Tutorial-4

IIIT Delhi

Instructor: Debarka Sengupta

Problem 1.

Suppose you are managing the construction of billboards on the Stephen Daedalus Memorial Highway, a heavily travelled stretch of road that runs west-east for M miles. The possible sites for billboards are given by numbers x_1 , x_2 , . . . , x_n , each in the interval [0, M] (specifying their position along the highway, measured in miles from its western end). If you place a billboard at location x_i , you receive revenue of $r_i > 0$. Regulations imposed by the county's Highway Department require that no two of the billboards be within less than or equal to 5 miles of each other. You'd like to place billboards at a subset of the sites so as to maximize your total revenue, subject to this restriction.

Example. Suppose M = 20, n = 4,
{x 1 , x 2 , x 3 , x 4 } = {6, 7, 12, 14},
and
{r 1 , r 2 , r 3 , r 4 } = {5, 6, 5, 1}.
Then the optimal solution would be to place billboards at x 1 and x 3 , for total revenue of 10. Give an algorithm that takes an instance of this problem as input and returns the maximum total revenue that can be obtained from any valid subset of sites. The running time of the algorithm should be polynomial in n.

Solution 1.

For a site x_j , we let e(j) denote the easternmost site x_i that is more than 5 miles from x_j . Since sites are numbered west to east, this means that the sites x_1 , x_2 , . . . , $x_{e(j)}$ are still valid options once we've chosen to place a billboard at x_j , but the sites $x_{e(j)+1}$, . . . , x_{j-1} are not. If we let OPT (j) denote the revenue from the optimal subset of sites among x_1 , . . . , x_j , then we have **OPT (j) = max(r_j+ OPT (e(j)), OPT (j − 1))**

Define an array M for OPT values.

```
Initialize M[0] = 0 and M[1] = r 1
For j = 2, 3, . . . , n :
Compute M[j] using the recurrence
End for
Return M[n]
```

For details, refer Algorithm Design by Tardos, pg 308

Problem 2.

The problem of searching for cycles in graphs arises naturally in financial trading applications. Consider a firm that trades shares in n different companies. For each pair i = j, they maintain a trade ratio r_{ij} , meaning that one share of i trades for r_{ij} shares of j. Here we allow the rate r to be fractional; that is, r_{ij} = 2/3 means that you can trade three shares of i to get two shares of j. A trading cycle for a sequence of shares i_1 , i_2 , . . . , i_k consists of successively trading shares in company i_1 for shares in company i_2 , then shares in company i_2 for shares i_3 , and so on, finally trading shares in i_k back to shares in company i_1 . After such a sequence of trades, one ends up with shares in the same company i_1 that one starts with. Trading around a cycle is usually a bad idea, as you tend to end up with fewer shares than you started with. But occasionally, for short periods of time, there are opportunities to increase shares. We will call such a cycle an opportunity cycle, if trading along the cycle increases the number of shares. This happens exactly if the product of the ratios along the cycle is above 1. In analyzing the state of the market, a firm engaged in trading would like to know if there are any opportunity cycles.
Give a polynomial-time algorithm that finds such an opportunity
cycle, if one exists

Solution 2.

Construct a graph with a node for each stock, and a directed edge (i,j) for each pair of stocks. The cost of edge (i,j) is -log(r_{ij}).

Given that a trading cycle is an opportunity cycle iff:

$$\prod r_{ij} > 1$$

For all (i,j) in the trading cycle.

From the above, you can derive that a cycle C is an opportunity cycle iff it is a negative cycle (take log on both sides). Use a negative cycle detection algorithm to detect whether an opportunity cycle exists.

Problem 3.

Given a tree, color as many nodes black as possible, without coloring two adjacent nodes.

Solution 3.

Subproblems:

1. We arbitrarily decide the root node r

2. Bv: the optimal solution for a subtree having v as the root, where we color v black
3. Wv: the optimal solution for a subtree having v as the root, where we don't color v
4. The answer is max{Br, Wr}

Problem 4.

Implement wildcard pattern matching with support for '?' and '*'.

- '? ': Matches any single character.
- '*' : Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

Examples:

isMatch("aa","a") → 0
isMatch("aa","aa") → 1
isMatch("aaa","aa") → 0
isMatch("aa", "*") → 1
isMatch("aa", "a*") → 1
isMatch("ab", "?*") → 1
isMatch("aab", "c*a*b") → 0

(isMatch is example function for the question)

Solution 4.

Think about the brute-force solution.
When you encounter ‘’, you would try to call the same isMatch function in the following manner:
If p[0] == ‘’, then isMatch(s, p) is true if isMatch(s+1, p) is true OR isMatch(s, p+1) is true.
else if p[0] is not ‘*’ and the characters s[0] and p[0] match (or p[0] is ‘?’), then isMatch(s,p) is true only if isMatch(s+1, p+1) is true.
If the characters don’t match isMatch(s, p) is false.
This approach is exponential. Think why.
Let's see how we can make this better. Note that isMatch function can only be called with suffixes of s and p. As such, there could only be length(s) * length(p) unique calls to isMatch.
Let's just memoize the result of the calls so we only do processing for unique calls. This makes the time and space complexity O(len(s) * len(p)).
There could be ways of optimizing the approach rejecting certain suffixes without processing them. For example, if len(non-star characters in p) > len(s), then we can return false without checking anything.

```
public class Solution {
  public int isMatch(final String A, final String B) {
    boolean[][] dp =new boolean[A.length()+1][B.length()+1];
    dp[0][0]=true;
    for(int i=1;i<=B.length();i++)
    {
      if(B.charAt(i-1)=='*')
        dp[0][i]=true;
      else
        break;
    }
    for(int i=1;i<=A.lngth();i++)
    {
      for(int j=1;j<=B.length();j++)
      {
        if(B.charAt(j-1)=='?' || A.charAt(i-1)==B.charAt(j-1))
          dp[i][j]=dp[i-1][j-1];
        else
        if(B.charAt(j-1)=='*')
          dp[i][j]=dp[i-1][j-1] || dp[i-1][j] || dp[i][j-1];
      }
    }
    return dp[A.length()][B.length()]?1:0;
  }
}
```

Problem 5.

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.
For "(()", the longest valid parentheses substring is "()", which has length = 2.
Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

Solution 5.

Let's construct longest[i] where longest[i] denotes the longest set of parenthesis ending at index.

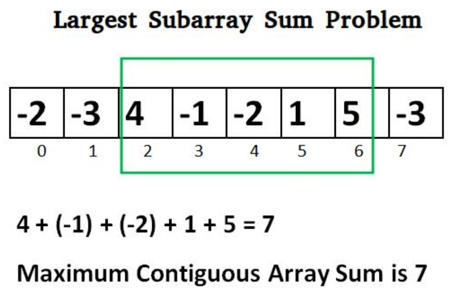
1. If s[i] is ‘(‘, set longest[i] to 0, because any string end with ‘(‘ cannot be a valid one.

```
2. Else if s[i] is ‘)’
   If s[i-1] is ‘(‘, longest[i] = longest[i-2] + 2
   Else if s[i-1] is ‘)’ and s[i-longest[i-1]-1] == ‘(‘, longest[i] = longest[i-1] + 2 +
   longest[i-longest[i-1]-2]
```

```
public class Solution {
   public int longestValidParentheses(String s) {
      int[] dp=new int[s.length()];
      Stack<Integer> stack=new Stack<>();
      for(int i=0;i<s.length();i++){
         char ch=s.charAt(i);
         if(ch=='('){
            stack.push(i);
         }else{
            if(!stack.isEmpty()){
               dp[i]=1;
               dp[stack.pop()]=1;
            }
         }
      }
      int count=0;
      int max=0;
      for(int i=0;i<dp.length;i++){
         if(dp[i]==1)
            count++;
         else {
            max=Math.max(count,max);
            count=0;
         }
      }
      max=Math.max(count,max);
      return max;
   }
}
```

Problem 6.

Write an efficient program to find the sum of contiguous subarray within a one-dimensional array of numbers which has the largest sum.



(source: GFG)

Solution 6.

Kadane's Algorithm:

```
Initialize:
    max_so_far = 0
    max_ending_here = 0

Loop for each element of the array
  (a) max_ending_here = max_ending_here + a[i]
  (b) if(max_ending_here < 0)
          max_ending_here = 0
  (c) if(max_so_far < max_ending_here)
          max_so_far = max_ending_here
return max_so_far
```

Explanation:

The simple idea of the Kadane's algorithm is to look for all positive contiguous segments of the array (max_ending_here is used for this). And keep track of the maximum sum contiguous segment among all positive segments (max_so_far is used for this). Each time we get a positive sum compare it with max_so_far and update max_so_far if it is greater than max_so_far

Lets take the example:

{-2, -3, 4, -1, -2, 1, 5, -3}

max_so_far = max_ending_here = 0

for i=0, a[0] = -2
max_ending_here = max_ending_here + (-2)
Set max_ending_here = 0 because max_ending_here < 0

for i=1, a[1] = -3
max_ending_here = max_ending_here + (-3)
Set max_ending_here = 0 because max_ending_here < 0

for i=2, a[2] = 4
max_ending_here = max_ending_here + (4)
max_ending_here = 4
max_so_far is updated to 4 because max_ending_here greater than max_so_far which was 0 till now

for i=3, a[3] = -1
max_ending_here = max_ending_here + (-1)
max_ending_here = 3

for i=4, a[4] = -2
max_ending_here = max_ending_here + (-2)
max_ending_here = 1

for i=5, a[5] = 1
max_ending_here = max_ending_here + (1)
max_ending_here = 2

for i=6, a[6] = 5
max_ending_here = max_ending_here + (5)
max_ending_here = 7
max_so_far is updated to 7 because max_ending_here is greater than max_so_far

for i=7, a[7] = -3
max_ending_here = max_ending_here + (-3)
max_ending_here = 4

```
class Kadane
{
   static int maxSubArraySum(int a[])
   {
      int size = a.length;
      int max_so_far = Integer.MIN_VALUE, max_ending_here = 0;

      for (int i = 0; i < size; i++)
```

```
        {
            max_ending_here = max_ending_here + a[i];
            if (max_so_far < max_ending_here)
                max_so_far = max_ending_here;
            if (max_ending_here < 0)
                max_ending_here = 0;
        }
        return max_so_far;
    }
}
```

Problem 7.

Imagine you have a special keyboard with the following keys:
Key 1: Prints 'A' on screen
Key 2: (Ctrl-A): Select screen
Key 3: (Ctrl-C): Copy selection to buffer
Key 4: (Ctrl-V): Print buffer on screen appending it
after what has already been printed.
If you can only press the keyboard for N times (with the above four keys), write a program to produce maximum numbers of A's. That is to say, the input parameter is N (No. of keys that you can press), the output is M (No. of As that you can produce).

Examples:
Input: N = 3
Output: 3
We can at most get 3 A's on screen by pressing
following key sequence.
A, A, A

Input: N = 7
Output: 9
We can at most get 9 A's on screen by pressing
following key sequence.
A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V

Input: N = 11
Output: 27
We can at most get 27 A's on screen by pressing

following key sequence.
A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V, Ctrl A,
Ctrl C, Ctrl V, Ctrl V

Solution 7.

Below are a few important points to note.
a) For N < 7, the output is N itself.
b) Ctrl V can be used multiple times to print current buffer (See last two examples above). The idea is to compute the optimal string length for N keystrokes by using a simple insight. The sequence of N keystrokes which produces an optimal string length will end with a suffix of Ctrl-A, a Ctrl-C, followed by only Ctrl-V's. (For N > 6) The task is to find out the break=point after which we get the above suffix of keystrokes. Definition of a breakpoint is that instance after which we need to only press Ctrl-A, Ctrl-C once and the only Ctrl-V's afterwards to generate the optimal length. If we loop from N-3 to 1 and choose each of these values for the break-point, and compute that optimal string they would produce. Once the loop ends, we will have the maximum of the optimal lengths for various breakpoints, thereby giving us the optimal length for N keystrokes.

```
class MaxA {
  static int findoptimal(int N)
  {
    if (N <= 6)
      return N;
    // Initialize result
    int max = 0;
    // TRY ALL POSSIBLE BREAK-POINTS
    // For any keystroke N, we need to
    // loop from N-3 keystrokes back to
    // 1 keystroke to find a breakpoint
    // 'b' after which we will have Ctrl-A,
    //  Ctrl-C and then only Ctrl-V all the way.
    int b;
    for (b = N - 3; b >= 1; b--)
    {
        // If the breakpoint is s at b'th
        // keystroke then the optimal string
        // would have length
        // (n-b-1)*screen[b-1];
        int curr = (N - b - 1) * findoptimal(b);
```

```
            if (curr > max)
                max = curr;
        }
        return max;
    }
```