Разбор задач

Задача 1. Большой квадрат

Рассмотрим разные способы разместить квадрат внутри двух прямоугольников. Если квадрат полностью размещён внутри первого прямоугольника, то максимально возможная длина стороны равна наименьшей стороне прямоугольника $\min(a, b)$. Если он полностью размещён внутри второго прямоугольника, то $\min(c, d)$. Наконец, рассмотрим вариант, когда для вырезания квадрата понадобятся оба прямоугольника, как на картинке в условии. Пусть прямоугольники сложены так, что стороны a и c являются продолжением друг друга, а стороны b и d касаются. Тогда длина стороны квадрата не может быть больше значений $a+c$, b и d, и ответом будет $\min(a+c, b, d)$. Очевидно, что в этом случае нужно выбрать стороны так, чтобы a была наименьшей стороной первого прямоугольника, то есть $a \leqslant b$. Аналогично, c должна быть наименьшей стороной второго прямоугольника, то есть $c \leqslant d$.

Итак, если упорядочить стороны прямоугольников, то есть сделать так, что $a \leqslant b$ и $c \leqslant d$, то ответ равен $\max(a, c, \min(a + c, b, d))$. Пример такого решения.

```
a = int(input())
b = int(input())
c = int(input())
d = int(input())
if a > b:
    a, b = b, a
if c > d:
    c, d = d, c
print(max(a, c, min(a + c, b, d)))
```

Задача 2. План эвакуации

Если комната имеет координаты (r, c), то расстояние до лестницы с координатами (r_i, c_i) равно $|r - r_i| + |c - c_i|$ (так называемое «манхэттенское расстояние»). Посчитаем минимум расстояний от комнаты до двух лестниц. Из комнаты нужно перейти в ту из четырёх соседних комнат, для которой минимум расстояний до лестниц будет меньше, чем в этой комнате. Именно в эту комнату и направим стрелку из текущей комнаты. Задача имеет только реализационную трудность.

В примере решения ниже используются вспомогательные функции. dist возвращает расстояние между двумя комнатами, а dist_to_exit — расстояние от комнаты до ближайшей лестницы. Вложенными циклами проходим по всем комнатам. Для перебора направлений переходов в соседние комнаты удобно использовать цикл, в котором переменная c — это символ, соответствующий направлению перемещения, а переменные dx и dy — это значение изменения координат при переходе в данном направлении.

```
n = int(input())
m = int(input())
y1 = int(input())
x1 = int(input())
y2 = int(input())
x2 = int(input())

def dist(a1, b1, a2, b2):
    return abs(a1 - a2) + abs(b1 - b2)

def dist_to_exit(y, x):
    return min(dist(y, x, y1, x1), dist(y, x, y2, x2))

for y in range(1, n + 1):
    for x in range(1, m + 1):
```

```
        curr_dist = dist_to_exit(y, x)
        move = 'S'
        for c, dy, dx in [('^',-1,0), ('v',1,0), ('<',0,-1), ('>',0,1)]:
            if dist_to_exit(y + dy, x + dx) < curr_dist:
                move = c
        print(move, end='')
    print()
```

Многие участники написали решение по-другому. Для каждой комнаты сначала определим, к какому из двух выходов нужно двигаться, сравнив расстояния от комнаты до выходов. Пусть координаты рассматриваемой комнаты (r, c), а ближайший выход находится в комнате (r_i, c_i). Если $r < r_i$, нужно вывести указатель «вниз», если $r > r_i$ — указатель «вверх». Если $c < c_i$ — указатель «вправо», а если $c > c_i$ —- указатель «влево».

Наконец, встречались и решения с нахождением кратчайшего маршрута при помощи алгоритма обхода графа в ширину, но в этой задаче это избыточно сложное решение.

Задача 3. Аркадий Аркадьевич делает грядки

В этой задаче нужно выбрать четыре различных элемента массива a_{i_1}, a_{i_2}, a_{i_3}, a_{i_4} так, чтобы минимизировать значение $\max(|a_{i_1} - a_{i_2}|, |a_{i_3} - a_{i_4}|)$.

Наименее эффективное решение сложности $O(n^4)$ заключается в переборе всех четырёх сторон прямоугольника вложенными циклами. Проверим, что выбраны разные доски (то есть значения индексов элементов i_1, i_2, i_3, i_4 попарно различны) и посчитаем для выбранных индексов значение размера уголка $\max(|a_{i_1} - a_{i_2}|, |a_{i_3} - a_{i_4}|)$. Запомним четвёрку элементов массива с минимальным значением размера уголка.

Такое решение набирает 20 баллов, пример такого решения.

```
n = int(input())
a = [int(input()) for i in range(n)]
r = 10**9
for i1 in range(n):
    for i2 in range(n):
        for i3 in range(n):
            for i4 in range(n):
                if len(set([i1, i2, i3, i4])) == 4:
                    d = max(abs(a[i1] - a[i2]), abs(a[i3] - a[i4]))
                    if d < r:
                        r = d
                        ans = [a[i1], a[i2], a[i3], a[i4]]
print(r)
print(*ans)
```

Переборное решение можно улучшить, если заметить, что четвёрку выбранных досок нужно объединять в пары так — две короткие доски вместе и две длинные вместе. Это позволит сократить перебор в $4! = 24$ раза. Один раз упорядочим список всех досок, затем переберём 4 индекса $i_1 < i_2 < i_3 < i_4$ и рассмотрим пары досок (a_{i_1}, a_{i_2}) и (a_{i_3}, a_{i_4}). Пример такого решения, которое также набирает 20 баллов.

```
n = int(input())
a = [int(input()) for i in range(n)]
a.sort()
r = 10**9
for i1 in range(n):
    for i2 in range(i1 + 1, n):
        for i3 in range(i2 + 1, n):
            for i4 in range(i3 + 1, n):
```

```
                    d = max(a[i2] - a[i1], a[i4] - a[i3])
                    if d < r:
                        r = d
                        ans = [a[i1], a[i2], a[i3], a[i4]]
print(r)
print(*ans)
```

Можно получить более эффективное решение, сократив перебор. Если упорядочить длины досок, то в качестве противоположных сторон грядки нужно выбирать две доски, которые идут рядом в упорядоченном массиве. То есть нужно выбрать для некоторого значения индекса i пару досок a_i и a_{i+1} и для некоторого значения индекса j пару досок a_j и a_{j+1}. Если перебирать два индекса i и j, то сложность алгоритма получится $O(n^2)$. Такое решение набирает 65 баллов.

```
n = int(input())
a = [int(input()) for i in range(n)]
a.sort()
r = 10**9
for i in range(n):
    for j in range(i+2, n-1):
            d = max(a[i+1] - a[i], a[j+1] - a[j])
            if d < r:
                r = d
                ans = [a[i], a[i+1], a[j], a[j+1]]
print(r)
print(*ans)
```

Для перехода к полному решению заметим, что после упорядочивания массива длин досок, нам нужно выбрать две пары элементов, идущих в упорядоченном массиве рядом, с наименьшим расстоянием между ними. Переберём соседние элементы упорядоченного массива длин досок и составим другой массив m из троек: разность этих элементов и их индексы $(a_{i+1} - a_i, i, i+1)$. Затем упорядочим этот массив и возьмём в нём первые два элемента, то есть две пары досок с наименьшим расстоянием.

Пример такого решения.

```
n = int(input())
a = [int(input()) for i in range(n)]
a.sort()
m = []
for i in range(n - 1):
    m.append((a[i+1] - a[i], i, i+1))
m.sort()
print(m[1][0])
print(a[m[0][1]], a[m[0][2]], a[m[1][1]], a[m[1][2]])
```

Это решение неправильное, оно выдаёт неверный ответ на многих тестах и набирает 55 баллов. Ошибка заключается в том, что выбранные две пары досок могут пересекаться, то есть одна доска может входить в две пары. Например, если длины досок равны 1, 2, 3, 10, то решение выберет две пары досок с минимальной разностью, это пары $(2, 1)$ и $(3, 2)$, при этом доска 2 вошла в обе пары. Это решение можно исправить следующим образом. После упорядочивания массива m нужно посмотреть на первые два его элемента (два минимальных элемента). Если они не имеют общей доски, то это и есть ответ. Иначе нужно рассмотреть вторую и третью минимальные пары, а также первую и третью пары. Хотя бы в одном из этих вариантов не будет пересечений, выберем эти две пары досок в качестве ответа.

Пример такого решения.

```
n = int(input())
```

```
a = [int(input()) for i in range(n)]
a.sort()
m = []
for i in range(n - 1):
    m.append((a[i+1] - a[i], i, i+1))
m.sort()
if m[0][2] != m[1][1] and m[0][1] != m[1][2]:
    print(m[1][0])
    print(a[m[0][1]], a[m[0][2]], a[m[1][1]], a[m[1][2]])
elif m[0][2] != m[2][1] and m[0][1] != m[2][2]:
    print(m[2][0])
    print(a[m[0][1]], a[m[0][2]], a[m[2][1]], a[m[2][2]])
else:
    print(m[2][0])
    print(a[m[1][1]], a[m[1][2]], a[m[2][1]], a[m[2][2]])
```

Возможны и другие способы решения. Все они используют сортировку длин досок. После сортировки можно пройти циклом, рассматривая разность двух соседних элементов. Также необходимо поддерживать минимальную разность двух соседних элементов на префиксе массива, до рассматриваемых элементов. Возьмём максимум из этих двух разностей, это есть необходимый размер уголка. Выберем наименьшее из полученных размеров уголков. Такое решение имеет сложность $O(n \log n)$, ввиду использования сортировки.

```
n = int(input())
a = [int(input()) for i in range(n)]
a.sort()
r = 10**9
ans = [0, 0, 0, 0]
min_pair = [a[0], a[1]]
for i in range(2, n - 1):
    if max(a[i+1] - a[i], min_pair[1] - min_pair[0]) < r:
        ans = min_pair + [a[i], a[i+1]]
        r = max(a[i+1] - a[i], min_pair[1] - min_pair[0])
    if a[i] - a[i-1] < min_pair[1] - min_pair[0]:
        min_pair = [a[i-1], a[i]]
print(r)
print(*ans)
```

Можно также использовать двоичный поиск по ответу (размеру уголка). Зафиксировав размер уголка, пройдём по упорядоченному массиву досок и посчитаем, можно ли выбрать две непересекающиеся пары досок, разность длин которых не превосходит выбранный размер уголка. Пример такого решения.

```
n = int(input())
a = [int(input()) for i in range(n)]
a.sort()
l = -1
r = 10**9
while r - l > 1:
    mid = (l + r) // 2
    cnt = 0
    i = 0
    while i < n - 1 and cnt < 2:
        if a[i+1] - a[i] <= mid:
            cnt += 1
```

```
            i += 2
        else:
            i += 1
    if cnt == 2:
        r = mid
    else:
        l = mid
print(r)
cnt = 0
i = 0
while cnt < 2:
    if a[i+1] - a[i] <= r:
        print(a[i], a[i+1])
        cnt += 1
        i += 2
    else:
        i += 1
```

Задача 4. Гравитационная сортировка

Решения, моделирующие описанный процесс сортировки, перемещая бусинки по одной, могут набрать 30 баллов. Для этого создадим двумерный массив размера $n \times n$, отмечая в нём бусинки и свободные места разными значениями, например, 1 и 0. Переберём в цикле стержни слева направо, на каждом стержне будем передвигать бусинки вниз, начиная с самой нижней. Бусинки передвигаются, пока ниже них есть свободное место. Ответом для данного стержня является количество перемещений самой верхней бусинки на стержне.

Такое решение имеет сложность $O(n^3)$. Пример такого решения.

```
n = int(input())
a = [[0] * n for i in range(n)]

for i in range(n):
    k = int(input())
    for j in range(k):
        a[i][j] = 1

for j in range(n):
    ans = 0
    for i in range(n):
        if a[i][j]:
            a[i][j] = 0
            k = i - 1
            while k >= 0 and a[k][j] == 0:
                k -= 1
            k += 1
            a[k][j] = 1
            ans = i - k
    print(ans)
```

Чтобы набрать 60 баллов, необходимо понять, как получается ответ для k-го стержня. Найдём на этом стержне самую верхнюю бусинку. Номер ряда этой бусинки – это такое наибольшее i, что $a_i \geqslant k$. Бусинка переместится вниз столько раз, сколько существует пустых мест на этом стержне в рядах ниже её, то есть это количество таких значений j, что $j < i$ и $a_j < k$. Если одним циклом перебирать номер стержня k, а вложенным циклом подсчитать количество таких рядов i, что $a_i < k$,

при этом есть ряд с большим номером, значение которого $a_i > k$, то такое решение будет иметь сложность $O(n^2)$ и получит 60 баллов. Пример такого решения.

```
n = int(input())
a = [int(input()) for i in range(n)]
for k in range(1, n + 1):
    ans = 0
    cnt = 0
    for i in range(n):
        if a[i] < k:
            cnt += 1
        else:
            ans = cnt
    print(ans)
```

Чтобы набрать 100 баллов, нужно научиться вычислять ответ быстро, общая сложность решения должна быть $O(n)$. Для этого не будем искать верхнюю бусинку на каждом стержне, а, наоборот, найдём ряды, в которых бусинки будут самым верхними на своих стержнях. Если в ряду находятся a_i бусинок, а во всех рядах выше бусинок меньше, то есть $a_j < a_i$ для всех $j > i$, то в ряду i некоторые бусинки будут верхними на своих стержнях. В примере из условия три бусинки из ряда $i = 5$ будут верхними на стержнях 1, 2, 3, а две бусинки из ряда $i = 4$ будут верхними на стержнях 4 и 5.

Такие значения в массиве, которые больше всех предыдущих значений, называются рекордами. Все рекорды можно найти за $O(n)$ однократным проходом по массиву. В этой задаче нужно искать рекорды с конца массива, то есть элементы, которые больше всех элементов с большими индексами.

Пусть a_i — рекорд. Определим, на каких стержнях бусинки в ряду i будут верхними. Если $m = \max\limits_{j=i+1\ldots n} a_j$ (наибольшее число бусинок в рядах выше i), то в ряду i бусинки на стержнях с номерами $m+1$, ..., a_j будут верхними. При обнаружении рекорда найдём ответ для тех стержней, на которых находятся эти бусинки. Ответ для стержня k ($m + 1 \leqslant k \leqslant a_i$) мы уже научились считать — это количество значений $a_j < k$, где $j < i$. Но нужно вычислить это значение быстро.

Для этого научимся отвечать на запросы, какое количество элементов массива меньше заданного числа. Это можно сделать подсчётом — создадим массив cnt, затем пройдём циклом по массиву и увеличим cnt[elem] на 1 для каждого элемента массива elem. После выполнения этого цикла значение cnt[i] будет равно количеству элементов в массиве, равных i. Затем снова пройдём циклом по массиву cnt, увеличивая значение cnt[i] на значение предыдущего элемента cnt[i+1]. После такого прохода значение cnt[i] будет равно количеству элементов массива, не превосходящих i.

Тем самым число элементов массива, меньших k, будет равно cnt[k−1]. Но из этого числа нужно вычесть число рядов, находящихся выше ряда i, оно равно $n-1-i$, потому что в этих рядах число бусинок также меньше k, но эти ряды не нужно учитывать в ответе.

Пример решения, набирающего 100 баллов. Сложность решения $O(n)$.

```
n = int(input())
a = [int(input()) for i in range(n)]

cnt = [0] * (n + 1)
for elem in a:
    cnt[elem] += 1  # Подсчёт числа элементов, равных i
for i in range(1, n + 1):
    cnt[i] += cnt[i - 1]  # Теперь это число элементов, не превосходящих i

rec = 0  # Текущий рекорд - максимум на суффиксе массива
for i in range(n - 1, -1, -1):
    if a[i] > rec:
        new_rec = a[i]  # Новое значение рекорда будет a[i]
```

```
        for k in range(rec + 1, new_rec + 1):
            print(cnt[k - 1] - (n - 1 - i))
        rec = new_rec

# Нужно вывести нули для всех оставшихся стержней - на них нет бусинок
for i in range(rec + 1, n + 1):
    print(0)
```

Задача 5. Шифр mint

В случае, когда строка короткая, можно использовать полный перебор способов назначить числа буквам, но такое решение не очень просто пишется и не поможет придумать полное решение задачи. Переборные решения могут набрать 20 баллов.

Для приближения к полному решению лучше разобрать частные случаи задачи, когда строка состоит только из букв «A» и «B» и когда строка состоит из букв от «A» до «I».

Если строка состоит из букв «A» и «B», то одну из букв нужно заменить на «1», а другую — на «2». Чтобы результат был как можно меньше, нужно первую букву строки и все такие же буквы заменить на «1», а другую букву — на «2». Пример такого решения (20 баллов).

```
s = input()
if s[0] == "A":
    s = s.replace("A", "1")
    s = s.replace("B", "2")
else:
    s = s.replace("B", "1")
    s = s.replace("A", "2")
print(s)
```

Если строка состоит только из букв от «A» до «I», нужно использовать однозначные числа от «1» до «9», при этом чем раньше буква встречается в строке, тем меньшее число нужно использовать для этой буквы. Переберём символы строки с начала, и при появлении ранее не встреченной буквы, заменим эту букву во всей строке на минимальное неиспользованое однозначное число. Пример такого решения (44 балла).

```
s = input()
d = 1
for i in range(len(s)):
    if "A" <= s[i] <= "I":
        s = s.replace(s[i], str(d))
        d += 1
print(s)
```

В полном решении используется та же идея — чем раньше стоит буква, тем меньшим числом нужно её заменить. Но числа при замене могут быть однозначными и двузначными. Чтобы полученное число была минимальным, прежде всего необходимо, чтобы оно содержало как можно меньше цифр. Поэтому часто встречающиеся буквы нужно заменять на однозначные числа, а редко встречающиеся буквы — на двузначные. Пример решения, в котором производится подсчёт числа появлений каждой буквы в строке, затем производится сортировка всех букв по частоте появления и 9 наиболее часто встречающихся букв заменяются на однозначные числа, а 9 менее часто встречающихся букв — на двузначные числа. Для определения, какое же именно число будет назначено конкретной букве, используется описанный выше жадный алгоритм (чем раньше происходит первое появление буквы, тем меньшее ей назначается число), но отдельно для однозначных и двузначных чисел.

```
s = input()
cnt = {chr(ord("A") + i): 0 for i in range(18)}
```

```
for c in s:
    cnt[c] += 1

letters = list(cnt.keys())
letters.sort(key = lambda c: cnt[c])

one_digit = letters[9:]

min1 = 1
min2 = 10

cipher = dict()
ans = ""
for c in s:
    if c not in cipher:
        if c in one_digit:
            cipher[c] = str(min1)
            min1 += 1
        else:
            cipher[c] = str(min2)
            min2 += 10
    ans += cipher[c]
print(ans)
```

Это решение набирает 60 баллов, оно неверно работает в случае, когда некоторые буквы в строке встречаются одинаковое число раз, поэтому их можно будет заменить как на однозначное число, так и на двузначное. Например, в строке «ABCDEFGHIJKLMNOPQR» все буквы встречаются по одному разу, из них половина будет заменена на однозначные, а половина — на двузначные числа. Но необходимо понять, как разбить буквы на классы однозначных и двузначных чисел для последующей замены.

Может возникнуть желание сделать полный перебор всех разбиений 18 букв на два подмножества по 9 букв. Число таких разбиений $C_{18}^{9} = 48620$. Для каждого разбиения необходимо также обработать всю строку, то есть общее число действий будет порядка 50 миллионов, что много для Python. Но эту задачу можно решить и без перебора.

Опять вернёмся к списку всех букв, упорядоченному по частоте их появления в строке. Наиболее часто встречающихся букв отнесём к множеству букв, которые будут заменены на однозначные числа, а наиболее редко встречающиеся буквы — на двузначные числа. Возможно появление и третьей группы «неопределённых» букв. Если в упорядоченном списке буквы, стоящие на 9-м и 10-м месте имеют одинаковую частоту появления, то они, а также все буквы с такой же частотой, попадут в группу неопределённых. Им может быть назначено как однозначное, так и двузначное число.

Далее так же идём по строке сначала, назначая буквам числа. Для этого нужно хранить минимальное неиспользованное однозначное и двузначное число. Буквам из множества однозначных или двузачных чисел сразу назначается минимальное число нужной длины. Для букв из группы неопределённых нужно сравнить минимальное доступное однозначное и двузначное число и выбрать то из них, у которого меньше первая цифра. Если начальные цифры равны, например, если можно выбрать число «2» или «20», то нужно выбирать двузначное. Например, для строки «ABCDEFGHIJKLMNOPQR», в которой все буквы неопределённые, ответ получится «101202303...909».

Также нужно учесть, что буквам неопределённой группы можно назначить ровно определённое число однозначных и двузначных чисел, в зависимости от того, сколько букв попало в другие группы. Нужно считать, сколько осталось однозначных и двузначных чисел, доступных для назначения неопределённым буквам, и если, например, неопределёным буквам уже назначили необходимое чис-

ло однозначных чисел, оставшимся неопределённым буквам будут назначаться только двузначные числа и наоборот.

Сложность алгоритма будет всего лишь $O(n)$, что позволяет решить задачу на любом языке программирования. Пример такого решения.

```
s = input()
cnt = {chr(ord("A") + i): 0 for i in range(18)}
for c in s:
    cnt[c] += 1

letters = sorted(cnt.keys(), key=lambda c: cnt[c])

middle_freq = (cnt[letters[8]] + cnt[letters[9]]) / 2

one_digit = {c for c in letters if cnt[c] > middle_freq}
two_digits = {c for c in letters if cnt[c] < middle_freq}

cnt_free_one_digit = 9 - len(one_digit)
cnt_free_two_digits =  9 - len(two_digits)

min1 = 1
min2 = 10

cipher = dict()
ans = ""
for c in s:
    if c not in cipher:
        if c in one_digit:
            cipher[c] = str(min1)
            min1 += 1
        elif c in two_digits:
            cipher[c] = str(min2)
            min2 += 10
        elif cnt_free_two_digits==0 or cnt_free_one_digit and min1<min2/10:
            cipher[c] = str(min1)
            min1 += 1
            cnt_free_one_digit -= 1
        else:
            cipher[c] = str(min2)
            min2 += 10
            cnt_free_two_digits -= 1
    ans += cipher[c]

print(ans)
```