

```

#customer penalty
from typing import List
def compute_penalty(log: str, close: int) -> int:
    penalty = 0
    for i in range(len(log)):
        if i < close and log[i] == 'N':
            penalty += 1
        elif i >= close and log[i] == 'Y':
            penalty += 1
    return penalty

def get_closing_with_min_penalty(log: str) -> int:
    curr = 0
    best = 0      # best penalty value
    time = 0      # best closing time
    n = len(log)
    for i in range(n):
        c = log[i]
        if c == 'Y':
            curr -= 1
        else:      # 'N'
            curr += 1
        if curr < best:
            best = curr
            time = i + 1
    return time

def get_all_closing(log: str) -> List[int]:
    tokens = log.split()
    stack: List[List[str]] = []
    results: List[int] = []
    for tok in tokens:
        if tok == "B":
            stack.append([])
        elif tok == "E":
            if stack:
                s = ''.join(stack.pop())
                results.append(get_closing_with_min_penalty(s))
            else:
                # append Y/N into current top block
                if stack:
                    stack[-1].append(tok)
        return results
    import sys
    t = int(sys.stdin.readline())
    for _ in range(t):
        log = sys.stdin.readline().strip()
        if " " in log:
            print(*get_all_closing(log))
        else:
            print(get_closing_with_min_penalty(log))

# ----- UTIL -----
def exists_in_range(lst, lo, hi):
    i = bisect.bisect_left(lst, lo)
    return i < len(lst) and lst[i] <= hi

# ----- QUERY -----
def search_query(query: str, k: int, index):
    words = query.split()
    first = words[0]
    result = []
    if first not in index:
        return result
    anchors = index[first]
    for pos in anchors:
        ok = True
        for w in words[1:]:
            if w not in index or not exists_in_range(index[w], pos, pos + k):
                ok = False
                break
        if ok:
            result.append(pos)
    return result

# ----- DEMO / DRIVER -----
if __name__ == "__main__":
    text = "The quick brown fox is quick something something quick fox"
    words, idx = preprocess(text)
    print("Text:", text)
    print()
    queries = [
        ("quick fox", 2),
        ("brown fox", 1),
        ("quick something", 3),
        ("notpresent fox", 2),
    ]
    for q, k in queries:
        print(f"Query: '{q}', k={k} ->", search_query(q, k, idx))

#part 1 http
def parse_accept_language(header: str, supported):
    supported = set(supported)  # fast lookup
    result = []
    for lang in header.split(","):
        tag = lang.strip()
        if tag in supported:
            result.append(tag)
    return result

#part2 http
def parse_accept_language(header: str, supported):
    supported = list(supported)
    prefs = []      # store parsed header: (tag, q, order)
    order = 0
    for token in header.split(","):
        token = token.strip()
        if not token:
            continue
        tag, q = token.split(";")
        q = float(q[1:-1])
        prefs.append((tag, q, order))
        order += 1
    return prefs

```

```

continue

if ";q=" in token:
    tag, q = token.split(";q=", 1)
    q = float(q)
else:
    tag, q = token, 1.0

prefs.append((tag.strip(), q, order))
order += 1
result = []
used = set()
# helper to evaluate each preference entry
def consider(tag, q, ord_index):
    if q == 0:
        return
    for lang in supported:
        if lang in used:
            continue
        if (
            tag == "*" or
            tag == lang or
            (len(tag) == 2 and lang.startswith(tag + "-"))
        ):
            used.add(lang)
            result.append((q, ord_index, lang))

for tag, q, ord_index in prefs:
    consider(tag, q, ord_index)
# sort: highest q first, earlier header order first

token = ""
while i < len(s) and s[i] != '{':
    token += s[i]
    i += 1
groups.append([token])

return [".".join(p) for p in product(*groups)]
```

```

class Solution:

    def calcEquation(self, equations: List[List[str]], values: List[float], queries: List[List[str]]) -> List[float]:
        hashmap = {}
        for i in range(len(equations)):
            x, y = equations[i]
            if x not in hashmap:
                hashmap[x] = {}
            hashmap[x][y] = values[i]
            if y not in hashmap:
                hashmap[y] = {}
            hashmap[y][x] = 1 / values[i]

        def bfs(query):
            q = deque()
            visited = set()
            val = 1
            startVar, endVar = query
            q.append((startVar, val))
            while q:
                currentVar, currentVal = q.popleft()
                if currentVar not in hashmap:
                    return -1
                if currentVar == endVar:
                    return currentVal
                for neighbor in hashmap[currentVar]:
                    if neighbor not in visited:
                        visited.add(neighbor)
                        q.append((neighbor, currentVal * hashmap[currentVar][neighbor]))
```

```

result.sort(key=lambda x: (-x[0], x[1]))
return [lang for _, _, lang in result]
```

```

from itertools import product

def brace_expansion(s: str):
    groups = []
    i = 0

    while i < len(s):
        if s[i] == '{':
            j = i + 1
            temp = []
            # read until we hit '}'
            while s[j] != '}':
                if s[j] == ',':
                    temp.append(token)
                    token = ""
                else:
                    token += s[j]
                j += 1
            temp.append(token) # last token
            groups.append(sorted(temp)) # sort alternatives
            i = j + 1
        else:
            # collect consecutive plain characters as one token
            token += s[i]
            i += 1
```

```

elif currentVar == endVar:
    return currentValue
else:
    visited.add(currentVar)
    for nextVar in hashmap[currentVar].keys():
        if nextVar not in visited:
            q.append((nextVar, currentValue * hashmap[currentVar][nextVar]))
return -1
```

```

output = []
for query in queries:
    output.append(bfs(query))
return output
```

```

class Logger:

    def __init__(self):
        self.last = {} # message -> last timestamp printed

    def shouldPrintMessage(self, timestamp: int, message: str) -> bool:
        if message not in self.last:
            self.last[message] = timestamp
            return True
        if timestamp - self.last[message] >= 10:
            self.last[message] = timestamp
            return True
        return False
```

```

class Solution:
    def merge(self, intervals):
        # sort intervals by start time
        intervals.sort(key=lambda x: x[0])

        merged = []

        for interval in intervals:
            # if merged is empty OR no overlap, append
            if not merged or merged[-1][1] < interval[0]:
                merged.append(interval)
            else:
                # overlap -> merge by updating the end
                merged[-1][1] = max(merged[-1][1], interval[1])

        return merged

```

```

from collections import Counter

class Solution:
    def topKFrequent(self, nums, k):
        count = Counter(nums)
        buckets = [[] for _ in range(len(nums) + 1)]

        for num, c in count.items():
            buckets[c].append(num)

        result = []
        for freq in range(len(nums), -1, -1):
            for num in buckets[freq]:

```

```

balance[to] += amt

# keep only non-zero balances
debts = [b for b in balance.values() if b != 0]

def dfs(start):
    # skip settled debts
    while start < len(debts) and debts[start] == 0:
        start += 1

    if start == len(debts):
        return 0

    ans = float('inf')

    for i in range(start + 1, len(debts)):
        # only try pairing opposite signs
        if debts[start] * debts[i] < 0:
            debts[i] += debts[start]
            ans = min(ans, 1 + dfs(start + 1))
            debts[i] -= debts[start]

    # optimization: stop early if perfect match
    if debts[i] + debts[start] == 0:
        break

    return ans

```

```

return dfs(0)

```

```

result.append(num)
if len(result) == k:
    return result

```

```

class Solution:
    def findCheapestPrice(self, n, flights, src, dst, k):
        INF = float("inf")
        dist = [INF] * n
        dist[src] = 0

        # We allow at most k stops => at most k+1 edges
        for _ in range(k + 1):
            new_dist = dist[:]

            for u, v, w in flights:
                if dist[u] != INF and dist[u] + w < new_dist[v]:
                    new_dist[v] = dist[u] + w

            dist = new_dist

```

```

return -1 if dist[dst] == INF else dist[dst]

```

```

from collections import defaultdict

class Solution:
    def minTransfers(self, transactions):
        balance = defaultdict(int)

        # compute net balance
        for frm, to, amt in transactions:
            balance[frm] -= amt

```