**Hamming Distance**

# program to calculate hamming distance

'''

The Hamming distance between two integers is the number of positions at which the corresponding bits are different.

Given two integers x and y, calculate the Hamming distance.

Input: x = 1, y = 4

Output: 2

Explanation:

1   (0 0 0 1)

4   (0 1 0 0)

       ↑   ↑

The above arrows point to positions where the corresponding bits are different.

'''

def binary(n):

  return '{0:b}'.format(n).rjust(4,'0')

def hammingDist(num1, num2):

  result = 0

  a = map(int, binary(num1))

  b = map(int, binary(num2))

  for i in range(4):

    #print i

    if a[i] != b[i]:

      result = result + 1

  return result

print hammingDist(4,1)

**Roman Number to Integer**

# convert Roman number to Integer

'''''

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

'''

def romanToInt(x):

  result = 0

  # create a hash table of roman numbers

  roman = {}

  roman['I'] = 1

  roman['IV'] = 4

  roman['V'] = 5

  roman['IX'] = 9

  roman['X'] = 10

  roman['XL'] = 40

  roman['L'] = 50

  roman['XC'] = 90

  roman['C'] = 100

  roman['CD'] = 400

  roman['D'] = 500

  roman['CM'] = 900

  roman['M'] = 1000

  for i in x:

    result = result + roman[i]

  return result

print romanToInt('MM')

**Best Time to Sell Stocks**

1. '''''
2. Say you have an array for which the ith element is the price of a given stock on day i.
3. If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.
4. Example 1:
5. Input: [7, 1, 5, 3, 6, 4]
6. Output: 5
7. max. difference = 6-1 = 5 (not 7-1 = 6, as selling price needs to be larger than buying price)
9. Example 2:
10. Input: [7, 6, 4, 3, 1]
11. Output: 0
12. In this case, no transaction is done, i.e. max profit = 0.
14. '''
15. #Brute Force O(n\*n)
16. **def** max\_profit(arr):
17. buy\_price = 0
18. sell\_price = 0
19. profit = 0
20. **for** i **in** range(len(arr)):
21. buy\_price = arr[i]
22. **print** "buy\_price: " + str(buy\_price)
23. **print** "i:" + str(i)
24. **for** j **in** range(i+1, len(arr)):
25. **print** "j:" + str(j)
26. **if** arr[j] > arr[i]:
27. sell\_price = arr[j]
28. **print** "sell\_price: " + str(sell\_price)
29. **if** profit < (arr[j] - arr[i]):
30. profit = sell\_price - buy\_price
31. **print** "profit" + str(profit)
32. **return** profit
34. arr1 = [7, 6, 4, 3, 1]
35. **print** max\_profit(arr1)
37. ##Another approach O(n)
38. ##assign the buy and sell price initially at start
39. #as we loop, if the next number is less than previous buy price then we buy again again i.e. buy = arr[i]
40. #if we assign buy then we can sell only next day so assign sell price = a[i+1]
41. #also check every time the sell price, if it is more than previous sell price, sell on that day i.e. sell price = arr[i]
42. **def** max\_profit(arr):
43. buy\_price = 0
44. sell\_price = 0
45. profit = 0
46. **for** i **in** range(len(arr)-1):
47. **if** i == 0:
48. buy\_price = arr[i]
49. sell\_price = arr[i]
50. **if** arr[i] < buy\_price **and** i < len(arr):
51. buy\_price = arr[i]
52. sell\_price = arr[i+1]
53. **if** sell\_price < arr[i]:
54. sell\_price = arr[i]
55. **if** profit < sell\_price - buy\_price:
56. profit = sell\_price - buy\_price
58. **return** profit

61. arr1 = [7, 1, 5, 3, 6, 4]
62. **print** max\_profit(arr1)

**Java**

1. **public** **class** **Solution** **{**
2. **public** **int** **maxProfit(int** prices**[])** **{**
3. **int** minprice **=** Integer**.**MAX\_VALUE**;**
4. **int** maxprofit **=** 0**;**
5. **for** **(int** i **=** 0**;** i **<** prices**.**length**;** i**++)** **{**
6. **if** **(**prices**[**i**]** **<** minprice**)**
7. minprice **=** prices**[**i**];**
8. **else** **if** **(**prices**[**i**]** **-** minprice **>** maxprofit**)**
9. maxprofit **=** prices**[**i**]** **-** minprice**;**
10. **}**
11. **return** maxprofit**;**
12. **}**
13. **}**

**Remove Duplicates from Sorted Array**

1. '''''
2. Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length.
3. Do not allocate extra space for another array, you must do this in place with constant memory.
4. For example,
5. Given input array nums = [1,1,2],
6. Your function should return length = 2, with the first two elements of nums being 1 and 2 respectively. It doesn't matter what you leave beyond the new length.
7. '''
9. # first solution is by creating another array
10. arr = [1,2,2,5,6,7,7,8,9]
11. rem = []
12. # copy to the second array only when elements are not duplicate
13. **for** i **in** range(len(arr)):
14. **if** i == 0:
15. rem.append(arr[i])
16. **elif** i > 0 **and** arr[i] != arr[i-1]:
17. rem.append(arr[i])
18. count = 0
19. **print** rem
20. **print** arr
22. # another approach  - not using another array - O(n\*n)
23. arr = [1,2,2,5,6,7,7,8,9]
24. temp = 0
25. dupCount = 0
27. **for** i **in** range(len(arr)):
28. **if** i > 0 **and** arr[i] == arr[i-1]:
29. # we found a duplicate
30. dupCount += 1
31. # loop through rest of the array to change the index by -1
32. **for** j **in** range(i, len(arr)-1):
33. arr[j] = arr[j+1]
35. # trim the array original length minus duplicates
36. arr = arr[0:len(arr)-dupCount+1]
37. **print** arr

40. # another approach  - not using another array - O(n)
41. arr = [1,2,3,5,6,7,7,8,9,9]
42. i = 0
43. nondup = 0
44. **for** j **in** range(len(arr)):
45. **if** arr[i] != arr[j]:
46. # till the time arr[i] = arr[j] then dont increment i
47. # as soon as they are not equal that means there is a new element
48. # so copy the new element
49. i = i + 1
50. nondup = nondup + 1
51. arr[i] = arr[j]
53. arr = arr[0:nondup + 1]
54. **print** arr