1. Binary String Transition:

First, create list c and list d, iterate through binary string a and b, if a_i equals to b_i, go next; if a_i is not equal to b_i, add a_i into c and b_i into d until the iteration process is finished. Time Complexity: O(len(a)) = O(n)

Now we get list c and list d, they consist of different element in string a and string b.

Consider the swap and negate process, based on the condition that c_i is not equal to d_i, given pointer i and pointer j, we can easily conclude that if $|i-j|=1$, the cost of swap is smaller than the cost of negate, otherwise, it is cheaper to do negate.

Back to the algorithm, set a pointer iterate in list c, and the iterative process is shown below:

cost = 0
i = 0
if $c_i = c_{i+1}$:
    cost += 2 // negate both c_i and c_{i+1}
    $c_i = 1 - c_i$
    $c_{i+1} = 1 - c_{i+1}$
    i += 2 // pointer points to c_{i+2}
else:
    cost += 1 // swap c_i and c_{i+1}
    $temp = c_i$
    $c_i = c_{i+1}$
    $c_{i+1} = temp$
    i += 2 // pointer points to c_{i+2}

Because we iterate from head to tail of list c in order to calculate the cost, so the time complexity is O(length(c))

Pseudo-Code:

```
def BST(a, b):
    c, d = get_different(a, b)
    cost = calculate_cost(c, d)
    return cost

def get_different(a, b):
    c = [], d = [], j = 0
    for i in range(len(a)):
        if not a[i] = b[i]:
            c[j] = a[i]
            d[j] = b[i]
            j += 1
    return c, d
```

```
def calculate_cost(c, d):
    cost = 0
    for i in range(len(c)):
        if c_i = c_{i+1}:
            cost += 2        // negate both c_i and c_{i+1}
            i += 2           // pointer points to c_{i+2}
        else:
            cost += 1        // swap c_i and c_{i+1}
            i += 2           // pointer points to c_{i+2}
    return cost
```

Total Time Complexity: O(n)

2. Longest Valid Parenthesis:

Construct dp[n][n], dp[i][j] means the longest valid parenthesis in the substring which starts at position i and ends at position j.

Recursive Formula:

$$dp[i][j] = \begin{cases} 2, & if\ i+1=j\ and\ (\ (\ s[i]='('\ and\ s[j]=')'\ )\ or\ (\ s[i]='['\ and\ s[j]=']'\ )\ ) \\ dp[i+1][j-1]+2, & if\ dp[i+1][j-1]>0\ and\ (\ (\ s[i]='('\ and\ s[j]=')'\ )\ or\ (\ s[i]='['\ and\ s[j]=']'\ )\ ) \\ dp[i][k]+dp[k+1][j], & if\ dp[i][k]>0\ and\ dp[k+1][j]>0,\ \ k\in(i,j) \end{cases}$$

when doing recursion, i ranges from len(S)-1 to 0, j ranges from i+1 to len(S) – 1. then, dp[0][len(S)-1] is the result of the longest valid parenthesis of the string S.

3. Data Change Problem

Every change-function is denoted as $CHG_i(l,r,x)$ and $CS(i,\dots,j)$ is the set of some change-functions. That is, $CS(i,\dots,j)$ consists of $CHG_i(l,r,x)$ to $CHG_j(l,r,x)$

This is a dynamic programming problem, cause it has the optimal sub-structure, the bigger problem can be solved by solving the smaller problems, and the sub-problems overlap. For example, if we want to calculate the optimal answer when Q = 4, we need to calculate CS[1,2,3], CS[1,2,4], CS[1,3,4], CS[2,3,4],

Then, it can be confirmed that CS[1,2], CS[1,3], CS[1,4], CS[2,3], CS[2,4] and CS[3,4] are calculated twice in order to calculate the four subproblems above.

If we can calculate all the sub-problems and save the answer down-up, we can solve the original problem in a time-consuming way.

The solving formula is shown below:

$$\min(n) = \min\{CS[2,3,\dots,n], CS[1,3,\dots,n], \dots, CS[1,\dots,n-3,n-1], CS[1,\dots,n-3,n-2]\}$$

In order to solve the origin problem $\min(n)$, we need to establish a space to store all the answer to solve $CS[i,\dots,j]$, the space-complexity is $O(n^2)$.

First, we calculate $CS[i,i+1]$, after all two-item CSs solved, we turn to solve all the three-item CSs based on all the two-item CSs, that is, $CS[i,i+1,i+2]$.

Iteratively, num of items rise from two to n-1, after number-n of n-1_CSs solved, the origin problem is the minimum of these n subproblems.

$$CS[i,\dots,j]=\begin{cases}CS[i,\dots,j-1], & if\ \text{max\_interval of CS[i, \dots, j} - 1]\ do\ not\ overlap\ with\ (l_j,r_j)\\ CS[i,\dots,j-1]+x_j, & if\ \text{max\_interval of CS[i, \dots, j} - 1]\ overlap\ with\ (l_j,r_j)\end{cases}$$

$$CS[i,i+1]=\begin{cases}\max(l_i,l_{i+1}), & if\ (l_i,r_i)\ and\ (l_{i+1},r_{i+1})\ do\ not\ overlap\\ l_i+l_{i+1}, & if\ (l_i,r_i)\ and\ (l_{i+1},r_{i+1})\ overlap\end{cases}$$

4. Max Matrix Problem

Max-Matrix(M) // Greedy Algorithm

Input: Matrix M

Output: Max-value of changed matrix M

```
zero_item = 0
for i in range(row(M)):
    if M_i0 = 0:
        zero_item += 1
if zero_item >= row(M)/2:
    M = Reverse-Column(M, 0)
for i in range(row(M)):
    if M_i0 = 0:
        M = Reverse-Row(M, i)
// above operations make every element in first column of M equals to 1
zero_item = 0
for j in range(1, column(M)):    // Change other columns 1 more than half
    for i in range(row(M)):      // Calculate zero in other columns
        if M_ij = 0:
            zero_item += 1
    if zero_item >= row(M)/2:
        M = Reverse-Column(M, j)
    zero_item = 0
// above operations greedy change M into new Max-Matrix M'
total = 0
for i in range(row(M)):
    for j in range(column(M)):
        if M_ij = 1:
            exp = column(M) – 1 - j
            total += 2^(exp)
return total
```