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 to the Knapsack problem. Following is a set of examples.

- 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 the 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 >0
- Profit for i^{th} item pi >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.

$0 \leqslant xi \leqslant 1$

The i^{th} item contributes the weight $xi.wi$ to the total weight in the knapsack and profit $xi.pi$ to the total profit.

Hence, the objective of this algorithm is to

$$maximize \sum_{n=1}^{n} (x_i . pi)$$

subject to constraint,

$$\sum_{n=1}^{n} (x_i . wi) \leqslant W$$

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

$$\sum_{n=1}^{n} (x_i . wi) = W$$

In this context, first we need to sort those items according to the value of

$$\frac{p_i}{w_i} \text{ , so that } \frac{p_i+1}{w_i+1} \leq \frac{p_i}{w_i} .$$

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 pi/wi , then the while loop 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 (*pi /wi*)	7	10	6	5

As the provided items are not sorted based on pi/wi .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 (*pi/wi*)	10	7	6	5

Solution

After sorting all the items according to *pi/wi*. First all of *B* is chosen as the 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, a 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.

Conclusion:

Hence , we can conclude that we have successfully implemented fractional knapsack using greedy strategy.

FAQs:

1. **Solve Job sequencing with deadline using greedy algorithm**

Job	J_1	J_2	J_3	J_4	J_5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

2. **What are the components of a greedy algorithm?**
3. **State the best, average and worst case complexities of binary search for successful and unsuccessful search.**
4. **State the principle of optimality. Find two problems for which the principle does not hold.**
5. **Differentiate between Greedy strategy and Dynamic Programming**

Assignment No. 4

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

OBJECTIVE: Learn how to implement algorithms that follow algorithm design strategies namely divide and conquer, greedy, dynamic programming, backtracking, branch and bound.

OUTCOME: Implement an algorithm that follows one of the following algorithm design strategies: divide and conquer, greedy, dynamic programming, backtracking, branch and bound.

THEORY:

Dynamic Programming:
Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.
Dynamic programming is a step by step solution. At each step, the decision is taken to get the optimal solution.

Knapsack Problem:
Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).

Dynamic programming for 0/1 knapsack problem:

Knapsack Problem-

You are given the following-

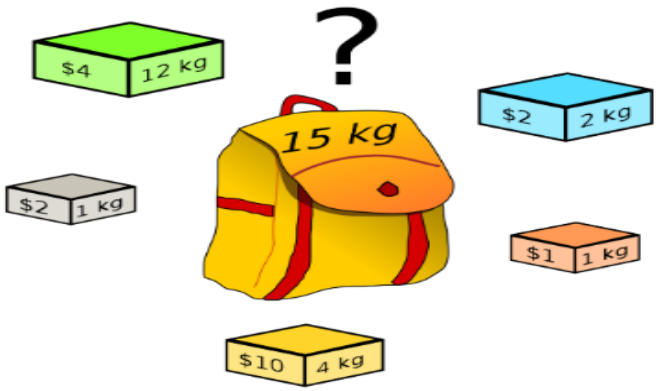
- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value.

The problem states-

Which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.

- And the weight limit of the knapsack does not exceed.



Knapsack Problem

Knapsack Problem Variants-

Knapsack problem has the following two variants-

1. Fractional Knapsack Problem
2. 0/1 Knapsack Problem

0/1 Knapsack Problem-

In 0/1 Knapsack Problem,

- As the name suggests, items are indivisible here.
- We can not take a fraction of any item.
- We have to either take an item completely or leave it completely.
- It is solved using a dynamic programming approach.

0/1 Knapsack Problem Using Dynamic Programming-

Consider-

- Knapsack weight capacity = w
- Number of items each having some weight and value = n

0/1 knapsack problem is solved using dynamic programming in the following steps-

Step-01:

- Draw a table say 'T' with (n+1) number of rows and (w+1) number of columns.
- Fill all the boxes of 0^{th} row and 0^{th} column with zeros as shown-

	0	1	2	3	.....	W
0	0	0	0	0	.....	0
1	0					
2	0					
	.....					
n	0					

T-Table

Step-02:

Start filling the table row wise top to bottom from left to right.

Use the following formula-

$$\mathbf{T\ (i\ ,\ j) = max\ \{\ T\ (\ i\text{-}1\ ,\ j\ )\ ,\ value_i + T(\ i\text{-}1\ ,\ j - weight_i\ )\ \}}$$

Here, T(i , j) = maximum value of the selected items if we can take items 1 to i and have weight restrictions of j.

- This step leads to completely filling the table.
- Then, value of the last box represents the maximum possible value that can be put into the knapsack.

Step-03:

To identify the items that must be put into the knapsack to obtain that maximum profit,

- Consider the last column of the table.
- Start scanning the entries from bottom to top.
- On encountering an entry whose value is not same as the value stored in the entry immediately above mark the row label of that entry.
-

- After all the entries are scanned, the marked labels represent the items that must be put into the knapsack.

Time Complexity-

- Each entry of the table requires constant time θ(1) for its computation.
- It takes θ(nw) time to fill (n+1)(w+1) table entries.
- It takes θ(n) time for tracing the solution since the tracing process traces the n rows.
- Thus, overall θ(nw) time is taken to solve the 0/1 knapsack problem using dynamic programming.

Problem-

Find the optimal solution for the 0/1 knapsack problem making use of a dynamic programming approach. Consider-

$$n = 4$$

$$w = 5 \text{ kg}$$

$$(w1, w2, w3, w4) = (2, 3, 4, 5)$$

$$(b1, b2, b3, b4) = (3, 4, 5, 6)$$

OR

A thief enters a house to rob it. He can carry a maximal weight of 5 kg into his bag. There are 4 items in the house with the following weights and values. What items should the thief take if he either takes the item completely or leaves it completely?

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

Solution-

Given-

- Knapsack capacity (w) = 5 kg
- Number of items (n) = 4

Step-01:

- Draw a table say 'T' with (n+1) = 4 + 1 = 5 number of rows and (w+1) = 5 + 1 = 6 number of columns.
- Fill all the boxes of 0^{th} row and 0^{th} column with 0.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

T-Table

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$\mathbf{T(i, j) = max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}}$$

Finding T(1,1)-

We have,

- i = 1
- j = 1
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,1) = max { T(1-1 , 1) , 3 + T(1-1 , 1-2) }

T(1,1) = max { T(0,1) , 3 + T(0,-1) }

T(1,1) = T(0,1) { Ignore T(0,-1) }

T(1,1) = 0

Finding T(1,2)-

We have,

- i = 1
- j = 2
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,2) = max { T(1-1 , 2) , 3 + T(1-1 , 2-2) }

T(1,2) = max { T(0,2) , 3 + T(0,0) }

T(1,2) = max {0 , 3+0}

T(1,2) = 3

Finding T(1,3)-

We have,

- i = 1
- j = 3
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,3) = max { T(1-1 , 3) , 3 + T(1-1 , 3-2) }

T(1,3) = max { T(0,3) , 3 + T(0,1) }

T(1,3) = max {0 , 3+0}

T(1,3) = 3

Finding T(1,4)-

We have,

- i = 1
- j = 4
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,4) = max { T(1-1 , 4) , 3 + T(1-1 , 4-2) }

T(1,4) = max { T(0,4) , 3 + T(0,2) }

T(1,4) = max {0 , 3+0}

T(1,4) = 3

Finding T(1,5)-

We have,

- i = 1
- j = 5
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,5) = max { T(1-1 , 5) , 3 + T(1-1 , 5-2) }

T(1,5) = max { T(0,5) , 3 + T(0,3) }

T(1,5) = max {0 , 3+0}

T(1,5) = 3

Finding T(2,1)-

We have,

- i = 2
- j = 1
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,1) = max { T(2-1 , 1) , 4 + T(2-1 , 1-3) }

T(2,1) = max { T(1,1) , 4 + T(1,-2) }

T(2,1) = T(1,1) { Ignore T(1,-2) }

T(2,1) = 0

Finding T(2,2)-

We have,

- i = 2
- j = 2
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,2) = max { T(2-1 , 2) , 4 + T(2-1 , 2-3) }

T(2,2) = max { T(1,2) , 4 + T(1,-1) }

T(2,2) = T(1,2) { Ignore T(1,-1) }

T(2,2) = 3

Finding T(2,3)-

We have,

- i = 2
- j = 3
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,3) = max { T(2-1 , 3) , 4 + T(2-1 , 3-3) }

T(2,3) = max { T(1,3) , 4 + T(1,0) }

T(2,3) = max { 3 , 4+0 }

T(2,3) = 4

Finding T(2,4)-

We have,

- i = 2
- j = 4
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,4) = max { T(2-1 , 4) , 4 + T(2-1 , 4-3) }

T(2,4) = max { T(1,4) , 4 + T(1,1) }

T(2,4) = max { 3 , 4+0 }

T(2,4) = 4

Finding T(2,5)-

We have,

- i = 2
- j = 5
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,5) = max { T(2-1 , 5) , 4 + T(2-1 , 5-3) }

T(2,5) = max { T(1,5) , 4 + T(1,2) }

T(2,5) = max { 3 , 4+3 }

T(2,5) = 7

Similarly, compute all the entries.

After all the entries are computed and filled in the table, we get the following table-

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

- The last entry represents the maximum possible value that can be put into the knapsack.
- So, maximum possible value that can be put into the knapsack = 7.

Identifying Items To Be Put Into Knapsack-

Following Step-04,

- We mark the rows labeled "1" and "2".
- Thus, items that must be put into the knapsack to obtain the maximum value 7 are- **Item-1 and Item-2**

Algorithm

Dynamic-0-1-knapsack (v, w, n, W)

for w = 0 to W do

```
c[0, w] = 0
for i = 1 to n do
        c[i, 0] = 0
for w = 1 to W do
        if wi ≤ w then
                if vi + c[i-1, w-wi] then
                        c[i, w] = vi + c[i-1, w-wi]
                else c[i, w] = c[i-1, w]
        else
        c[i, w] = c[i-1, w]
```

The set of items to take can be deduced from the table, starting at c[n, w] and tracing backwards where the optimal values came from.

If c[i, w] = c[i-1, w], then item i is not part of the solution, and we continue tracing with c[i-1, w]. Otherwise, item i is part of the solution, and we continue tracing with c[i-1, w-W].

Complexity Analysis:
This algorithm takes θ(n, w) times as table c has (n + 1).(w + 1) entries, where each entry requires θ(1) time to compute.

Conclusion: Two concepts are understood by executing this assignment

- The concept of dynamic programming
- Solving knapsack problem using dynamic programming

FAQS:

1. What is dynamic programming?
2. Write another name for a dynamic programming method.
3. What is the complexity of the 0/1 knapsack problem using dynamic programming?
4. Describe the Traveling salesperson problem and discuss how to solve it using dynamic programming.

Assignment No. 5

TITLE: Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final 8-queen's matrix.

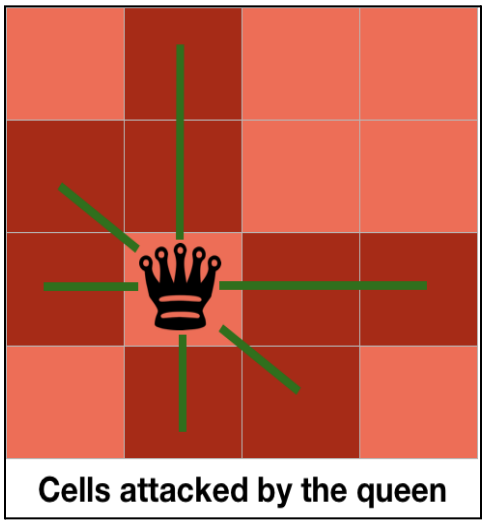
OBJECTIVE: Learn how to implement algorithms that follow algorithm design strategies namely divide and conquer, greedy, dynamic programming, backtracking, branch and bound.

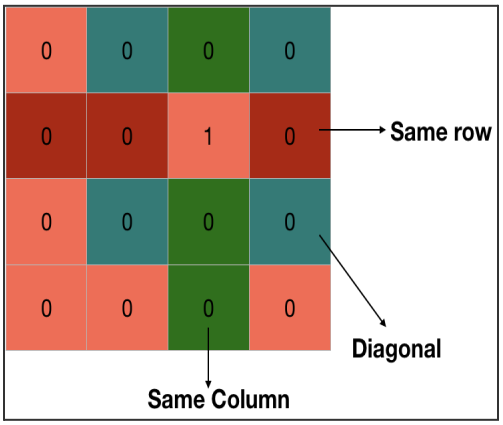
OUTCOME: Implement an algorithm that follows one of the following algorithm design strategies: divide and conquer, greedy, dynamic programming, backtracking, branch and bound.

THEORY:

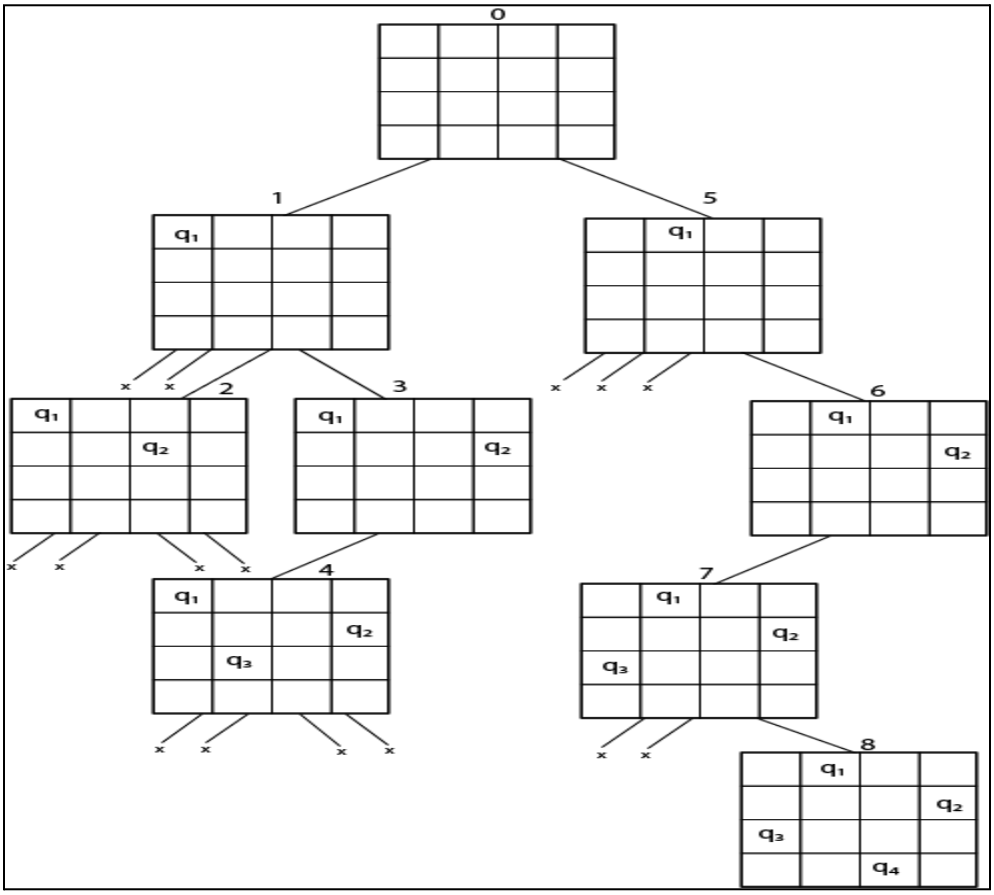
The N queens puzzle is the problem of placing N chess queens on an N×N chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

N queens problem is one of the most common examples of backtracking. Our goal is to arrange N queens on an NxN chessboard such that no queen can strike down any other queen. A queen can attack horizontally, vertically, or diagonally.





State Space for 4 Queens problem



Solution :

1. Q1-2,Q2- 4,Q3- 1,Q4- 3
2. Q1-3,Q2-1,Q3-4,Q4-2

Algorithm:

```
IS-ATTACK(i, j, board, N)
// checking in the column j
for k in 1 to i-1
        if board[k][j]==1
        return TRUE
 // checking upper right diagonal
k = i-1
l = j+1
while k>=1 and l<=N
        if board[k][l] == 1
        return TRUE
        k=k+1
         l=l+1
// checking upper left diagonal
k = i-1
l = j-1
```

```
while k>=1 and l>=1
        if board[k][l] == 1
        return TRUE
        k=k-1
        l=l-1
return FALSE
N-QUEEN(row, n, N, board)
 if n==0
          return TRUE
for j in 1 to N
        if !IS-ATTACK(row, j, board, N)
        board[row][j] = 1
        if N-QUEEN(row+1, n-1, N, board)
        return TRUE
        board[row][j] = 0 //backtracking, changing current decision return FALSE
```

	(i--,j--)(Upper left Diagonal)		
		(i,j) (Queen Position)	
	(i++,j--)(Bottom right diagonal)		

Time Complexity Analysis

1. the isPossible method takes O(n) time

2. for each invocation of loop in nQueenHelper, it runs for O(n) time

3. the isPossible condition is present in the loop and also calls nQueenHelper which is recursive adding this up, the recurrence relation is:

T(n) = O(n2) + n * T(n-1) solving the above recurrence by iteration or recursion tree, the time complexity of the nQueen problem is = O(N!)

CONCLUSION: Hence , We have Successfully implemented n queens problem using Backtracking and Branch and Bound Method

FAQS:

1. Present a model for the n-queen problem. List what variables are needed for the problem , their domains , and show how constraints can be presented.

Model 1

Variables: xi,j: 1, if there is a queen in (i,j)

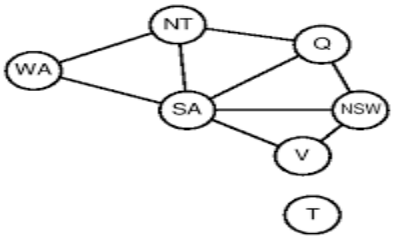
Domain: xi,j = {0, 1}

Constraint: list explicitly the invalid values for each pair of variables.

Model 2

Variables: x1, …, xn: position of queens on board
Domain: {1 … n2 }: tile index of each queen
Constraints: calculate the row (R) and column (C) index.
Given R1, R2 and C1, C2 of two queens' positions
R1 ≠ R2, C1 ≠ C2, and R1 – R2 ≠ C1 – C2, R1 – R2 ≠ C2 – C1
Model 3
Variables: x1, …, xn: rows of the chessboard
Domain: {1 … n}: column index of each queen placed
Constraints: for all i, j = {1, …, n}, xi ≠ xj, xi - xj ≠ i - j & xi - xj ≠ j – i)

2. Use an example to define the map coloring problem as a Constraint Satisfaction Problem.



An example is given here to present a possible answer. Marks are given for any answers addressing the question. For the map coloring problem on the left:
Variables: {WA, NT, Q, SA, NSW, VIC, T}
Domains: {red, green, blue}
Constraints: {not_same(WA,NT), not_same(WA, SA), not_same(NT,SA), …}

3. Explain why depth-first search is used in the search tree for Constraint Satisfaction Problems. Compare it with breadth-first search.

Answer: The solutions of CSP in a search tree will always be on the nth level.
If we use the breadth-first search, which searches all the nodes on the current level before it goes to the next level, the quickest it may find a solution will be after it finishes all the nodes on the 1st to the (n-1)th level.
Using depth first search, there is a chance that the search may find a solution the first time it goes down to the nth level. Although this does not often happen, the chance of finding a solution quicker is still much higher than that of breadth-first search.

4. Write control abstraction for backtracking and branch and bound search

Assignment No. 6

TITLE: Write a program for analysis of quick sort by using deterministic and randomized variants.
OBJECTIVE: Analyze performance of an algorithm.

OUTCOME: Analyze performance of an algorithm.

THEORY:
Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes n log n comparisons in the average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.
Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Algorithm

Pseudo Code for recursive QuickSort function:

```
/* low  –> Starting index,  high  –> Ending index */

quickSort(arr[], low, high) {

   if (low < high) {
```

```
        /* pi is partitioning index, arr[pi] is now at right place */
        pi = partition(arr, low, high);
        quickSort(arr, low, pi – 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

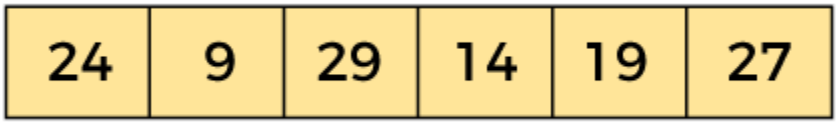
```
/* This function takes last element as pivot, places the pivot element at its correct
position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all
greater elements to right of pivot */
partition (arr[], low, high)
{
   // pivot (Element to be placed at right position)
pivot = arr[high];
 i = (low – 1)  // Index of smaller element and indicates the
// right position of pivot found so far
for (j = low; j <= high- 1; j++){
 // If current element is smaller than the pivot
if (arr[j] < pivot){
i++;    // increment index of smaller element
 swap arr[i] and arr[j]
    }
 }
   swap arr[i + 1] and arr[high])
return (i + 1)
}
```

Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

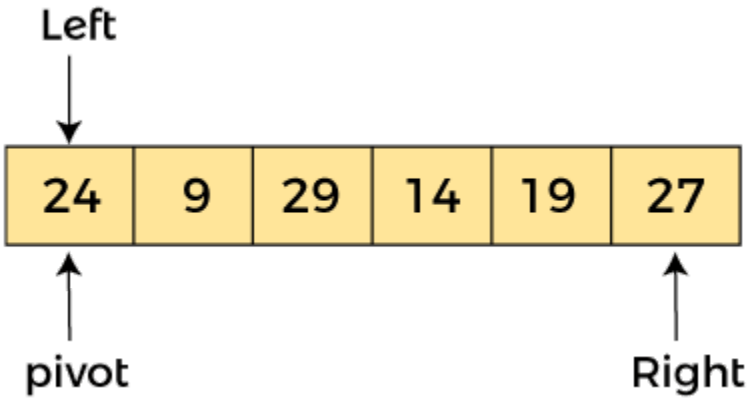
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

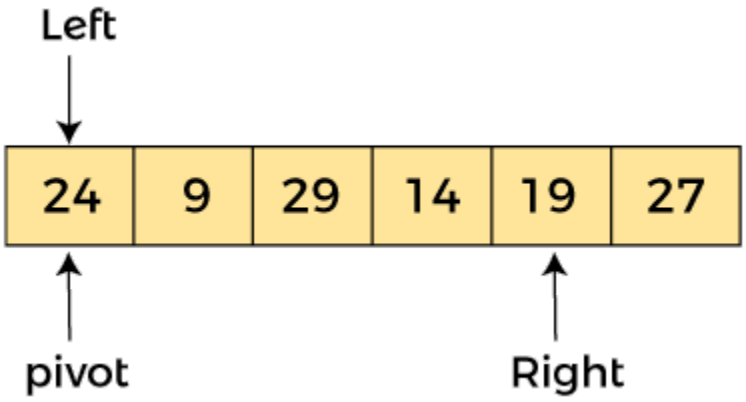


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.



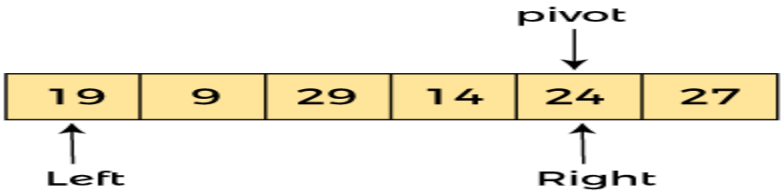
Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

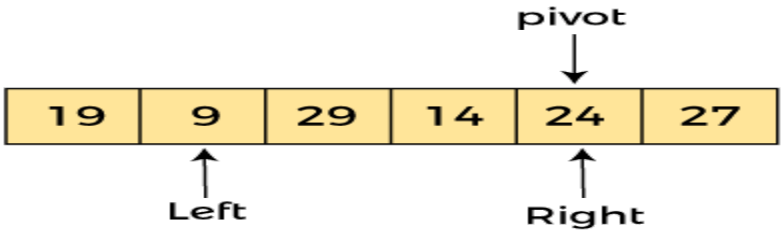
Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right,

as -

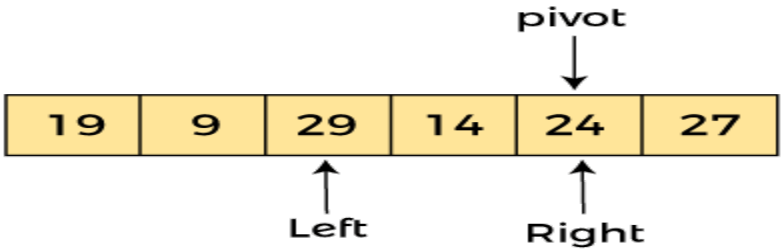


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

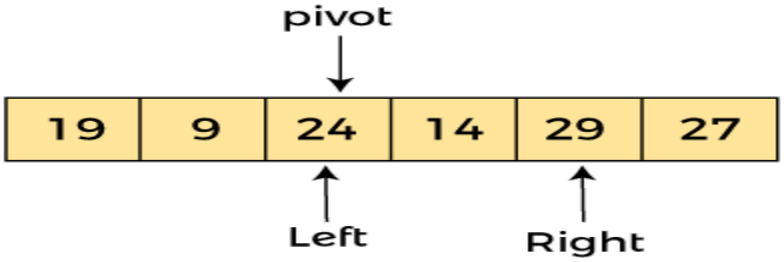
As a[pivot] > a[left], so algorithm moves one position to right as -



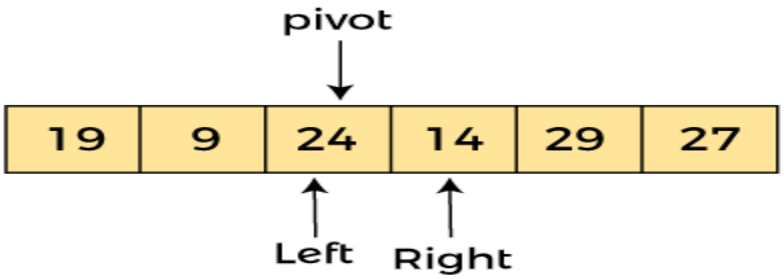
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



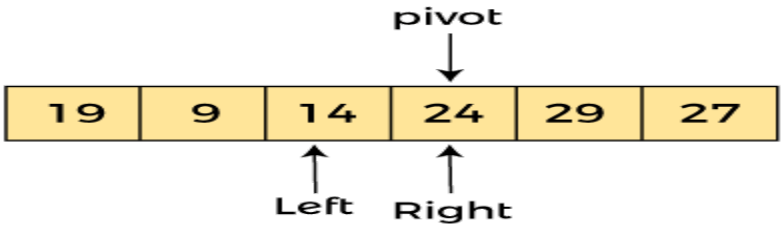
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



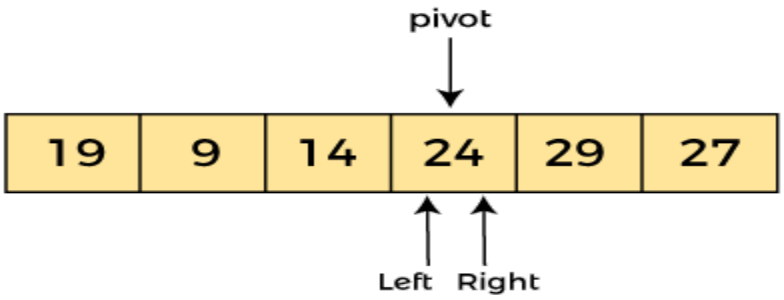
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.

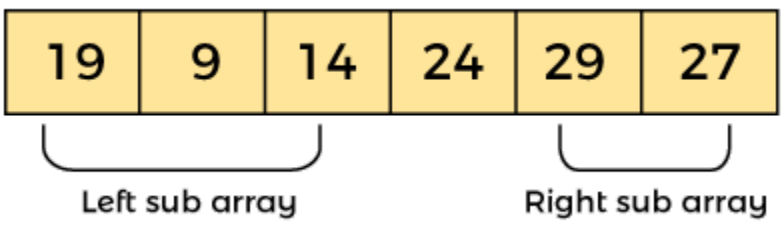


Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

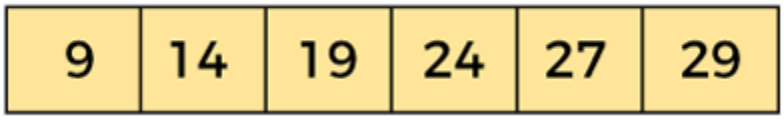
Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.

it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays.

After sorting gets done, the array will be -



Quicksort complexity

Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$

- Best Case Complexity - In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is O(n*logn).
- Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is O(n*logn).
- Worst Case Complexity - In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is $O(n^2)$.
- Space Complexity

Space Complexity	O(n*logn)
Stable	NO

Randomized Quick sort

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called a Randomized Algorithm. For example, in Randomized Quick Sort, we use a random number to pick the next pivot (or we randomly shuffle the array)

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

Conclusion:

Hence we have successfully implemented deterministic and randomized quick sort algorithms

FAQS:

1. **Write control abstraction for divide and conquer algorithm?**
2. **What is the time complexity of a quick sort algorithm in best case, average case and worst case?**
3. **What is a parallel algorithm? Can we implement parallel quicksort?**

Assignment No. 7A

TITLE: Write a program to implement matrix multiplication. Also implement multithreaded matrix multiplication with either one thread per row or one thread per cell. Analyze and compare their performance

OBJECTIVE: Analyze performance of an algorithm.

OUTCOME: Analyze performance of an algorithm.

THEORY:

In mathematics, particularly in linear algebra, matrix multiplication is a binary operation that produces a matrix from two matrices. For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The resulting matrix, known as the matrix product, has the number of rows of the first and the number of columns of the second matrix. The product of matrices A and B is denoted as AB.

Matrix multiplication was first described by the French mathematician Jacques Philippe Marie Binet in 1812,[2] to represent the composition of linear maps that are represented by matrices. Matrix multiplication is thus a basic tool of linear algebra, and as such has numerous applications in many areas of mathematics, as well as in applied mathematics, statistics, physics, economics, and engineering.[3][4] Computing matrix products is a central operation in all computational applications of linear algebra.

Pseudo code for matrix multiplication:

```
matrixMultiply(A, B):
Assume dimension of A is (m x n), dimension of B is (p x q)
Begin
  if n is not same as p, then exit
  otherwise define C matrix as (m x q)
  for i in range 0 to m - 1, do
    for j in range 0 to q – 1, do
      for k in range 0 to p, do
        C[i, j] = C[i, j] + (A[i, k] * A[k, j])
      done
    done
  done
End
```

Conclusion:

Hence , We have successfully implemented matrix multiplication using multithreading.

FAQ’s

1. Analyze the performance of multithreaded matrix multiplication
2. Explain Naive string matching algorithm
3. Explain airline Crew Scheduling problem

Assignment No. 7B

TITLE: Implement merge sort and multithreaded merge sort. Compare time required by both the algorithms. Also analyze the performance of each algorithm for the best case and the worst case
OBJECTIVE: Analyze performance of an algorithm.
OUTCOME: Analyze performance of an algorithm.

THEORY:

Merge sort is based on divide-and-conquer technique. Merge sort method is a two phase process-

1. Dividing
2. Merging

Dividing Phase: During the dividing phase, each time the given list of elements is divided into two parts. This division process continues until the list is small enough to divide.

Merging Phase: Merging is the process of combining two sorted lists, so that, the resultant list is also the sorted one. Suppose A is a sorted list with n element and B is a sorted list with n 2 elements. The operation that combines the elements of A and B into a single sorted list C with n=n 1 + n 2, elements is called merging.

Algorithm-(Divide algorithm)

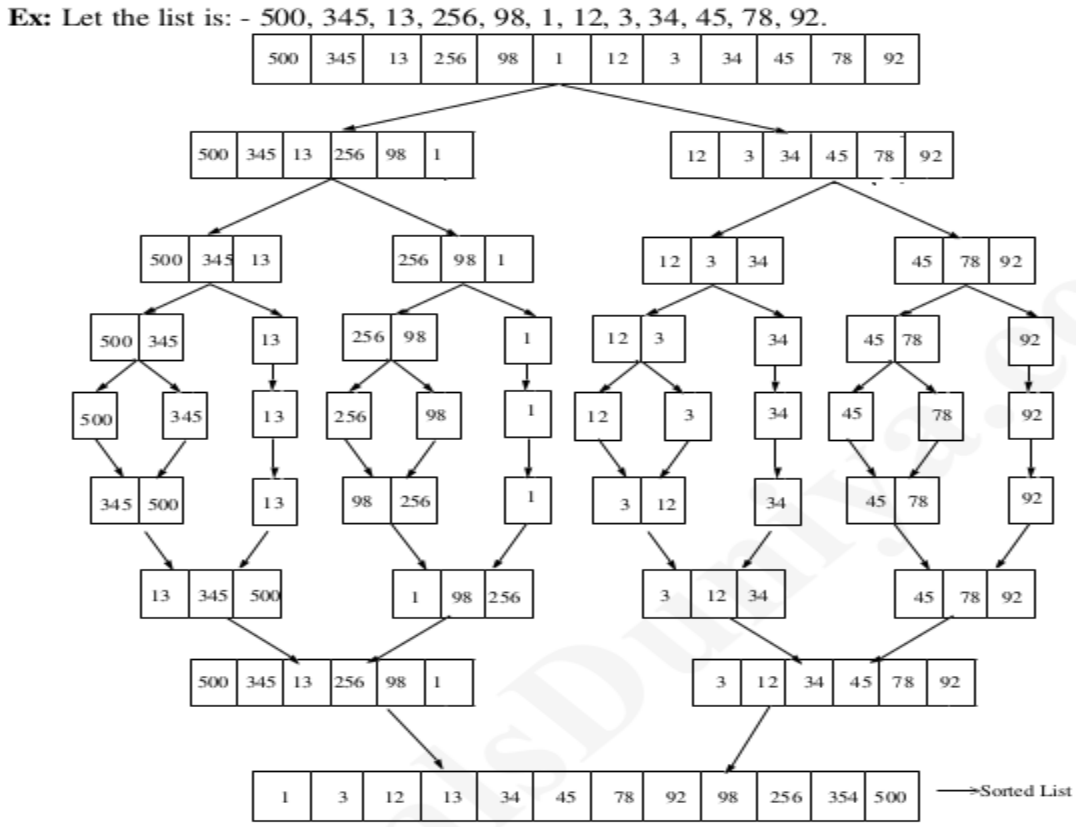
```
Algorithm Divide (a, low, high)
{
// a is an array, low is the starting index and high is the end index of a
If( low < high) then
{
Mid: = (low + high) /2;
Divide( a, low, mid);
Divide( a, mid +1, high);
Merge(a, low, mid, high);
}
}
```

The merging algorithm is as follows-

```
Algorithm Merge( a, low, mid, high)
{
L:= low;
H:= high;
J:= mid +1;
K:= low;
```

```
While (low  mid AND j  high) do
{
If (a[low < a[j]) then
{
B[k] = a[low];
K:= k+1;
Low:= low+1;
}
Else
{
B[k]= a[j];
K: = k+1;
J: = j+1;
}
}
While (low<=  mid) do
{
B[k]=a[low];
K: = k+1;
Low: =low + 1;
}
While (j<=  high) do
{
B[k]=a[j];
K: = k+1;
j: =j + 1;
}
//copy elements of b to a
For i: = 1 to n do
{
A[i]: =b[i];
}
}
```

Example:



The merge sort algorithm works as follows-

Step 1: If the length of the list is 0 or 1, then it is already sorted, otherwise,

Step 2: Divide the unsorted list into two sub-lists of about half the size.

Step 3: Again sub-divide the sub-list into two parts. This process continues until each element in the list becomes a single element.

Step 4: Apply merging to each sub-list and continue this process until we get one sorted list.

Efficiency of Merge List:

Let ‘n’ be the size of the given list/ then the running time for merge sort is given by the recurrence relation.

$$T(n) = \begin{cases} a & \text{if } n=1,\ a \text{ is a constant} \\ 2T(n/2) + Cn & \text{if } n>1,\ C \text{ is constant} \end{cases}$$

Assume that ‘n’ is a power of 2 i.e. $n=2^k$.

This can be rewritten as $k=\log_2 n$.

Let $T(n) = 2T(n/2) + Cn$ ——— (1)

We can solve this equation by using successive substitution.

Replace n by n/2 in equation, (1) ,we get

$$T(n/2) = 2T(n/4) + \frac{Cn}{2} \quad \text{——} (2)$$

$$\text{Thus, } T(n) = 2\left(2T(n/4) + \frac{Cn}{2}\right) + Cn$$

$$= 4T(n/4) + 2Cn$$

$$= 4T\left(2T(n/8) + \frac{Cn}{4}\right) + 2Cn$$

...

...

...

$$= 2^k T(1) + KCn \quad \left(\because k = \log_2 n\right)$$

$$= a\,n + Cn \log n$$

$$\therefore T(n) = O(n \log n)$$

Worst case	O(n log n)
Best case	O(n log n)
Average case	O(n log n)
Space Complexity	O($2n$)

CONCLUSION: Hence , We have Successfully implemented merge sort using multithreading.

FAQ's:

1. Write control abstraction for Divide and Conquer strategy
2. What are the basic asymptotic efficiency classes?
3. Compare the order of growth n(n-1)/2 and n^2.

Assignment No. 7C

TITLE: Implement the Naive string matching algorithm and Rabin-Karp algorithm for string matching. Observe difference in working of both the algorithms for the same input.

OBJECTIVE: Analyze performance of an algorithm.

OUTCOME: Analyze performance of an algorithm.

THEORY:

String Matching Algorithm is also called "String Searching Algorithm." This is a vital class of string algorithm that is declared as "this is the method to find a place where one of several strings are found within the larger string."

Given a text array, T [1.....n], of n character and a pattern array, P [1.....m], of m characters. The problems are to find an integer s, called a valid shift where $0 \leq s < n-m$ and T [s+1.....s+m] = P [1.....m]. In other words, to find even if P in T, i.e., where P is a substring of T. The item of P and T are characters drawn from some finite alphabet such as {0, 1} or {A, B .....Z, a, b..... z}.

Given a string T [1.....n], the substrings are represented as T [i.....j] for some $0 \leq i \leq j \leq n-1$, the string formed by the characters in T from index i to index j, inclusive. This process that a string is a substring of itself (take i = 0 and j =m).

The proper substring of string T [1.....n] is T [1.....j] for some $0 < i \leq j \leq n-1$. That is, we must have either $i > 0$ or $j < m-1$.

Using these descriptions, we can say given any string T [1.....n], the substrings are

T [i.....j] = T [i] T [i +1] T [i+2].....T [j] for some $0 \leq i \leq j \leq n-1$.

And proper substrings are

T [i.....j] = T [i] T [i +1] T [i+2].....T [j] for some $0 \leq i \leq j \leq n-1$.

Note: If i>j, then T [i.....j] is equal to the empty string or null, which has length zero.

Following are the string matching algorithms:

1. The Naive String Matching Algorithm
2. The Rabin-Karp-Algorithm
3. Finite Automata
4. The Knuth-Morris-Pratt Algorithm
5. The Boyer-Moore Algorithm

1. Naïve string matching algorithm:

The naïve approach tests all the possible placement of Pattern P [1.....m] relative to text T [1.....n]. We try shift s = 0, 1.....n-m, successively and for each shift s. Compare T [s+1.....s+m] to P [1.....m].

The naïve algorithm finds all valid shifts using a loop that checks the condition P [1.....m] = T [s+1.....s+m] for each of the n - m +1 possible value of s.

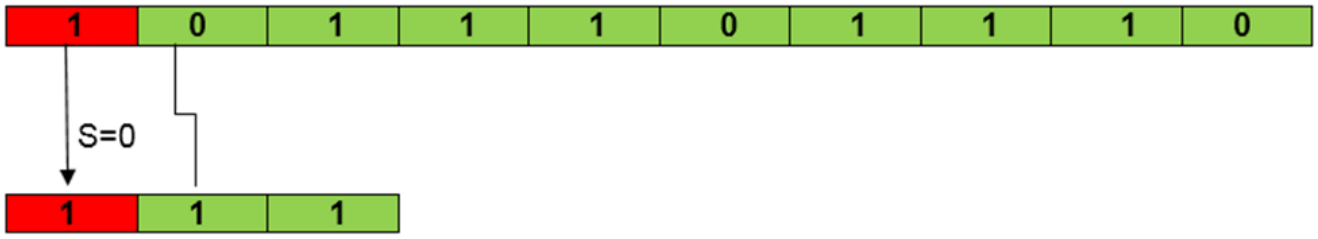
- **Pseudocode for Naïve string-matching algorithm:**

NAIVE-STRING-MATCHER (T, P)

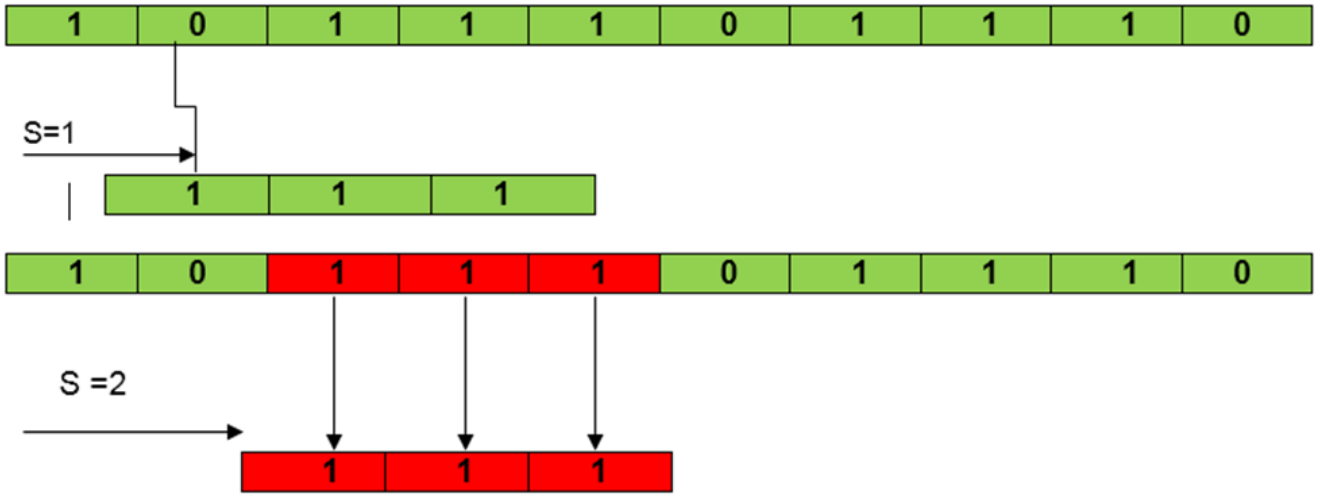
1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n -m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s

- **Analysis:**
  This for loop from 3 to 5 executes for n-m + 1(we need at least m characters at the end) times and in iteration we are doing m comparisons. So the total complexity is O (n-m+1).
- Suppose T = 1011101110
- P = 111
- Find all the Valid Shift

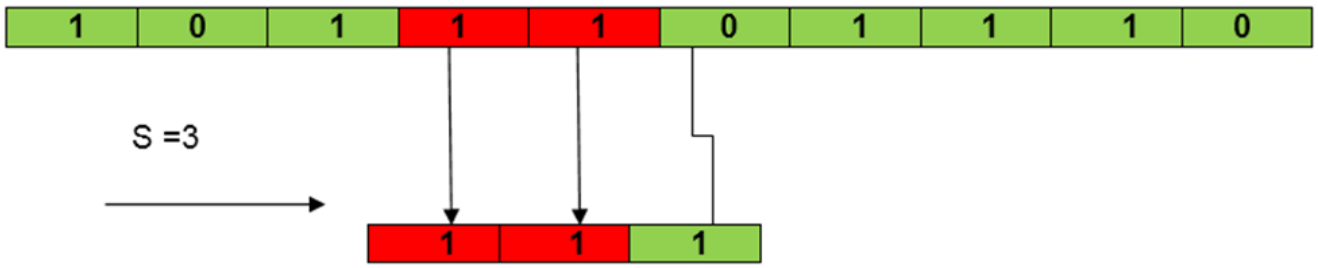
T = Text

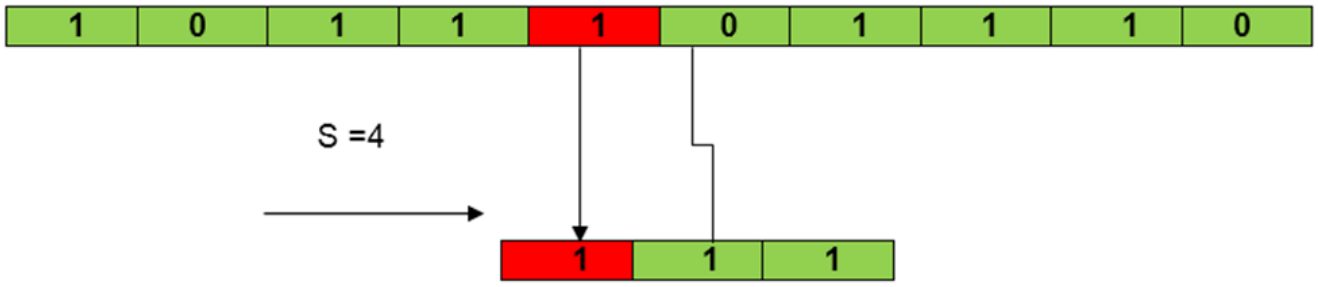


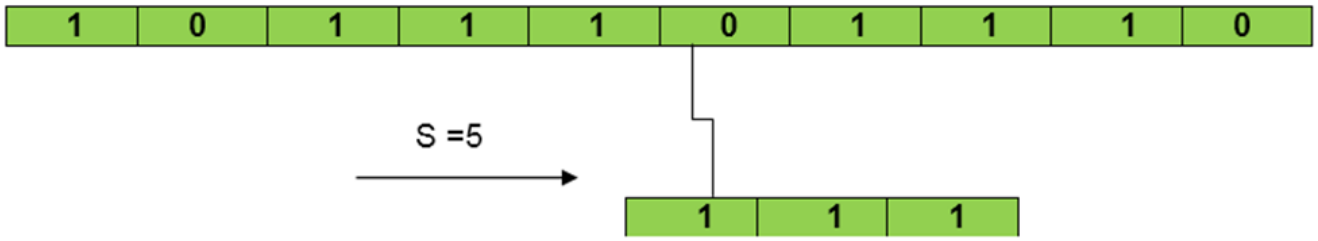
P = Pattern

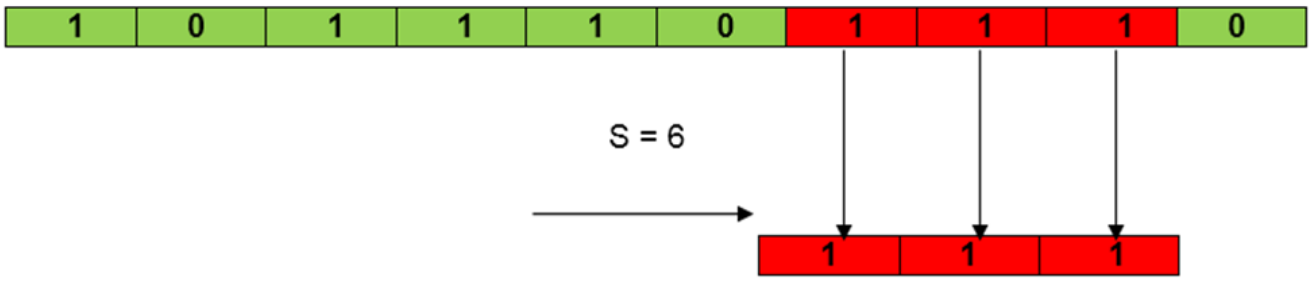


So, S=2 is a Valid Shift

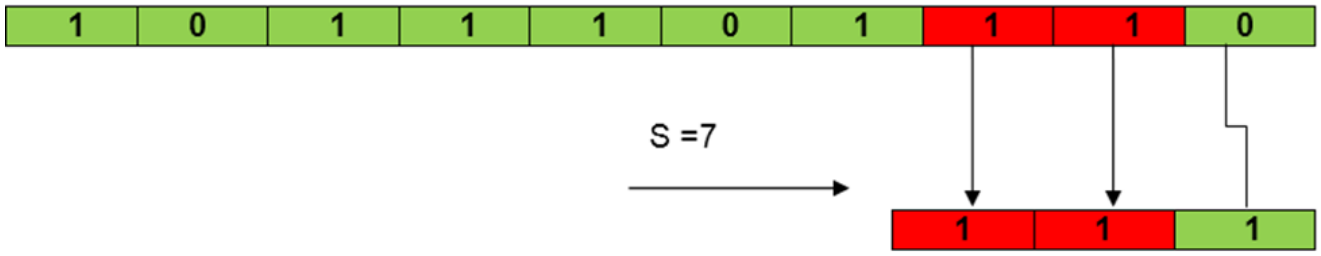








So, S=6 is a Valid Shift



2. Rabin-Karp-Algorithm:

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for the next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

- **Pseudocode for Rabin-Karp algorithm:**

RABIN-KARP-MATCHER (T, P, d, q)

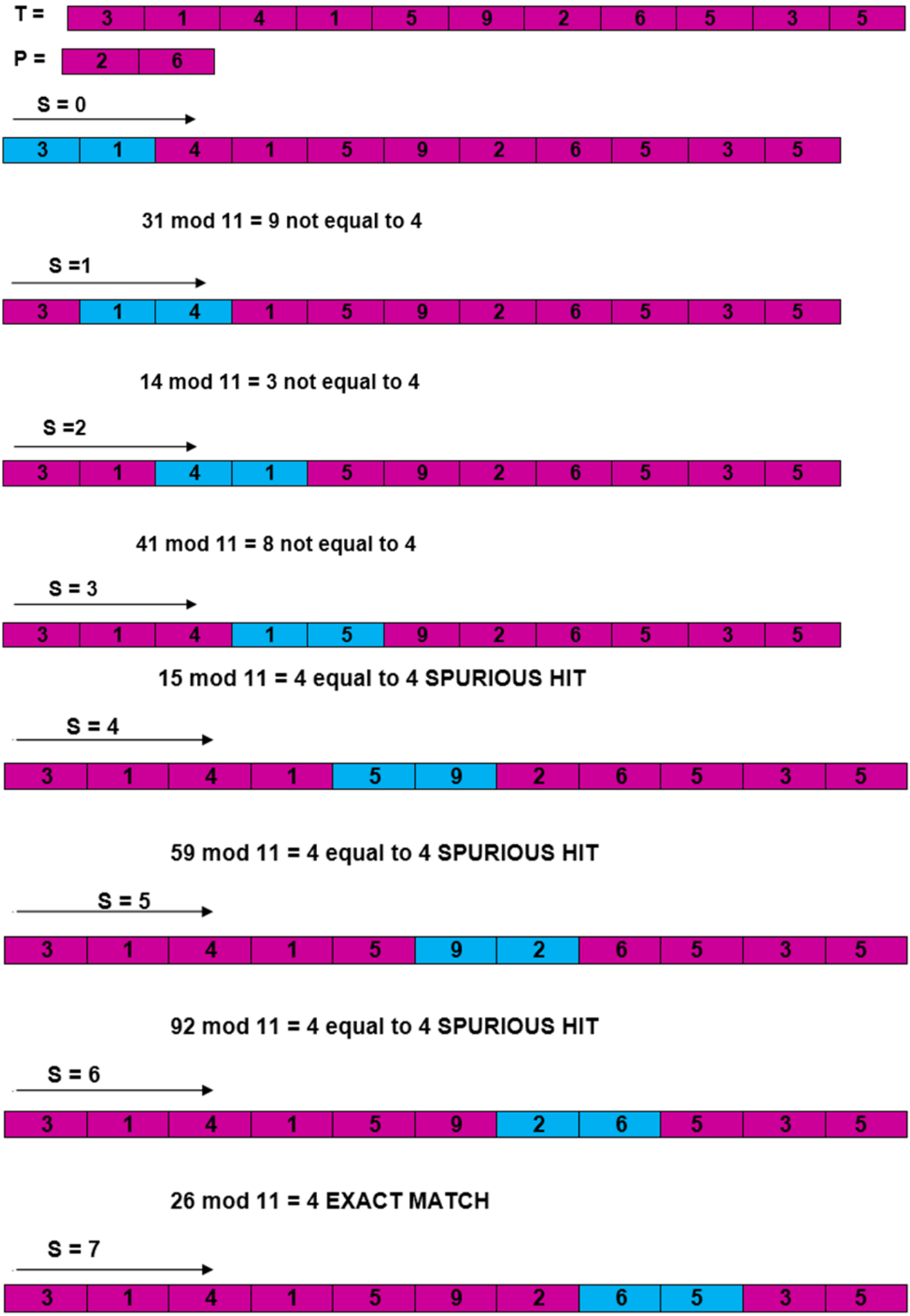
1. n ← length [T]
2. m ← length [P]
3. h ← dm-1 mod q
4. p ← 0
5. t0 ← 0
6. for i ← 1 to m
7. do p ← (dp + P[i]) mod q
8. t0 ← (dt0+T [i]) mod q
9. for s ← 0 to n-m
10. do if p = ts
11. then if P [1.....m] = T [s+1.....s + m]
12. then "Pattern occurs with shift" s
13. If s < n-m
14. then ts+1 ← (d (ts-T [s+1]h)+T [s+m+1])mod q

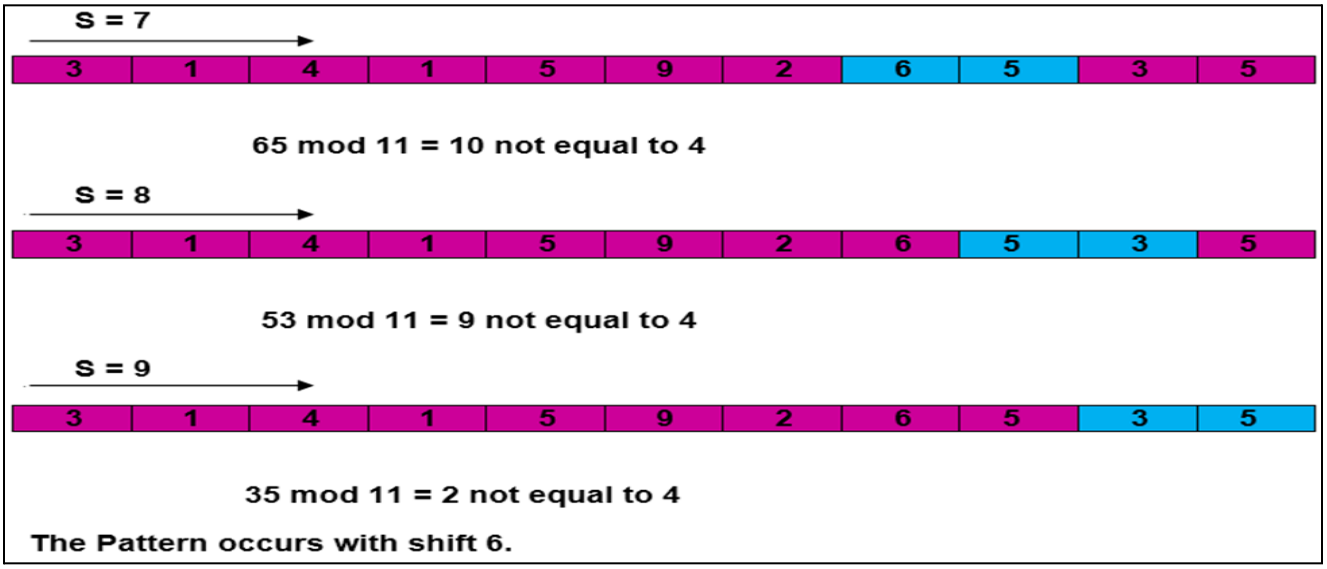
- **Example:**

For string matching, working module q = 11, how many spurious hits does the Rabin-Karp matcher encounter in Text T = 31415926535.....

1. T = 31415926535.....
2. P = 26
3. Here T.Length =11 so Q = 11
4. And P mod Q = 26 mod 11 = 4
5. Now find the exact match of P mod Q...

Solution:


T = 3 1 4 1 5 9 2 6 5 3 5
P = 2 6
S = 0
3 1 4 1 5 9 2 6 5 3 5
31 mod 11 = 9 not equal to 4
S =1
3 1 4 1 5 9 2 6 5 3 5
14 mod 11 = 3 not equal to 4
S =2
3 1 4 1 5 9 2 6 5 3 5
41 mod 11 = 8 not equal to 4
S = 3
3 1 4 1 5 9 2 6 5 3 5
15 mod 11 = 4 equal to 4 SPURIOUS HIT
S = 4
3 1 4 1 5 9 2 6 5 3 5
59 mod 11 = 4 equal to 4 SPURIOUS HIT
S = 5
3 1 4 1 5 9 2 6 5 3 5
92 mod 11 = 4 equal to 4 SPURIOUS HIT
S = 6
3 1 4 1 5 9 2 6 5 3 5
26 mod 11 = 4 EXACT MATCH
S = 7
3 1 4 1 5 9 2 6 5 3 5



Complexity:

The running time of RABIN-KARP-MATCHER in the worst-case scenario O ((n-m+1) m but it has a good average case running time. If the expected number of strong shifts is small O (1) and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time O (n+m) plus the time required to process spurious hits.

Conclusion:Hence , We have successfully implemented Robin Karp algorithm

FAQ:

1.What are the applications of the Robin Karp algorithm?

2.What is the difference between breadth first search and distributed breadth first search?

3.Explain the distributed minimum spanning tree .

Assignment No. 7D

TITLE: Different exact and approximation algorithms for Travelling-Sales-Person Problem

OBJECTIVE: Analyze performance of an algorithm.

OUTCOME: Analyze performance of an algorithm.

Theory:

The Traveling Salesman Problem is based on a real life scenario, where a salesman from a company has to start from his own city and visit all the assigned cities exactly once and return to his home till the end of the day. The exact problem statement goes like this,

"Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point."

There are two important things to be cleared about in this problem statement,

- Visit every city exactly once
- Cover the shortest path

The approximate algorithm for TSP works only if the problem instance satisfies Triangle-Inequality.

"The least distant path to reach a vertex j from i is always to reach j directly from i, rather than through some other vertex k (or vertices)" i.e.,

dis(i, j) <= dis(i, k) + dist(k, j)

dis(a,b) = distance between a & b, i.e. the edge weight.

The simplest approximation algorithms for the traveling salesman problem are based on the greedy technique. We will discuss two such algorithms.

Nearest-neighbor algorithm

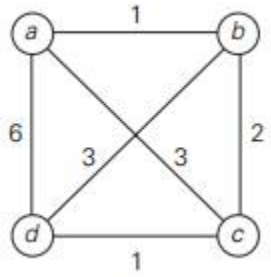
The following well-known greedy algorithm is based on the nearest-neighbor heuristic: always go next to the nearest unvisited city.

Step 1 Choose an arbitrary city as the start.

Step 2 Repeat the following operation until all the cities have been visited: go to the unvisited city nearest the one visited last (ties can be broken arbitrarily).

Step 3 Return to the starting city.

Example: For the instant represented by the graph in Figure 12.10, with a as the starting vertex, the nearest-neighbor algorithm yields the tour (Hamiltonian circuit) sa: a − b − c − d − a of length 10.



The optimal solution, as can be easily checked by exhaustive search, is the tour s*: a − b − d − c − a of length 8. Thus, the accuracy ratio of this approximation is

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{10}{8} = 1.25$$

(i.e., tour s_a is 25% longer than the optimal tour s^*).

indeed, if we change the weight of edge (a, d) from 6 to an arbitrary large number w ≥ 6 in Example 1, the algorithm will still yield the tour a − b − c − d − a of length 4 + w, and the optimal solution will still be a − b − d − c − a of length 8. The major drawback of this method is that there may exist some long path which has to be traversed to return to the starting city.

There are approximation algori-thms for the traveling salesman problem that exploit a connection between Hamil-tonian circuits and spanning trees of the same graph. Since removing an edge from a Hamiltonian circuit yields a spanning tree, we can expect that the structure of a minimum spanning tree provides a good basis for constructing a shortest tour approximation. Here is an algorithm that implements this idea in a rather straight-forward fashion.

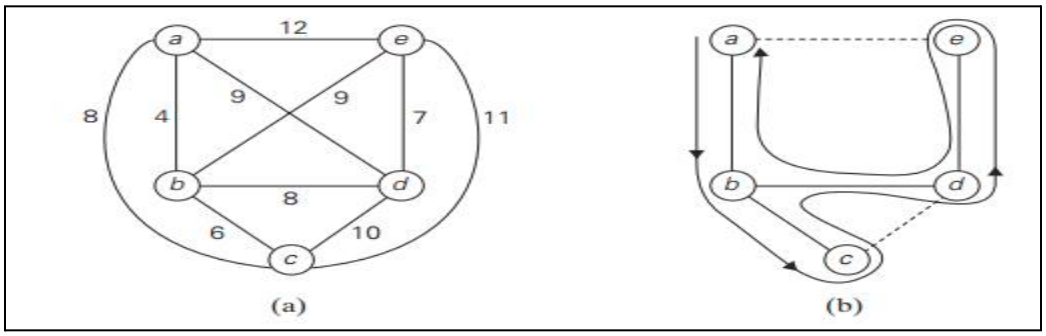
Twice-around-the-tree algorithm

Step 1 Construct a minimum spanning tree of the graph corresponding to a given instance of the traveling salesman problem.

Step 2 Starting at an arbitrary vertex, perform a walk around the minimum spanning tree recording all the vertices passed by. (This can be done by a DFS traversal.)

Step 3 Scan the vertex list obtained in Step 2 and eliminate from it all repeated occurrences of the same vertex except the starting one at the end of the list. (This step is equivalent to making shortcuts on the walk.) The vertices remaining on the list will form a Hamiltonian circuit, which is the output of the algorithm.

Let us apply this algorithm to the graph in Figure 12.11a. The minimum spanning tree of this graph is made up of edges (a, b), (b, c), (b, d), and (d, e) (Figure 12.11b). A twice-around-the-tree walk that starts and ends at a is



Graph b.Walk around the minimum spanning tree with shortcuts

a, b, c, b, d, e, d, b, a.

Eliminating the second b (a shortcut from c to d), the second d, and the third b (a shortcut from e to a) yields the Hamiltonian circuit a, b, c, d, e, a of length 39.

Conclusion : Hence we have successfully implemented traveling salesperson problem

FAQs:

1. Explain the concept of randomized algorithm in brief with example
2. Explain the concept of approximation algorithm in brief with example
3. What is an embedded system? Explain embedded sorting algorithm

Content Beyond Syllabus

TITLE: Implement n queen algorithm using branch and bound method

OBJECTIVE: Analyze performance of an algorithm.

OUTCOME: Analyze performance of an algorithm.

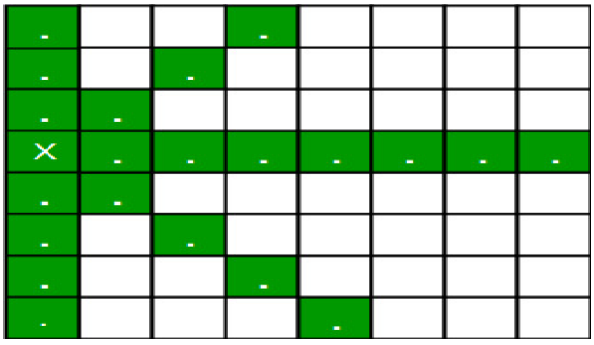
THEORY:

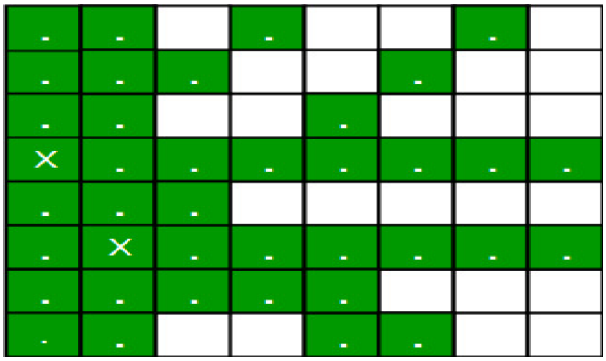
N Queen Problem using Branch And Bound

The N queens puzzle is the problem of placing N chess queens on an N×N chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

In the backtracking solution we backtrack when we hit a dead end. In the Branch and Bound solution, after building a partial solution, we figure out that there is no point going any deeper as we are going to hit a dead end.

Let's begin by describing a backtracking solution. "The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false."





- For the 1st Queen, there are a total 8 possibilities as we can place the 1st Queen in any row of the first column. Let's place Queen 1 on row 3.

- After placing 1st Queen, there are 7 possibilities left for the 2nd Queen. But wait, we don't really have 7 possibilities. We cannot place Queen 2 on rows 2, 3 or 4 as those cells are under attack from Queen 1. So, Queen 2 has only 8 – 3 = 5 valid positions left.

- After picking a position for Queen 2, Queen 3 has even fewer options as Most of the cells in its column are under attack from the first 2 Queens.

We need to figure out an efficient way of keeping track of which cells are under attack.

Basically, we have to ensure 4 things:

1. No two queens share a column.

2. No two queens share a row.

3. No two queens share a top-right to left-bottom diagonal.

4. No two queens share a top-left to bottom-right diagonal.

Number 1 is automatic because of the way we store the solution.
For number 2, 3 and 4, we can perform updates in O(1) time.

The idea is to keep three Boolean arrays that tell us which rows and which diagonals are occupied.

Lets do some pre-processing first. Let's create two N x N matrix one for / diagonal and other one for \ diagonal. Let's call them slashCode and backslashCode respectively.

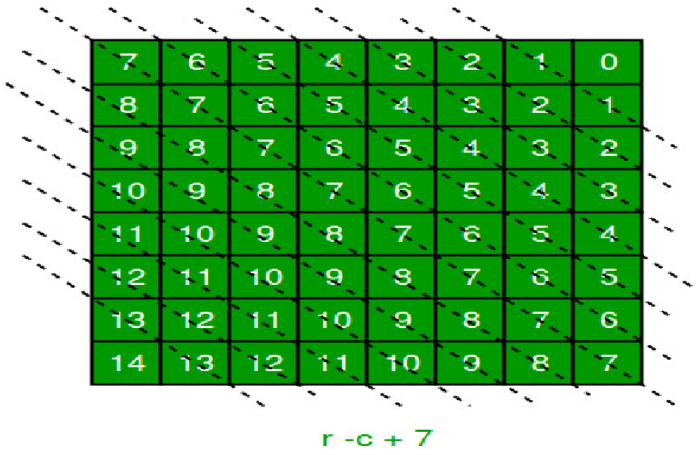
The trick is to fill them in such a way that two queens sharing a same / diagonal will have the same value in matrix slashCode, and if they share same \ diagonal, they will have the same value in backslashCode matrix.

For an N x N matrix, fill slashCode and backslashCode matrix using below formula –

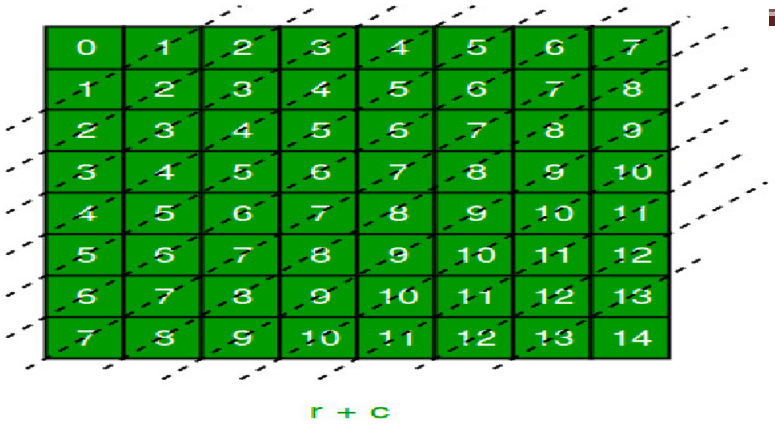
slashCode[row][col] = row + col

backslashCode[row][col] = row – col + (N-1)

Using above formula will result in below matrices



//backslashCode



//Slashcode

The ‘N – 1’ in the backslash code is there to ensure that the codes are never negative because we will be using the codes as indices in an array.

For placing a queen i on row j, check the following :

- whether row 'j' is used or not
- whether slashDiagnol 'i+j' is used or not
- whether backSlashDiagnol 'i-j+7' is used or not

If the answer to any one of the following is true, we try another location for queen i on row j, mark the row and diagonals; and recur for queen i+1.

Explanation

consider the example of a 4x4 chessboard
initially, board matrix:
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0

rowLook array:

[false, false, false, false]

slashDiagnolLook array (size = 2xn-1 = 7):

[false, false, false, false, false, false, false]

backSlashDiagnolLook array (size = 2xn-1 = 7):

[false, false, false, false, false, false, false]

preprocessed matrices are as follows:

3	2	1	0
4	3	2	1
5	4	3	2
6	5	4	3

slashDiagnol[4][4]

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

backSlashDiagnol[4][4]

Conclusion:

Hence, We have successfully implemented 8 queens using branch and bound strategy

FAQs:

1. Write Control abstraction for branch and bound method?
2. Discuss FIFO branch and bound strategy with example?
3. Discuss LC branch and bound strategy with example?

Virtual lab
Assignment No: 09

TITLE: Implement deterministic quick sort

OBJECTIVE: Analyze performance of an algorithm.

OUTCOME: Analyze performance of an algorithm.

THEORY:

Virtual lab link: https://ds1-iiith.vlabs.ac.in/exp/quick-sort/index.html

In this experiment, we will be able to do the following:

- Given an unsorted array of numbers, generate a sorted array of numbers by applying Quick Sort.
- Given an unsorted array of numbers, we will understand how to partition an array. This is an important concept in Quick Sort which is used recursively on the partitioned arrays to sort the numbers.

Conclusion: Hence we have successfully studied quick sort and analysis of algorithm