筛素数：

```python
import math

def sieve(n):
    sieve_list = [False, False] + [True] * (n - 1)
    for i in range (2, int(n**0.5)):
        if sieve_list[i]:
            for j in range(i*i, n+1, i):
                sieve_list[j] = False
    return sieve_list


sieve_list = sieve(1000000)
ignored = input()
nums = map(int, input().split())
for i in nums:
    print('YES' if math.sqrt(i) == int(math.sqrt(i)) and sieve_list[int(math.sqrt(i))] else 'NO')
```

调度场：

```python
prec = {'+':1, '-':1, '*':2, '/':2}

for _ in range(int(input())):
    infix = input()
    s = []
    re = []
    num = ''

    for char in infix:
        if char.isnumeric() or char == '.':
            num += char
        else:
            if num:
                re.append(num)
                num = ''
            if char == '(':
                s.append(char)
            if char == ')':
                while s and s[-1] != '(':
                    re.append(s.pop())
                s.pop()
            if char in '+-*/':
                while s and s[-1] in '+-*/' and prec[char] <= prec[s[-1]]:
                    re.append(s.pop())
                s.append(char)

    if num:
        re.append(num)
    while s:
        re.append(s.pop())
    print(' '.join(str(x) for x in re))
```

Merge sort:

```python
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr)//2

        L = arr[:mid]   # Dividing the array e
        R = arr[mid:] # Into 2 halves

        mergeSort(L) # Sorting the first half
        mergeSort(R) # Sorting the second half

        i = j = k = 0
        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

查克拉：

```python
M, N, T = map(int, input().split())
graph = [list(input()) for i in range(M)]
direc = [(0,1), (1,0), (-1,0), (0,-1)]
start, end = None, None
for i in range(M):
    for j in range(N):
        if graph[i][j] == '@':
            start = (i, j)
def bfs():
    q = deque([start + (T, 0)])
    visited = [[-1]*N for i in range(M)]
    visited[start[0]][start[1]] = T
    while q:
        x, y, t, time = q.popleft()
        time += 1
        for dx, dy in direc:
            if 0<=x+dx<M and 0<=y+dy<N:
                if (elem := graph[x+dx][y+dy]) == '*' and t > visited[x+dx][y+dy]:
                    visited[x+dx][y+dy] = t
                    q.append((x+dx, y+dy, t, time))
                elif elem == '#' and t > 0 and t-1 > visited[x+dx][y+dy]:
                    visited[x+dx][y+dy] = t-1
                    q.append((x+dx, y+dy, t-1, time))
                elif elem == '+':
                    return time
```

合法出栈序列：

```
chars = input()
while True:
    try:
        wait = input()
        if len(wait) != len(chars):
            print('NO')
            continue
        c_r = chars[:]  # chars_rema
        fixed = ''
        while wait:
            f_l = wait[0]  # first_l
            if f_l not in chars:
                print('NO')
                break
            elif f_l not in fixed:
                f_i = c_r.index(f_l)
                fixed += c_r[:f_i]
                c_r = c_r[f_i+1:]
            elif f_l == fixed[-1]:
                fixed = fixed[:-1]
            else:
                print('NO')
                break
            wait = wait[1:]
        if not wait:
            print('YES')
    except EOFError:
        break
```

Huffman Tree:

```
import heapq

class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq

def huffman_encoding(char_freq):
    heap = [Node(char, freq) for char, freq in char_freq.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq) # note: 合并之
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]
```

单调栈：

4.单调栈：以O(n)的效率处理序列
括找到某个数据点以前或以后大于
点。下面为一递减栈：

```
for i in range(n):
  while stack and a[stack[-1]] < a[i]:
    a[stack.pop()] = i + 1
  stack.append(i)
#在原数据上改可以节省内存
while stack:
  a[stack.pop()] = 0
```

Dirs and files:

```
class dir:
    def __init__(self, dname):
        self.name = dname
        self.dirs = []
        self.files = []

    def getGraph(self):
        g = [self.name]
        for d in self.dirs:
            subg = d.getGraph()
            g.extend(["|     " + s for s in subg])
        for f in sorted(self.files):
            g.append(f)
        return g
```