EXPERIMENT NO 8

Title: To implement the Nqueen problem using a backtracking approach and analyze its complexity.

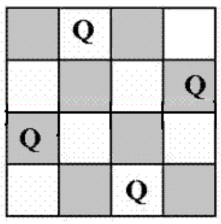
Aim: - Implement N Queens problem using backtracking.

Lab Outcomes: CSL401.4: Ability to implement the algorithm using Backtracking Programming approach.

Theory:-

In chess, a queen can move as far as she pleases, horizontally, vertically, or diagonally. A chess board has 8 rows and 8 columns. The standard 8 by 8 Queens' problem asks how to place 8 queens on an ordinary chess board so that none of them can hit any other in one move.

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



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

```
{0, 1, 0, 0}
{0, 0, 0, 1}
{1, 0, 0, 0}
{0, 0, 1, 0}
```

Backtracking Algorithm

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.

WHAT IS 8 QUEEN PROBLEMS?

✔ The **eight queen's puzzle** is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens attack each other.

✔ Thus, a solution requires that no two queens share the same row, column, or diagonal.

✔ The eight queen's puzzle is an example of the more general ***n*-queens problem** of placing *n* queens on an *n*×*n* chessboard, where solutions exist for all natural numbers *n* with the exception of 1*, 2 and 3.*

✔ The solution possibilities are discovered only up to *23 queens*.

BACKTRACKING CONCEPT

✔ Each recursive call attempts to place a queen in a specific column.

✔ For a given call, the state of the board from previous placements is known (i.e. where are the other queens?)

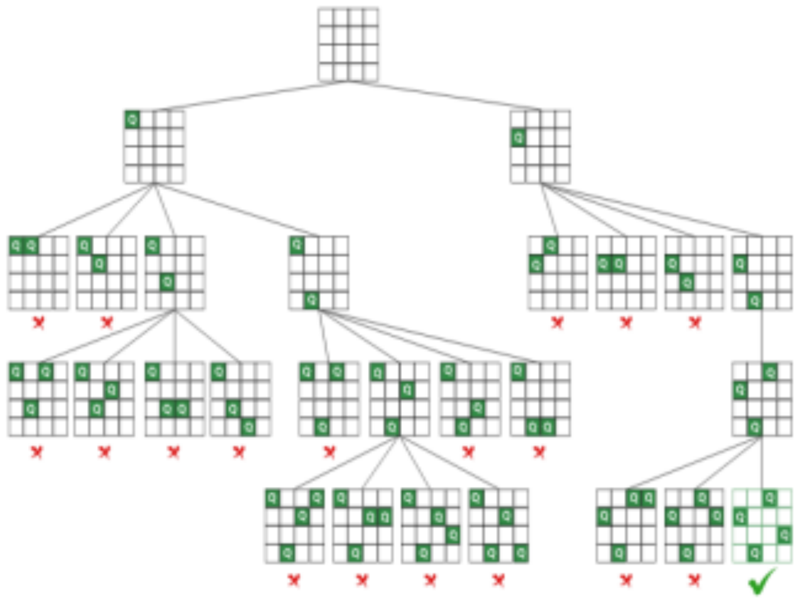
✔ ***Current step backtracking***: If a placement within the column does not lead to a solution, the queen is removed and moved ***"down"*** the column

✔ ***Previous step backtracking***: When all rows in a column have been tried, the call terminates and ***backtracks to the previous call*** (in the previous column)

✔ **Pruning:** If a queen cannot be placed into column i, do not even try to place one onto column i+1 – rather, backtrack to column i-1 and move the queen that had been placed there. Using this approach we can reduce the number of potential solutions even more

STEPS REVISITED – BACKTRACKING

1. Place the first queen in the left upper corner of the table.
2. Save the attacked positions.
3. Move to the next queen (which can only be placed to the next line).
4. Search for a valid position. If there is one go to step 8.
5. There is not a valid position for the queen. Delete it (the x coordinate is 0).
6. Move to the previous queen.
7. Go to step 4.
8. Place it to the first valid position.
9. Save the attacked positions.
10. If the queen processed is the last stop otherwise go to step 3.



Complexity Analysis

Time complexity: O(N!): The first queen has N placements, the second queen must not be in the same column as the first as well as at an oblique angle, so the second queen has N-1 possibilities, and so on, with a time complexity of O(N!).

Spatial Complexity: O(N): Need to use arrays to save information.

Program:-

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int a[30],count=0;
int place(int pos)
{
int i;
for(i=1;i<pos;i++)
{
if((a[i]==a[pos])||((abs(a[i]-a[pos])==abs(i-pos))))
return 0;
}
return 1;
```

```
}
void print_sol(int n)
{
int i,j;
count++;
printf("\n\nSolution #%d:\n",count);
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
if(a[i]==j)
printf("Q\t");
else
printf("*\t");
}
printf("\n");
}
}
void queen(int n)
{
int k=1;
a[k]=0;
while(k!=0)
{
a[k]=a[k]+1;
while((a[k]<=n)&&!place(k))
a[k]++;
if(a[k]<=n)
{
if(k==n)
print_sol(n);
else
{
k++;
a[k]=0;
}
}
else
k--;
}
}
void main()
{
int i,n;
clrscr();
printf("Enter the number of Queens\n");
```

```
scanf("%d",&n);
queen(n);
printf("\nTotal solutions=%d",count);
getch();
}
```

Output:-

```
Enter the number of Queens   4
Solution #1:
*    Q    *    *
*    *    *    Q
Q    *    *    *
*    *    Q    *

Solution #2:
*    *    Q    *
Q    *    *    *
*    *    *    Q
*    Q    *    *

Total solutions=2
```

Conclusion: - Thus we have implemented the N Queen's problem using Backtracking.

EXPERIMENT 9

Title: To implement Naïve String Matching Algorithm and Rabin Karp algorithm and analyze its complexity.

Aim: To implement Naive String Matching and Rabin Karp algorithm using String Matching.

Lab Outcomes: CSL401.5: Compare the complexity of the algorithms for the specific string matching problems.

Theory:

Naive String Matching

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.

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:

Best Case: O(n)

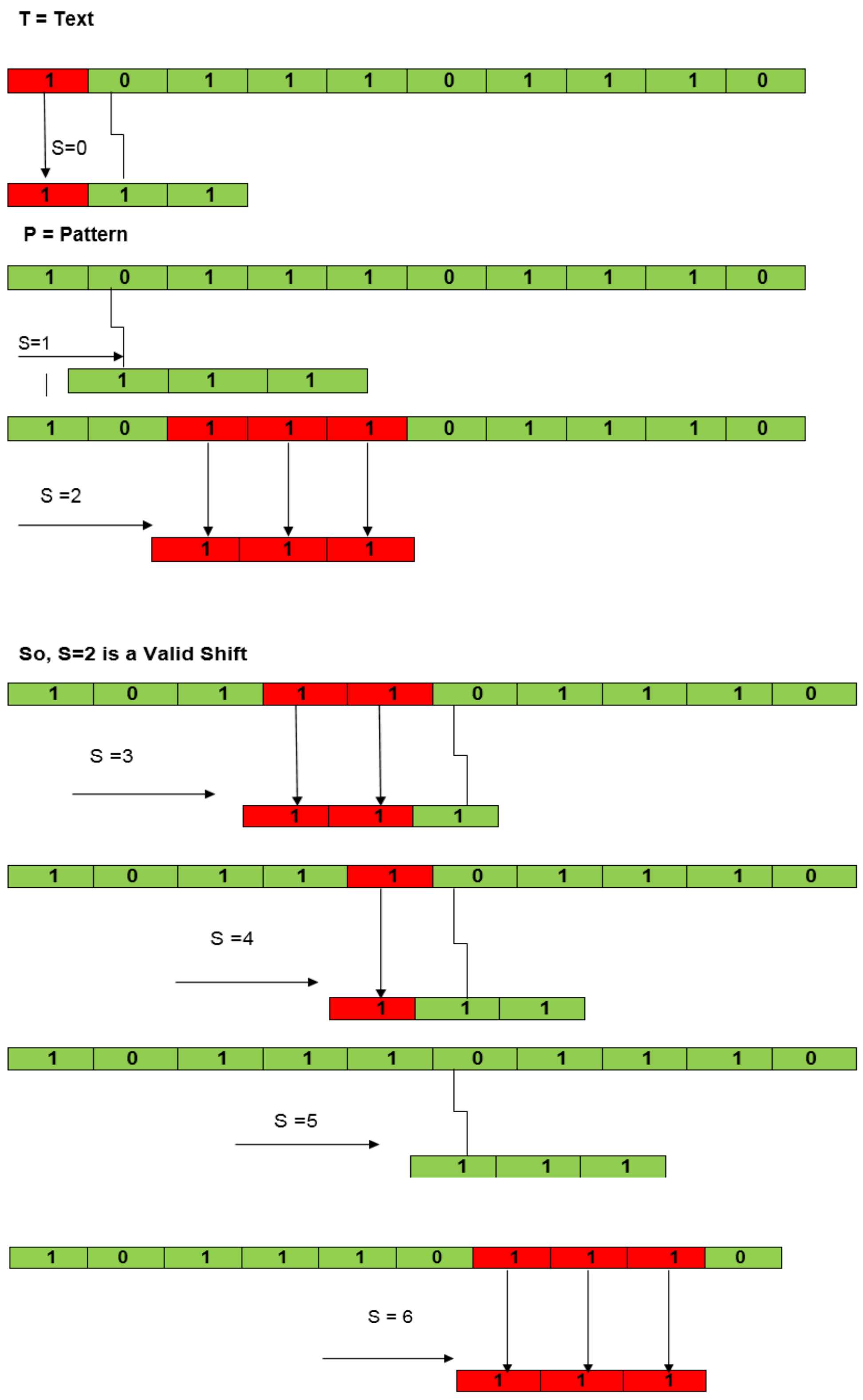
- When the **pattern** is found at the very beginning of the **text** (or very early on).
- The algorithm will perform a constant number of comparisons, typically on the order of **O(n)** comparisons, where n is the length of the **pattern**.

Worst Case: O(n^2)

- When the **pattern** doesn't appear in the **text** at all or appears only at the very end.
- The algorithm will perform **O((n-m+1)*m)** comparisons, where **n** is the length of the **text** and **m** is the length of the **pattern**.
- In the worst case, for each position in the **text**, the algorithm may need to compare the entire **pattern** against the text.
- Naive algorithm for Pattern Searching

Example:

1. Suppose T = 1011101110
2. P = 111
3. Find all the Valid Shift


T = Text
1 0 1 1 1 0 1 1 1 0
S=0
1 1 1
P = Pattern
1 0 1 1 1 0 1 1 1 0
S=1
1 1 1
1 0 1 1 1 0 1 1 1 0
S =2
1 1 1
So, S=2 is a Valid Shift
1 0 1 1 1 0 1 1 1 0
S =3
1 1 1
1 0 1 1 1 0 1 1 1 0
S =4
1 1 1
1 0 1 1 1 0 1 1 1 0
S =5
1 1 1
1 0 1 1 1 0 1 1 1 0
S = 6
1 1 1

Program:

```
#include <stdio.h>

#include <string.h>

int main (){

   char txt[] = "tutorialsPointisthebestplatformforprogrammers";

   char pat[] = "a";

   int M = strlen (pat);

   int N = strlen (txt);

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

      int j;

      for (j = 0; j < M; j++)

         if (txt[i + j] != pat[j])

      break;

      if (j == M)

         printf ("Pattern matches at index %d

", i);

   }

   return 0;

}
```

Output

```
Pattern matches at 6

Pattern matches at 25

Pattern matches at 39
```

Rabin Karp Algorithm

Rabin-Karp algorithm is an algorithm used for searching/matching patterns in the text using a hash function. Unlike Naive string matching algorithm, it does not travel through every character in the initial phase, rather it filters the characters that do not match and then performs the comparison.

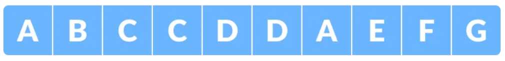
A hash function is a tool to map a larger input value to a smaller output value. This output value is called the hash value.

How does the Rabin-Karp Algorithm Work?

A sequence of characters is taken and checked for the possibility of the presence of the required string. If the possibility is found then, character matching is performed.

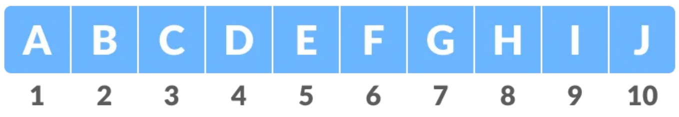
Let us understand the algorithm with the following steps:

Let the text be:



And the string to be searched in the above text be:

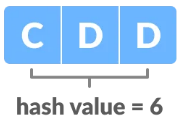
Let us assign a numerical value(v)/weight for the characters we will be using in the problem. Here, we have taken first ten alphabets only (i.e. A to J).


A B C D E F G H I J
1 2 3 4 5 6 7 8 9 10

n be the length of the pattern and m be the length of the text. Here, m = 10 and n = 3.

Let d be the number of characters in the input set. Here, we have taken the input set {A, B, C, ..., J}. So, d = 10. You can assume any suitable value for d.

Let us calculate the hash value of the pattern.



hash value for pattern(p) = Σ(v * dm-1) mod 13

= ((3 * 102) + (4 * 101) + (4 * 100)) mod 13

= 344 mod 13

= 6

In the calculation above, choose a prime number (here, 13) in such a way that we can perform all the calculations with single-precision arithmetic.

The reason for calculating the modulus is given below.

Calculate the hash value for the text-window of size m.

For the first window ABC,

hash value for text(t) = Σ(v * dn-1) mod 13

= ((1 * 102) + (2 * 101) + (3 * 100)) mod 13

= 123 mod 13

= 6

Compare the hash value of the pattern with the hash value of the text. If they match then, character-matching is performed.

In the above examples, the hash value of the first window (i.e. t) matches with p so, go for character matching between ABC and CDD. Since they do not match so, go for the next window.

We calculate the hash value of the next window by subtracting the first term and adding the next term as shown below.

t = ((1 * 102) + ((2 * 101) + (3 * 100)) * 10 + (3 * 100)) mod 13

= 233 mod 13

= 12

In order to optimize this process, we make use of the previous hash value in the following way.

t = ((d * (t - v[character to be removed] * h) + v[character to be added]) mod 13

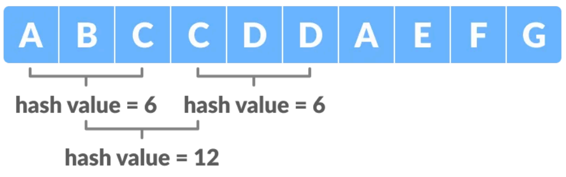
= ((10 * (6 - 1 * 9) + 3)mod 13

= 12

Where, h = dm-1 = 103-1 = 100.

For BCC, t = 12 (≠6). Therefore, go for the next window.

After a few searches, we will get the match for the window CDA in the text.



Algorithm:

```
RABIN-KARP-MATCHER$(T, P, d, q)$
    $n = T.length$
    $m = P.length$
    $h = d^{m-1} \bmod q$
    $p = 0$
    $t_0 = 0$
    for $i = 1$ to $m$                    // preprocessing
        $p = (dp + P[i]) \bmod q$
        $t_0 = (dt_0 + T[i]) \bmod q$
    for $s = 0$ to $n - m$                // matching
        if $p == t_s$
            if $P[1..m] == T[s+1..s+m]$
                print "Pattern occurs with shift" $s$
        if $s < n - m$
            $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$
```

Program:

```
#include< stdio.h>
#include< conio.h>
#define tonum(c) (c >= 'A' && c <= 'Z' ? c - 'A' : c - 'a' + 26)
int mod(int a,int p,int m)
{
 int sqr;

 if (p == 0)
  return 1;

 sqr = mod(a,p/2,m) % m;
 sqr = (sqr * sqr) % m;

 if (p & 1)
  return ((a % m) * sqr) % m;
 else
```

```
  return sqr;
}
int RabinKarpMatch(char *T,char *P,int d,int q)
{
 int i,j,p,t,n,m,h,found;
 n = strlen(T);
 m = strlen(P);
 h = mod(d,m-1,q);
 p = t = 0;
 for (i=0; i < m; i++)
 {
  p = (d*p + tonum(P[i])) % q;
  t = (d*t + tonum(T[i])) % q;
 }
 for (i=0; i<= n-m; i++)
 {
  if (p == t)
  {
   found = 1;

   for (j=0; j < m; j++)
    if (P[j] != T[i+j])
    {
     found = 0;
     break;
    }
       if (found) return i;
  }
  else
  {
   t = (d*(t - ((tonum(T[i])*h) % q)) + tonum(T[i+m])) % q;
  }
 }
return -1;
}
void main()
{
 char *str;
 char *p;
 int ans,q;
 clrscr();
 printf("\n Enter String:");
 gets(str);
 printf("\n Enter Pattern you want to search into string:");
 gets(p);
 printf("\n Enter value of q:");
 scanf("%d",&q);
 ans=RabinKarpMatch(str,p,10,q);
 if(ans==-1)
  printf("\n Pattern is not found.");
 else
  printf("\n Pattern is found at displacement:%d",ans);
getch();
}
```

Output:

```
Enter String: Rabin karp algorithm for pattern matching

Enter Pattern you want to search into string: pattern

Enter value of q: 11

Pattern is found at displacement: 25
```

Conclusion: Thus we have implemented the Naive String matching and Rabin Karp algorithm using string matching.

.