M. Rahul Ram,
CSE-B,
185001121.

15-01-2020

UCS 1403 - Design and analysis of algorithms.

1. a) Algorithm :-

Let us consider that array is used to describe a stack of n pancakes.

Step 1 : Iterate ~~through~~ from 0 to n-1 (using variable i).

Step 2 :- Inside that loop, again iterate from 0 to i. (using variable j).

inside first for loop (i+1)

Step 2 :- If sizeof pancake ~~(j)~~ index < sizeof pancake (i) index.

(if inversion exists).

→ Flip pancakes from (0 to i+1).
→ (last will be first)

→ For k in range (0, i+1)

→ check for pancake with size less than the first one.

→ if found ~~initialize~~ assign its index to flip_index.

→ if not assign (i+2) to flip_index.

→ perform flip (0 : flip_index - 1)

→ perform flip (0 : flip_index - 2).

Step 3 : inside first for loop. perform flip (0 : i+1)

algorithm for flip: (starting always 0)

flip(~~start-index~~, end-index) :

→ for i in range (end-index / 2)

→ swap (i, end-index - i).

b) consider sequence ~~2 3 6~~ | 2 | 4 | 6 | 3 | 8 |
0 1 2 3 4

(i) inversion at 6, 3.

(ii) fliping (0, 3) → | 3 | 6 | 4 | 2 | 8 |

(iii) finding flip-index as ~~3~~ 2.

(iv) flipping $(0, \frac{3+1}{2})$ → | 4 | 6 | 3 | 2 | 8 |

(v) flipping $(0, \frac{3-2}{1})$ → | 6 | 4 | 3 | 2 | 8 |

(vi) flipping (0, 3) → | 2 | 3 | 4 | 6 | 8 |

c) ~~A X 4~~ $(n-1) \times 4$

d) ~~A X 4~~. $(n-1) \times 4$.

4)

a) Here's the algorithm / code

```
def isBalanced(w):
    # w is empty string
    if len(w) == 0:
        return True

    # else create a empty stack
    stack = []
    # Now traverse through the string.
    for character in w:
        if character == '(':
            # append it to stack
            stack.append(character)
        # else -> character == ')'
        else:
            # no opening parenthesis
            if len(stack) == 0:
                return False

            else:
                c = stack.pop()
                if c != '(':
                    return False

    # if control reaches here,
    # if stack is no empty
    if len(stack) != 0:
        return False

    return True # every parenthesis is matched.
```

Analysis

(i) if string is empty, return true

(ii) return false when no matching '(' is found in at the stack when ')' is found in the string.

(iii) return false when stack has element, after traversing through the string means no matching ')' for '(' found in stack.

This loop runs till the ends of the string (n -> len of string), if it is if not returned false before it the for loop ends

worst case complexity - O(n).

4 b)

```
def longestSequence(w)
    if len(w) == 0:
        return 0.

    stak = []

    max = 0
    length = 0
    for char in w:
        if char == '(':
            stack.append(char)

        else:
            # if no matching '(' is found.
            if len(stack) == 0:
                # make length as 0. subsequence ends
                length = 0.
            # else add 2 to length (')' and '(')
            else:
                stack.pop()

                length += 2
                # if length is greater than max
                if length > max:
                    # assign new length to max.
                    max = length


    # return max.
    return max.
```

Analysis

(i) if string is empty, length of longest subsequence = 0.

(ii) Here whenever a matching pair of '(' and ')' is found we increment 2 to length. and check if it is greater than max if yes, assign length to max.

(iii) if no matching '(' found for ')' → Then the subsequence ends. we again start from (length) = 0.

(iv) finally, we return max.

worst - time complexity = O(n).

3)

Dimensions of envelope is $x_i x y_i$.

There are two cases of each envelope choose it or not select it.

Use dp array to hold maximum number of nests per envelope, then.

$$dp[x] = MAX(dp[e_1] + 1, dp[e_2] + 1, dp[e_3] + 1, \ldots).$$

where $x \rightarrow$ envelopes e_1, e_2, e_3

$dp[e_1] \rightarrow$ corresponds largest envelope representing n_1, the number of nests.

Algorithm:

1) sort the envelop,
   ~~one~~ based on x_i
   ~~another~~ based on y_i.

2) dp array will hold the maximum value of each envelop, iterating over the array and update it.

3) when iterating, we need to check if all envelopes after this can be nested within it.

4) return max value.