Best Time to Buy and Sell Stock

You are given an array `prices` where `prices[i]` is the price of a given stock on the `i`th day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0`.

Example 1:

```
Input: prices = [7,1,5,3,6,4]

Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.
```

Approach 1: Brute Force

```
public class Solution {
   public int maxProfit(int prices[]) {
      int maxprofit = 0;
      for (int i = 0; i < prices.length - 1; i++) {
         for (int j = i + 1; j < prices.length; j++) {
            int profit = prices[j] - prices[i];
            if (profit > maxprofit)
               maxprofit = profit;
         }
      }
      return maxprofit;
   }
}
```

Complexity Analysis

- Time complexity: $O(n^2)$. Loop runs $\dfrac{n (n-1)}{2}$ times.
- Space complexity: $O(1)$. Only two variables - maxprofit and profit are used.

Approach 2:

This problem can be solved using greedy approach. To maximize the profit we have to minimize the buy cost and we have to sell it on maximum price.

Follow the steps below to implement the above idea:

Declare a buy variable to store the buy cost and max_profit to store the maximum profit.

Initialize the buy variable to first element of profit array.

Iterate over the prices array and check if the current price is minimum or not.

If the current price is minimum then buy on this ith day.

If the current price is greater then previous buy then make profit from it and maximize the max_profit.

Finally return the max_profit.

```
int maxProfit(int prices[], int n)
{
   int buy = prices[0], max_profit = 0;
   for (int i = 1; i < n; i++) {

      // Checking for lower buy value
      if (buy > prices[i])
         buy = prices[i];

      // Checking for higher profit
      else if (prices[i] - buy > max_profit)
         max_profit = prices[i] - buy;
   }
   return max_profit;
}
```

Time Complexity: O(N). Where **N** is the size of prices array.
Auxiliary Space: O(1). We do not use any extra space.

Maximum Subarray

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.

A **subarray** is a **contiguous** part of an array.

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Output: 6

Explanation: [4,-1,2,1] has the largest sum = 6.
```

The naive method is to run two loops. The outer loop picks the beginning element, the inner loop finds the maximum possible sum with first element picked by outer loop and compares this maximum with the overall maximum. Finally, return the overall maximum. The time complexity of the Naive method is O(n^2).

Using **Divide and Conquer** approach, we can find the maximum subarray sum in O(nLogn) time. Following is the Divide and Conquer algorithm.

1. Divide the given array in two halves
2. Return the maximum of following three
    - Maximum subarray sum in left half (Make a recursive call)
    - Maximum subarray sum in right half (Make a recursive call)
    - Maximum subarray sum such that the subarray crosses the midpoint

```
3. static int maxCrossingSum(int arr[], int l, int m,
4.                                 int h)
5.     {
6.         // Include elements on left of mid.
7.         int sum = 0;
8.         int left_sum = Integer.MIN_VALUE;
9.         for (int i = m; i >= l; i--) {
10.                 sum = sum + arr[i];
11.                 if (sum > left_sum)
12.                     left_sum = sum;
13.             }
14.
15.             // Include elements on right of mid
16.             sum = 0;
17.             int right_sum = Integer.MIN_VALUE;
18.             for (int i = m + 1; i <= h; i++) {
19.                 sum = sum + arr[i];
20.                 if (sum > right_sum)
21.                     right_sum = sum;
22.             }
23.
```

```
            // Return sum of elements on left
            // and right of mid
            // returning only left_sum + right_sum will fail for
            // [-2, 1]
            return Math.max(left_sum + right_sum,
                            Math.max(left_sum, right_sum));
        }

        // Returns sum of maximum sum subarray
        // in aa[l..h]
        static int maxSubArraySum(int arr[], int l, int h)
        {
            // Base Case: Only one element
            if (l == h)
                return arr[l];

            // Find middle point
            int m = (l + h) / 2;

            /* Return maximum of following three
            possible cases:
            a) Maximum subarray sum in left half
            b) Maximum subarray sum in right half
            c) Maximum subarray sum such that the
            subarray crosses the midpoint */
            return Math.max(
                Math.max(maxSubArraySum(arr, l, m),
                         maxSubArraySum(arr, m + 1, h)),
                maxCrossingSum(arr, l, m, h));
        }

```

Kadane's Algorithm:

Initialize:

max_so_far = INT_MIN

max_ending_here = 0

Loop for each element of the array

(a) max_ending_here = max_ending_here + a[i]

(b) if(max_so_far < max_ending_here)

max_so_far = max_ending_here

(c) if(max_ending_here < 0)

max_ending_here = 0

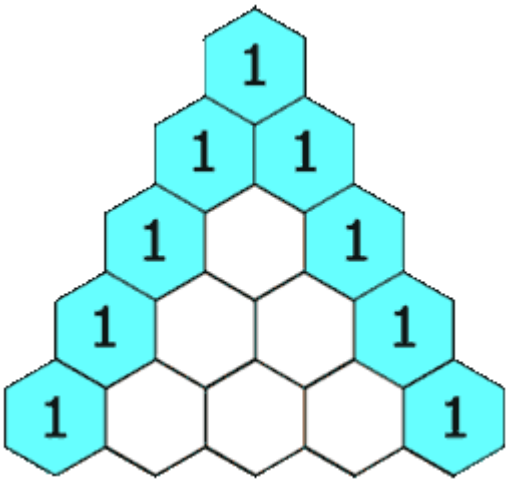
return max_so_far

```
public int maxSubArray(int[] nums) {
        int maxsum=Integer.MIN_VALUE;
        int sum=0;
        int n=nums.length;
        for(int i=0;i<n;i++)
        {
           if(nums[i]>sum+nums[i])
           {
              sum=nums[i];
           }
           else
           {
              sum+=nums[i];
           }
           maxsum=Math.max(maxsum,sum);
        }
        return maxsum;
    }
```

Pascal triangle

Given an integer `numRows`, return the first numRows of **Pascal's triangle**.

In **Pascal's triangle**, each number is the sum of the two numbers directly above it as shown:




```
public class Solution {

public List<List<Integer>> generate(int numRows)

{

        List<List<Integer>> allrows = new ArrayList<List<Integer>>();

        ArrayList<Integer> row = new ArrayList<Integer>();

        for(int i=0;i<numRows;i++)

        {        row.add(0, 1);

                for(int j=1;j<row.size()-1;j++)

                        row.set(j, row.get(j)+row.get(j+1));

                allrows.add(new ArrayList<Integer>(row));

        }

        return allrows;


}
```

}

https://www.geeksforgeeks.org/pascal-triangle/

SET MATRIX ZEROES

The question seems to be pretty simple but the trick here is that we need to modify the given matrix in place i.e. our space complexity needs to $O(1)$.

We will go through two different approaches to the question. The first approach makes use of additional memory while the other does not.

Approach 1: Additional Memory Approach

Intuition

If any cell of the matrix has a zero we can record its row and column number. All the cells of this recorded row and column can be marked zero in the next iteration.

Algorithm

1. We make a pass over our original array and look for zero entries.
2. If we find that an entry at `[i, j]` is 0, then we need to record somewhere the row `i` and column `j`.
3. So, we use two `sets`, one for the rows and one for the columns.

```
4.  if cell[i][j] == 0 {
5.      row_set.add(i)
6.      column_set.add(j)
    }
```

7. Finally, we iterate over the original matrix. For every cell we check if the row `r` or column `c` had been marked earlier. If any of them was marked, we set the value in the cell to 0.

```
8.  if r in row_set or c in column_set {
9.      cell[r][c] = 0
    }
```

Complexity Analysis

- Time Complexity: $O(M \times N)$ where M and N are the number of rows and columns respectively.
- Space Complexity: $O(M+N)$.

Approach 2: O(1) Space, Efficient Solution

Intuition

Rather than using additional variables to keep track of rows and columns to be reset, we use the matrix itself as the *indicators*.

The idea is that we can use the **first cell** of every row and column as a **flag**. This flag would determine whether a row or column has been set to zero. This means for every cell instead of going to $\mathrm{M}+\mathrm{N}M+N$ cells and setting it to zero we just set the flag in two cells.

```
if cell[i][j] == 0 {

    cell[i][0] = 0

    cell[0][j] = 0

}
```

These flags are used later to update the matrix. If the first cell of a row is set to zero this means the row should be marked zero. If the first cell of a column is set to zero this means the column should be marked zero.

Algorithm

1. We iterate over the matrix and we mark the first cell of a row `i` and first cell of a column `j`, if the condition in the pseudo code above is satisfied. i.e. if `cell[i][j] == 0`.

2. The first cell of row and column for the first row and first column is the same i.e. `cell[0][0]`. Hence, we use an additional variable to tell us if the first column had been marked or not and the `cell[0][0]` would be used to tell the same for the first row.

3. Now, we iterate over the original matrix starting from second row and second column i.e. `matrix[1][1]` onwards. For every cell we check if the row `r` or column `c` had been marked earlier by checking the respective first row cell or first column cell. If any of them was marked, we set the value in the cell to 0. Note the first row and first column serve as the `row_set` and `column_set` that we used in the first approach.

4. We then check if `cell[0][0] == 0`, if this is the case, we mark the first row as zero.

5. And finally, we check if the first column was marked, we make all entries in it as zeros.

0	0	0	0
0	0	1	1
0	1	0	0
0	0	0	1

We iterate the matrix we got from the above steps and mark respective cells zeroes.

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Complexity Analysis

- Time Complexity : $O(M \times N)$
- Space Complexity : $O(1)$

```
class Solution {

  public void setZeroes(int[][] matrix) {
```

```
Boolean isCol = false;
int R = matrix.length;
int C = matrix[0].length;

for (int i = 0; i < R; i++) {

  // Since first cell for both first row and first column is the same i.e. matrix[0][0]
  // We can use an additional variable for either the first row/column.
  // For this solution we are using an additional variable for the first column
  // and using matrix[0][0] for the first row.
  if (matrix[i][0] == 0) {
    isCol = true;
  }

  for (int j = 1; j < C; j++) {
    // If an element is zero, we set the first element of the corresponding row and column to 0
    if (matrix[i][j] == 0) {
      matrix[0][j] = 0;
      matrix[i][0] = 0;
    }
  }
}

// Iterate over the array once again and using the first row and first column, update the elements.
for (int i = 1; i < R; i++) {
  for (int j = 1; j < C; j++) {
    if (matrix[i][0] == 0 || matrix[0][j] == 0) {
      matrix[i][j] = 0;
    }
  }
```

```
    }

    // See if the first row needs to be set to zero as well
    if (matrix[0][0] == 0) {
      for (int j = 0; j < C; j++) {
        matrix[0][j] = 0;
      }
    }

    // See if the first column needs to be set to zero as well
    if (isCol) {
      for (int i = 0; i < R; i++) {
        matrix[i][0] = 0;
      }
    }
  }
}
```

Sort Colors

Given an array `nums` with `n` objects colored red, white, or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers `0`, `1`, and `2` to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

Example 1:

```
Input: nums = [2,0,2,1,1,0]

Output: [0,0,1,1,2,2]
```

Example 2:

```
Input: nums = [2,0,1]

Output: [0,1,2]
```

Note: this problem is variation of Dutch national flag problem

```
// two pass O(m+n) space
void sortColors(int A[], int n) {
    int num0 = 0, num1 = 0, num2 = 0;

    for(int i = 0; i < n; i++) {
        if (A[i] == 0) ++num0;
        else if (A[i] == 1) ++num1;
        else if (A[i] == 2) ++num2;
    }

    for(int i = 0; i < num0; ++i) A[i] = 0;
    for(int i = 0; i < num1; ++i) A[num0+i] = 1;
    for(int i = 0; i < num2; ++i) A[num0+num1+i] = 2;
```

```
}
// one pass in place solution=
void sortColors(int A[], int n) {
    int n0 = -1, n1 = -1, n2 = -1;
    for (int i = 0; i < n; ++i) {
        if (A[i] == 0)
        {
            A[++n2] = 2; A[++n1] = 1; A[++n0] = 0;
        }
        else if (A[i] == 1)
        {
            A[++n2] = 2; A[++n1] = 1;
        }
        else if (A[i] == 2)
        {
            A[++n2] = 2;
        }
    }
}


// one pass in place solution
void sortColors(int A[], int n) {
    int j = 0, k = n - 1;
    for (int i = 0; i <= k; ++i){
        if (A[i] == 0 && i != j)
            swap(A[i--], A[j++]);
        else if (A[i] == 2 && i != k)
            swap(A[i--], A[k--]);
```

```
    }
}


// one pass in place solution
void sortColors(int A[], int n) {
    int j = 0, k = n-1;
    for (int i=0; i <= k; i++) {
        if (A[i] == 0)
            swap(A[i], A[j++]);
        else if (A[i] == 2)
            swap(A[i--], A[k--]);
    }
}
```