A Интеллектуальная снегоуборка

Задачу проверки соответствия строки заданному формату можно решить «в лоб», выполнив цепочку последовательных обрезаний, разрезаний и фильтраций исходной строки с последующей проверкой допустимости значений отдельных фрагментов.

Более изящная альтернатива — использовать регулярные выражения:

```
from re import compile

pattern = compile(
    "^([1-6]T[AB]X|7T[ABE]X) "
    "([1-9]\\d{3}|0[1-9]\\d{2}|00[1-9]\\d|000[1-9])$")

n = int(input())

print(sum(1 for _ in range(n) if pattern.match(input())))
```

B Умная парковка

Задача тривиально решается простым перебором по всем допустимым уровням рейтинга a_i с обновлением минимального расстояния, однако такое решение имеет временну́ю сложность $O(mn)$ и проходит лишь на частичный балл.

Для эффективного решения задачи целесообразно предварительно отсортировать допустимые уровни рейтинга a_i по возрастанию и находить ближайшее к ИРА значение a_i с помощью бинарного поиска.

Алгоритм бинпоиска можно реализовать самостоятельно, но удобнее воспользоваться готовой реализацией из стандартной библиотеки Python:

```
from bisect import bisect_left
```

При реализации нужно аккуратно рассмотреть краевые случаи:

```
i = bisect_left(data, x)
if i == len(a):
    print(a[i - 1])
elif i == 0:
    print(a[i])
else:
    if abs(a[i] - x) <= abs(a[i - 1] - x):
        print(a[i])
    else:
        print(a[i - 1])
```

C Эффективные уведомления

Для решения задачи необходимо научиться эффективно определять количество водителей, находившихся в состоянии активного вождения в момент отправки каждого из уведомлений.

Будем регистрировать события трёх типов:

- `(a[i], -1, i)` : в момент $a[i]$ начался i-й период активного вождения;
- `(c[k], 0, k)` : в момент $c[k]$ отправлено k–е уведомление;
- `(b[i], 1, i)` : в момент $b[i]$ закончился i-й период активного вождения.

Для того, чтобы найти количество водителей, получивших k-е уведомление, достаточно знать, сколько событий начала вождения `(a[i], -1, i)` предшествуют событию отправки уведомления `(c[k], 0, k)` при условии, что соответствующее событие окончания `(b[i], 1, i)` наступило не ранее момента отправки.

Упорядочим события по возрастанию и будем последовательно просматривать их, увеличивая на единицу счётчик количества водителей, находящихся в состоянии активного вождения, всякий раз, когда наблюдается событие начала, и, соответственно, уменьшая его на единицу при наступлении события окончания вождения. Тогда количество водителей, находившихся в состоянии активного вождения в момент отправки соответствующего уведомления, будет равно текущему значению счётчика.

Приведём ключевой фрагмент реализации вышеописанного алгоритма:

```
events.sort()
answer = [0] * m
count = 0
for event in events:
    _, event_type, event_id = event
    count -= event_type
    if event_type == 0:
        answer[event_id] = count
```

D Оптимальный экспресс

Для начала заметим, что можно перебрать всевозможные способы выбора остановок для экспресса и выбрать из них оптимальный вариант. Сложность данного решения по времени составляет $O(2^n)$, поэтому такие решения получают только частичный балл.

Более рациональной идеей является использование динамического программирования для вычисления величины $dp[i][j]$ — максимального количества пассажиров с остановок с номерами, не превышающими i, которые поедут на экспрессе, если он в сумме сделает j остановок. Возможны два варианта: экспресс может как пропустить остановку i, так и остановиться на ней.

Если экспресс принимает решение пропустить i-ю остановку, максимальное количество перевозимых им к этому моменту пассажиров не изменится, т.е. $dp[i][j]$ будет равно $dp[i-1][j]$. Если же экспресс останавливается на i-й остановке, то значение $dp[i][j]$ будет равно $dp[i-1][j-1] + c_i$, если экспресс с учётом всех j остановок доезжает до конечного пункта быстрее регулярного рейса, и $dp[i-1][j-1]$ в противном случае.

Данный алгоритм реализуется следующим образом:

```
dp[0][0] = 0;
for i in range(n):
    for j in range(i + 1):
        if dp[i][j] >= 0:
            if e[i] > p[i] + j + 1:
                dp[i + 1][j + 1] = max(dp[i + 1][j + 1], dp[i][j] + c[i])
            dp[i + 1][j] = max(dp[i + 1][j], dp[i][j])

answer = 0
for i in range(0, n + 1):
    answer = max(answer, dp[n][i])
```

Временнàя сложность полученного решения составляет $O(n^2)$, оно проходит все тесты.