Быстрые сортировки. Поиски.

1 A. Count me in

Разбор

Обратим внимание, что значения всех элементов массива находятся в диапазоне от 1 до 10. Тогда воспользуемся сортировкой подсчетом – будем подсчитывать количество единиц, двоек и т.д., а затем выведем в порядке возрастания каждый ключ нужное количество раз.

Асимптотика решения: $O(N + M)$

Решение

```
x = list(map(int, input().split()))
min_e, max_e = min(x), max(x)
cnt = [0] * (max_e - min_e + 1)
for v in x:
    cnt[v - min_e] += 1
k = 0
for i in range(len(cnt)):
    for c in range(cnt[i]):
        x[k] = i + min_e
        k += 1
print(' '.join(map(str, x)))
```

2 B. Инверсии

Разбор

Будем подсчитывать число инверсий в процессе сортировки слиянием. Когда мы соединяем два отсортированных списка, мы для каждого элемента знаем, сколько элементов больше него находятся в левой части списка. Просто посчитаем на каждом шаге слияния число инверсий и передадим наверх сумму.

Асимптотика решения: $O(N \times \log N)$

Решение

```
def merge_sort(a, l, r):
    if r - l < 2:
        return 0
    m = (r + l) // 2
    l_inv = merge_sort(a, l, m)
    r_inv = merge_sort(a, m, r)
    return l_inv + r_inv + merge(a, l, m, r)

def merge(a, l, m, r):
    l1, r1 = l, m
    l2, r2 = m, r
    result = []
    inv = 0
    while l1 < r1 and l2 < r2:
        if a[l1] < a[l2]:
            result.append(a[l1])
            l1 += 1
            inv += l2 - m
        else:
            result.append(a[l2])
            l2 += 1
    while l1 < r1:
```

```
        result.append(a[l1])
        l1 += 1
        inv += l2 - m
    while l2 < r2:
        result.append(a[l2])
        l2 += 1
    for i in range(l, r):
        a[i] = result[i - l]
    return inv

with open('inverse.in', 'r') as f:
    n = int(f.readline())
    x = list(map(int, f.readline().split()))
ans = merge_sort(x, 0, n)
with open('inverse.out', 'w') as f:
    print(ans, file=f)
```

3 C. Мегаинверсии

Разбор

В чём заключается ключевое отличие от предыдущей задачи? Теперь мы работаем с тройками вместо пар. Посмотрим на тройки – в них нас больше всего интересует средний элемент, т.к. левый и правый элементы не обязательно гарантируют наличие мега-инверсии.

Теперь воспользуемся идеей подсчёта обычных инверсий два раза – сначала найдём число раз, которое элемент a_i является **левой** инверсией (т.е. больше какого-то элемента, стоящего правее), а затем аналогично сколько раз этот же элемент является **правой** инверсией (т.е. меньше какого-то элемента, стоящего левее). Теперь, используя комбинаторные правила сложения и умножения, мы можем найти конечный ответ.

Асимптотика решения: $O(N \times \log N)$

Решение

```
with open('mega.in', 'r') as f:
    n = int(f.readline())
    x = list(map(int, f.readlines()))
INV_S = [0] * (n + 1)
INV_R = [0] * (n + 1)

def merge_sort(a, l, r, reverse=False):
    if r - l < 2:
        return 0
    m = (r + l) // 2
    merge_sort(a, l, m, reverse)
    merge_sort(a, m, r, reverse)
    merge(a, l, m, r, reverse)

def merge(a, l, m, r, reverse=False):
    l1, r1 = l, m
    l2, r2 = m, r
    result = []
    while l1 < r1 and l2 < r2:
        if a[l1] < a[l2]:
            if not reverse:
                result.append(a[l1])
                l1 += 1
                INV_S[a[l1 - 1]] += l2 - m
            else:
                result.append(a[l2])
                l2 += 1
                INV_R[a[l2 - 1]] += l1 - l
        else:
            if not reverse:
                result.append(a[l2])
```

```
                l2 += 1
            else:
                result.append(a[l1])
                l1 += 1
    while l1 < r1:
        result.append(a[l1])
        l1 += 1
        if not reverse:
            INV_S[a[l1 - 1]] += l2 - m
    while l2 < r2:
        result.append(a[l2])
        l2 += 1
        if reverse:
            INV_R[a[l2 - 1]] += l1 - l
    for i in range(l, r):
        a[i] = result[i - l]

merge_sort(x.copy(), 0, n, False)
merge_sort(x.copy(), 0, n, True)
res = sum([INV_S[i] * INV_R[i] for i in range(1, n + 1)])
with open('mega.out', 'w') as f:
    print(res, file=f)
```

4 D. Минимизируем максимум

Разбор

Рассмотрим два массива одинаковой длины, один из которых убывает, а второй возрастает. От нас требуется найти такую точку, где максимум из двух элементов, стоящих на одной позиции, будет минимальным. Если мы перейдём от дискретных величин к непрерывным, то этой точкой будет являться пересечение прямых. В случае дискретных величин нас интересует точка, в которой значение одного из элементов становится больше другого.

Такая задача замечательно сводится к идее бинпоиска – представим в виде нулей точки, в которых $a_i < b_i$, а в виде единичек – точки, в которых $a_i > b_i$. Когда наши указатели сойдутся, просто проверим обоих кандидатов и выберем оптимальный ответ.

Асимптотика решения: $O(\log L)$ на запрос

Обратите внимание, что для сдачи этой задачи на языке Python необходимо использовать оптимизированный ввод и вывод.

Решение

```
import io, os, sys

def read_all():
    bytes = io.BytesIO(os.read(0,os.fstat(0).st_size)).readlines()
    return list(map(lambda x: x.decode(), bytes))

def write_all(v):
    sys.stdout.write('\n'.join(v) + '\n')

all_it = iter(read_all())
n, m, l = map(int, next(all_it).split())
A = [list(map(int, next(all_it).split())) for _ in range(n)]
B = [list(map(int, next(all_it).split())) for _ in range(m)]

def solve(x, y):
    left, right = -1, l
    while right - left > 1:
        mid = (right + left) // 2
        if x[mid] < y[mid]:
            left = mid
        else:
            right = mid
    if right == l or left >= 0 and max(x[left], y[left]) < max(x[right], y[right]):
```

```
        return left
    return right

q = int(next(all_it))
out = [''] * q
for idx in range(q):
    i, j = map(int, next(all_it).split())
    a, b = A[i - 1], B[j - 1]
    out[idx] = str(solve(a, b) + 1)
write_all(out)
```

5 E. Дремучий лес

Разбор

Для начала посмотрим на то, как себя ведёт функция времени комиссии в пути в зависимости от выбора точки входа в лес.

Можно формально показать, что такая функция будет иметь единственный минимум на отрезке $[0; 1]$. Из-за этого мы можем искать её минимум как бинпоиском по производной, так и тернарным поиском. Но нам лень считать производную, поэтому сразу воспользуемся тернарным поиском.

Асимптотика решения: $O(\log \frac{r-l}{\epsilon})$

Решение

```
def f(x, y, v1, v2):
    s_to_entrance_dist = (pow(x, 2) + pow(1 - y, 2)) ** 0.5
    entrance_to_b_dist = (pow(1 - x, 2) + pow(y, 2)) ** 0.5
    return s_to_entrance_dist / v1 + entrance_to_b_dist / v2

vp, vf = map(int, input().split())
a = float(input())

eps = 1E-8
l, r = 0, 1
while abs(r - l) > eps:
    m1 = (2 * l + r) / 3
    m2 = (l + 2 * r) / 3
    y_m1 = f(m1, a, vp, vf)
    y_m2 = f(m2, a, vp, vf)
    if y_m1 < y_m2:
        r = m2
    else:
        l = m1
print('{:.7f}'.format((l + r) / 2))
```

6 F. Поиск позиции

Разбор

При первом взгляде на эту задачу может показаться, что её надо решать бинпоиском (возможно даже двумя). И действительно, нам надо максимизировать какое-то произведение. Давайте подумаем, что мы можем использовать для бинпоиска?

Попробуем искать бинпоиском индекс места, куда нам нужно встать. Но вспомним основное требование бинпоиска: наша последовательность должна быть представима в виде ноликов и единичек, а наша задача - отыскать единственную точку, где заканчиваются нолики и начинаются единички. Однако данная задача

не удовлетворяет этому требованию – где-то есть единственная оптимальная точка, которую нам надо найти.

Может быть, надо применить тернарный поиск? Но из-за того что количество и расположение элементов с нашим ростом нам заранее неизвестно, мы не можем гарантировать, что это будет унимодальная функция.

Посмотрим внимательно – ведь мы можем заранее предпосчитать всё, что нам нужно знать, чтобы ответить на вопрос! Сначала для каждого значения в массиве запомним, в каких позициях он встречается, а затем для всех позиций найдём число элементов с таким же значением слева и число элементов с отличающимся значением справа и запомним лучший ответ для заданного значения.

В итоге для каждого нашего предположения о росте мы можем ответить на запрос за константу времени.

Асимптотика решения: $O(N + M)$

Решение

```
n, m = map(int, input().split())
a = list(map(int, input().split()))
x = list(map(int, input().split()))

a2pos = {}
for i in range(n):
    v = a[i]
    if not v in a2pos:
        a2pos[v] = []
    a2pos[v].append(i + 1)

a2max = {}
for v in a2pos:
    v_pos = a2pos[v]
    max_f = 0
    for i in range(len(v_pos)):
        l_weight = i + 1
        r_weight = n - v_pos[i] - (len(v_pos) - (i + 1))
        max_f = max(max_f, l_weight * r_weight)
    a2max[v] = max_f

ans = [a2max.get(v, 0) for v in x]
print(*ans)
```

7 G. Oracle соло мид

Разбор

Будем использовать схематичные изображения всего происходящего в таком виде, что каждое изменение здоровья будет отображено в виде клетки, каждая строка будет обозначать одно применение заклинания, и каждый столбец будет обозначать секунду, в которую что-то происходит.

Во-первых, если $a > b \cdot c$, то ответ -1. В самом деле, после t секунд суммарный нанесённый урон равен $(a - bc)$ для всех заклинаний, действие которых уже прошло, а также ещё какой-то урон от заклинаний, которые ещё действуют. Первое слагаемое может быть сколь угодно большим, а второе ограничено снизу, например, числом $-bc^2$, поскольку максимум c заклинаний ещё не истекли, и каждое из них вылечило врага максимум на b единиц здоровья в секунду на протяжении максимум c секунд. Таким образом, нанесённый урон может быть сколь угодно велик.

С другой стороны, если $a \leq bc$, то ответ всегда существует. В самом деле, имеет смысл смотреть только на моменты времени, делящиеся на d, т. е. в точности на моменты, когда мы наносили урон. Очевидно, что для любого другого t в момент времени t у врага было не меньше здоровья, чем за секунду до этого. Также если $t \geq c$, то у врага не меньше здоровья, чем в момент времени $t - d$. В самом деле, разница в этих уровнях здоровья лишь в одном заклинании, которое целиком истекло, а это неотрицательное число:

РΘРъРуР«СЃРчСЂРёСЌ Рч СЃСЂСЃСЂРэСЂСЋСЃСЃСЌ РуРѳР,,Р,,СКСЖ РуРъCS

Теперь, когда мы знаем, что нужно рассматривать лишь $t < c$, мы точно знаем, что ответ существует. Поймём теперь, когда в общем случае следует вычитать d из t, чтобы уменьшить здоровье врага. Из таких же соображений следует, что если $t < c$, то при уменьшении t на d здоровье врага увеличивается на $a - tb$, а поскольку $t = dk$ для некоторого целого k, мы увеличиваем здоровье на $a - bdk$. Очевидно, это следует делать, пока $a - bdk < 0$:

□РѲРњРуР«СЃРчСЂРёСК Рч СҒСЂСЃСЂРэСЂСЂСЃСК РуРѳР,,Р,,СКСЖ РуРњС

Иными словами, задача звучит теперь следующим образом: найти наибольшее такое k, что $a \geq bdk$, и применить заклинание $(k + 1)$ раз. У врага будет наименьший уровень здоровья как раз как только мы применим заклинание в $(k + 1)$-й раз. Ответ, таким образом, равен $a(k + 1) - \frac{k(k+1)}{2}bd$. Время работы составляет $O(1)$ на тест.

Также можно было заметить, что здоровье врага — выпуклая функция от времени, и найти ответ тернарным поиском. Это занимает $O(\log maxanswer)$ времени на тест, что всё ещё достаточно быстро.

Асимптотика решения: $O(\log maxanswer)$ либо $O(1)$ на запрос

Обратите внимание, что решение с тернарным поиском не проходит ограничение по времени при реализации на языке Python.

Решение

```
t = int(input())
for _ in range(t):
    a, b, c, d = map(int, input().split())
    if b * c < a:
        print(-1)
    else:
        x = a // (b * d)
        s_cnt = (x * (x + 1)) // 2
        res = a * (x + 1) - s_cnt * b * d
        print(res)
```

8 H. When democracy fails

Разбор

“Я на выборы никогда не ходил, но в этот раз точно пойду“

Эта задача состоит из трёх основных этапов:

1. Предподсчёт – отсортируем последовательность партий для бинпоиска по ним и посчитаем префиксные суммы для быстрого ответа на запрос к отрезку
2. Поиск минимальной взятки – для каждой партии посчитаем, какое минимальное количество денег мы должны потратить, чтобы вывести заданную партию в победители
3. Обновление голосов – зная победителя, обновим голоса для всех партий, которые могут нам помешать выиграть выборы

Подробнее остановимся на втором и третьем пунктах.

Как мы ищем минимальную взятку? Предположим, что мы готовы потратить x денег на то, чтобы вывести нашу партию, за которую отдают v_i голосов, в лидеры. Сначала найдём бинпоиском индекс k, с которого начинаются конкуренты – те партии, у которых голосов больше чем $v_i + x$. Затем, хватит ли нам x голосов – сумма голосов за **все** партии-конкуренты за вычетом **желаемого** количества голосов $v_i + x - 1$ должна быть меньше либо равна x:

$$\sum_{j=k,k>i}^{N} v_j - (n - k) \times (v_i + x - 1) \leq x$$

Осталось обновить голоса за наши партии. Сначала обновим число голосов лидера – ему мы добавим то число голосов, которые мы нашли ранее:

$$v_{best} := v_{best} + v_{min}$$

Затем всем конкурентам мы присвоим число $v_{best} - 1$, при этом каждый раз обновляя

$$v_{min} := v_{min} - (v_i - v_{best} + 1)$$

А затем, если у нас остались в запасе голоса v_{min}, просто вычтем их из тех лидеров, которых мы решили "утопить".

Асимптотика решения: $O(N \times \log V \times \log N)$

Бонус: существует решение за $O(N \log N)$, проходящее тесты на языке Python. Решение, приведённое выше, при реализации на Python не проходит по лимиту времени.

Решение

```
class Cons:
    def __init__(self, v, b, idx):
        self.v = v
        self.b = b
        self.idx = idx

    @staticmethod
    def read(idx) -> 'Cons':
        v, b = map(int, input().split())
        return Cons(v, b, idx)

def lower_bound(v):
    l, r = -1, len(a)
    while r - l > 1:
        m = (r + l) // 2
        if a[m].v >= v:
            r = m
        else:
            l = m
    return r

n = int(input())
a = [Cons.read(i) for i in range(n)]
a.sort(key=lambda c: c.v)

max_b = max(map(lambda c: c.b, a))
pref = [0] * (n + 1)
for i in range(n - 1, -1, -1):
    pref[i] = pref[i + 1] + a[i].v

min_b = pref[0] + max_b + 1
best_i = -1
best_lower = -1
for i in range(n):
    if a[i].b < 0 or a[i].b >= min_b:
        continue
    l, r = a[i].v, pref[0] + max_b + 1
    res = -1
    while l <= r:
        m = (r + l) // 2
        idx = lower_bound(m)
        if idx < n and m == a[i].v:
            idx += 1
        diff_v_i = m - a[i].v
        diff_v_prefix = pref[idx] - (n - idx) * (m - 1)
        if idx == n or diff_v_i >= diff_v_prefix:
            r = m - 1
            res = m
        else:
            l = m + 1
```

```
    b = res - a[i].v + a[i].b
    if b < min_b:
        best_i = i
        min_b = b

votes = min_b - a[best_i].b
idx = lower_bound(a[best_i].v + votes)
a[best_i].v += votes
for i in range(idx, n):
    if i == best_i:
        continue
    v_upd = a[i].v - a[best_i].v + 1
    a[i].v = a[best_i].v - 1
    votes -= v_upd
i = n
while votes > 0:
    i -= 1
    if i == best_i:
        continue
    if a[i].v > votes:
        a[i].v -= votes
        votes = 0
    else:
        votes -= a[i].v
        a[i].v = 0

print(min_b)
print(a[best_i].idx + 1)
print(' '.join(map(lambda x: str(x.v), sorted(a, key=lambda c: c.idx))))
```