ASSIGNMENT-3

5/08/2024

M Satvi Reddy
192311038
CSA-0688-DAA

1. Compare the brute force approach with more efficient algorithms like knuth-Morris-Pratt (KMP) or Boyer-Moore.

→ Knuth-Morris-Pratt Algorithm.

- uses a preprocessing phase to create a partial match table that allows the algorithm to skip unnecessary compaissions.
- O (n+m) efficient for large texts.

→ Booyer-Moore Algorithm.

- compares pattern from right to left and uses two hurristics to skip reaction of the text.
- time complexity - O(n/m)
- Outperform many algorithm due to the efficiency to skip large reaction of text.

2. To analyze the the bubble sort algorithm, understand its working principles, efficiency and limitations. and compare it with other sorting algorithm.

Working principle: compares adjacent elements repeatedly, swaps then if they are in wrong order process in continued until the orders in sorted.

Efficiency : Best case: $O(n)$
Worst case: $O(n^2)$

In efficient for large lists.

Insution sort - $O(n^2)$ in worst case.

merge sort - $O(n \log n)$ all cases. efficient for large data.

Quick sort - $O(n \log n)$ one average $O(n^2)$ at worst.

3. Implement a brute force algorithm to stimulated a password cracking attempt. Given a hasted password and a character set, The algorithm should attempt all possible combinative. Go find the original password. Discuss the time complexity of your algorithm and how it values with the length of the password and size of the character set.

```
import itertools
import hashlib
```

```
def btc(h-p, closet, m-l):
    for i in range (1, m-l+1):
        for j in itertools.product(closet, repeat=i):
            guess = "".join(j)
            if hashlib.la25(guess.encode).hexdigest()
                == h-p:
                return guess
    return None.

h-p = hashlib.sta 256('pass'%.encode()).hexdigit()
closet = "abcdefghijhlmnopqrstuvwxyz"
maxlength = 8
find password = bfc(n-p, closet, maxlength)
print (find password)
```

4. Implement a brutefore algorithm for numbers matching given a text and a pattern, find all occurences of the pattern in the text. Analyse the time complexity of your brute force string matching algorithm.

```
def brute (text, pattern)
    n, m = len (text), len (pattern):
    Occurence = [ ]
    for i in range (n - m+1):
        if text [i: )+m ] == pattern:
            occurence.append (p)
```

```
        return occurence.
text = "abracadbra"
pattern = "abr"
print(brute(text, pattern))
```

5. Implement a bruteforce algorithm to solve sudoku puzzles. Given a axa grid, fill the empty cells so that each row, column and 3x3 grid contains all digits from 1 to 9. Discuss the time complexity of you bruteforce sudoku solves.

```
def B-valid(b, a, c, n):
    for i in range(a):
        if b[r][i] == n or b[i][c] == n
            or b[r - r%3 + r//3][c - c%3 + i%3] == 'N'.
        return False.
    return true
def sudoku_solves(b):
    for r in rangl(9):
        for c in rangl(9):
            if board[r][c] == 0:
                for n in rangl(1, 10):
                    if is valid(b, r, c, n):
                        b[r][c] = num
                        if sudoku solves(b):
                            return True.
                        b[r][c] = 0
```

return False.

return True.

b. [

[5, 3, 0, 0, 7, 0, 0, 0, 0],

[6, 0, 0, 1, 9, 5, 0, 0, 0],

[0, 9, 8, 0, 0, 0, 0, 6, 0],

[8, 0, 0, 0, 6, 0, 0, 0, 3],

[4, 0, 0, 8, 0, 3, 0, 0, 1],

[7, 0, 0, 0, 2, 0, 0, 0, 6],

[0, 6, 0, 0, 0, 2, 8, 0],

[0, 0, 0, 4, 1, 9, 0, 0, 9],

[0, 0, 0, 0, 8, 0, 0, 7, 9],

]

if sodoku.solves(h):

for x in board:

print(x)

else

print ("NO solution")