Millennium Management

[You are given an integer N. Is there a permutation of digits of integer that's divisible by 8?]

```
def solve(n):
  l=len(str(n))
  if l < 3:
      if int(n) % 8 == 0:
          return 'YES'
      n = n[::-1]
      if int(n) % 8 == 0:
          return 'YES'
      return 'NO'
  hash = 10 * [0]
  for i in range(0, l):
      hash[int(n[i]) - 0] += 1;
  for i in range(104, 1000, 8):
      dup = i
      freq = 10 * [0]
      freq[int(dup % 10)] += 1;
      dup = dup / 10
      freq[int(dup % 10)] += 1;
      dup = dup / 10
      freq[int(dup % 10)] += 1;

      dup = i
      if (freq[int(dup % 10)] > hash[int(dup % 10)]):
          continue;
      dup = dup / 10;

      if (freq[int(dup % 10)] > hash[int(dup % 10)]):
          continue;
      dup = dup / 10

      if (freq[int(dup % 10)] > hash[int(dup % 10)]):
          continue;
      return 'YES'
  return 'NO'
```

Stock Maximize

```
def stockmax(prices):
   m = prices.pop()
   maxsum = 0
   arrsum = 0
   for p in reversed(prices):
       m = max(m, p)
```

```
        maxsum+=m
        arrsum+=p
    return maxsum-arrsum
```

Missing Stock

```
import sys
from scipy.interpolate import UnivariateSpline
import numpy as np
n = int(sys.stdin.readline())
raw_prices = []
for i in range(0, n):
    line = sys.stdin.readline()
    timestamp, price = line.split("\t")
    raw_prices.append(price)
prices_ind = []
missing_prices_ind = []
prices = []
for i in range(0, n):
    if 'Missing' in raw_prices[i]:
        missing_prices_ind.append(i)
    else:
        prices_ind.append(i)
        prices.append(float(raw_prices[i]))
#Splie Interpolation
spline = UnivariateSpline(np.array(prices_ind), np.array(prices), s=2)
for i in missing_prices_ind:
    print(spline(i))
```

Longest Substring

```
def maxNormalSubstring(P, Q, K, N):
    if (K == 0):
        return 0
    # keeps count of normal characters
    count = 0

    # indexes of substring
    left, right = 0, 0

    # maintain length of longest substring
    # with at most K normal characters
    ans = 0

    while (right < N):
        while (right < N and count <= K):

            # get position of character
            pos = ord(P[right]) - ord('a')
```

```
            # check if current character is normal
            if (Q[pos] == '0'):

                # check if normal characters
                # count exceeds K
                if (count + 1 > K):
                    break
                else:
                    count += 1

            right += 1

            # update answer with substring length
            if (count <= K):
                ans = max(ans, right - left)

        while (left < right):

            # get position of character
            pos = ord(P[left]) - ord('a')

            left += 1

            # check if character is
            # normal then decrement count
            if (Q[pos] == '0'):
                count -= 1

            if (count < K):
                break

    return ans
```

Given an integer array A[ ] of size N, the task is to find a subsequence of size B such that the minimum difference between any two of them is maximum and print this largest minimum difference.

```
def can_place(A, n, B, mid):
    count = 1
    last_position = A[0]
    # If a subsequence of size B
    # with min diff = mid is possible
    # return true else false
    for i in range(1, n):
        if (A[i] - last_position >= mid):
            last_position = A[i]
            count = count + 1

            if (count == B):
                return bool(True)
```

```
    return bool(False)
# Function to find the maximum of
# all minimum difference of pairs
# possible among the subsequence
def find_min_difference(A, n, B):
    # Sort the Array
    A.sort()
    # Stores the boundaries
    # of the search space
    s = 0
    e = A[n - 1] - A[0]
    # Store the answer
    ans = 0
    # Binary Search
    while (s <= e):
        mid = (int)((s + e) / 2)
        # If subsequence can be formed
        # with min diff mid and size B
        if (can_place(A, n, B, mid)):
            ans = mid
            # Right half
            s = mid + 1

        else:
            # Left half
            e = mid - 1

    return ans
# Driver code
A = [ 1, 2, 3, 5 ]
n = len(A)
B = 3
min_difference = find_min_difference(A, n, B)
print(min_difference)
```

Given a Matrix arr[][] of size M x N having positive integers and a number K, the task is to find the size of the largest square sub-matrix whose sum of elements <= K.

```
def findMaxMatrixSize(arr, K):
    # N size of rows and M size of cols
    n = len(arr)
    m = len(arr[0])
    # To store the prefix sum of matrix
    sum = [[0 for i in range(m + 1)] for j in range(n + 1)]
    # Create Prefix Sum
    for i in range(n + 1):
        # Traverse each rows
        for j in range(m+1):
            if (i == 0 or j == 0):
```

```
                sum[i][j] = 0
                continue
            # Update the prefix sum
            # till index i x j
            sum[i][j] = arr[i - 1][j - 1] + sum[i - 1][j] + \
                sum[i][j - 1]-sum[i - 1][j - 1]
    # To store the maximum size of
    # matrix with sum <= K
    ans = 0
    # Traverse the sum matrix
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            # Index out of bound
            if (i + ans - 1 > n or j + ans - 1 > m):
                break
            mid = ans
            lo = ans
            # Maximum possible size
            # of matrix
            hi = min(n - i + 1, m - j + 1)
            # Binary Search
            while (lo < hi):
                # Find middle index
                mid = (hi + lo + 1) // 2
                # Check whether sum <= K
                # or not
                # If Yes check for other
                # half of the search
                if (sum[i + mid - 1][j + mid - 1] +
                    sum[i - 1][j - 1] -
                    sum[i + mid - 1][j - 1] -
                        sum[i - 1][j + mid - 1] <= K):
                    lo = mid
                # Else check it in first
                # half
                else:
                    hi = mid - 1
            # Update the maximum size matrix
            ans = max(ans, lo)
    # Print the final answer
    print(ans)
```