Всероссийская олимпиада школьников по информатике
Вологодская область, 2024-25 учебный год
II (муниципальный) этап
9 - 11 классы
Методические рекомендации по разбору задач

Задача 1. Снова ремонт

Каждый раз мы отрезаем от рулона полосу длины h. Если полоса – не последняя из этого рулона, то затем надо ещё отрезать остаток от обрезанного рисунка, чтобы рулон снова начинался с начала рисунка. То есть, мы отрежем h/k с округлением вверх полных рисунков.

Округлить вверх результат деления можно так: (h+k-1) // k, где операция ‘//’ означает деление нацело. Умножим это на k и получим длину полосы вместе с длиной дополнительного кусочка: segment_len = (h + k - 1) // k * k. Теперь найдём, сколько таких получится из одного рулона: segments = h // segment_len

После этого от рулона останется: restlen = h - segments * segment_len

Заметим, что когда от рулона отрезается последняя полоса, то дополнительный кусок отрезать уже не надо. Поэтому оставшейся длины restlen может хватить ещё на целую полосу – проверим это:

если restlen ≥ h, то segments = segments + 1

Итак, мы нашли, что из одного рулона получается segments полос. Они покроют на стене длину width = segments * m. Тогда, чтобы покрыть всю длину стены, потребуется a / width с округлением вверх рулонов. Округление вверх при делении можно снова записать как (a + width - 1) // width – это и будет ответ.

Пример решения на языке Python:

```
a = int(input())
h = int(input())
k = int(input())
m = int(input())
s = int(input())
segment_len = (h + k - 1) // k * k
segments = s // segment_len
restlen = s - segments * segment_len
if restlen >= h:
   segments += 1
width = segments * m
rolls = (a + width - 1) // width
print(rolls)
```

В решении на неполный балл можно, было, например, в цикле выполнять отрезание от рулона, пока он не закончится.

Задача 2. Согласование заявок

Заметим, что для согласования всех заявок в любом случае нужно n раз нажать кнопку «Согласовать» после того как отмечены все заявки очередного типа.

Также заметим, что кнопку «Отметить все» не имеет смысла нажимать более одного раза. Если она нажималась хотя бы дважды, то в не последний раз её использования галочки снимались с заявок. Но снятие галочек с заявок не даёт никакого выигрыша: можно было просто за такое же количество нажатий отметить заявки данного типа и согласовать их. Поэтому кнопку «Отметить все» нужно использовать, только когда остались заявки одного типа. И наиболее выгодно, чтобы это был тип с самым большим количеством заявок. Все же остальные заявки будем отмечать по одной. Поэтому ответ на вопрос задачи можно вычислить следующим образом:

$$n + \sum_{i=1}^{n} a_i - \max_{i=1}^{n} a_i + 1$$

Пример решения на языке Python:

```
n = int(input())
a = [0] * n
for i in range(n):
   a[i] = int(input())
print(n + sum(a) - max(a) + 1)
```

Частичные баллы за эту задачу могли получить, например, решения, в которых выполнялась сортировка неэффективным способом, и др.

Задача 3. Треугольники

Первая подзадача решается перебором троек с помощью трёх вложенных циклов и проверкой правила треугольника – каждая сторона меньше суммы двух других. Чтобы не было дубликатов, организуем перебор так, чтобы выполнялось $a \le b \le c$, то есть цикл по b стартует с a, цикл по c стартует с b: Пример такого решения на Python:

```
P = int(input())
Q = int(input())
```

```
ans = 0
for a in range(P, Q + 1):
   for b in range (a, Q + 1):
      for c in range(b, Q + 1):
         if a<b+c and b<a+c and c<a+b:
            ans += 1
print(ans)
```

Чтобы решить вторую подзадачу, устраним самый внутренний цикл. Заметим, что условие $c < a + b$ можно проверять не в теле цикла, а внести его под *range*. Два оставшихся условия в теле цикла избыточны: поскольку у нас $a \le b \le c$, то отсюда верно, что $a < b + c$ и также что $b < a + c$. Получаем такой вариант внутреннего цикла:

```
for c in range(b, min(a + b, Q + 1)):
        ans += 1
```

Отсюда понятно, что вместо прибавления единицы в цикле можно сразу прибавить к ans количество повторений цикла. Получается такое решение второй подзадачи:

```
P = int(input())
Q = int(input())
ans = 0
for a in range(P, Q + 1):
  for b in range (a, Q + 1):
      ans += min(a + b, Q + 1) - b
print(ans)
```

Чтобы решить третью подзадачу, устраним ещё один цикл – теперь по переменной *b*. Заметим, что в этом цикле функция *min* вначале каждый раз возвращает *a*+*b*. Затем, начиная с какого-то момента, *a*+*b* становится слишком большим, и функция *min* начнёт возвращать $Q + 1$. Это произойдёт в тот момент, когда выполнится равенство $a + b = Q + 1$, то есть когда $b = Q + 1 - a$. Перепишем программу уже с двумя циклами по *b*, не используя функцию min (это пока всё ещё решение второй подзадачи, только слегка изменённое):

```
P = int(input())
Q = int(input())
ans = 0
for a in range(P, Q + 1):
  for b in range (a, Q + 2 - a):
      ans += a
  for b in range (max(a, Q + 2 - a), Q + 1):
      ans += Q + 1 - b
print(ans)
```

Теперь заметим, что в первом цикле каждый раз к ответу прибавляется константа, поэтому можно заменить цикл прибавлением к ответу этой константы,

умноженной на количество итераций (при подсчёте числа итераций нужно проверить, что оно будет неотрицательным).

Во втором цикле прибавляется константа $Q + 1$ и вычитается переменная b, которая увеличивается на каждом шаге. Сумму этих вычитаний можно найти с помощью формулы суммы арифметической прогрессии, поэтому этот цикл тоже устраняем. Окончательно получается следующее решение на полный балл:

```
P = int(input())
Q = int(input())
ans = 0
for a in range(P, Q + 1):
  if Q + 2 - a > a:
     ans += a * (Q + 2 - a - a)
  if Q + 1 > max(a, Q + 2 - a):
     ans += (Q + 1) * (Q + 1 - max(a, Q + 2 - a))
     ans -= (max(a, Q+2-a) + Q) * (Q + 1 - max(a, Q+2-a)) // 2
print(ans)
```

Примечание. В принципе, можно было попытаться сделать ещё один шаг и устранить цикл по a, получив решение за константное время. Однако, в данной задаче это не требовалось.

Задача 4. Маша и Глеб переезжают

В задаче требуется найти два соседних отрезка массива длиной один или более элементов, чтобы суммы на отрезках отличались как можно сильнее. При этом могут быть отрицательные числа.

В цикле переберём позицию i, где кончается первый отрезок (соответственно, в следующей позиции начинается второй). Пусть *minSumLeft*[i] – минимально возможная сумма среди всех отрезков, которые кончаются в позиции i, *minSumRight*[i+1] – минимально возможная сумма среди всех отрезков, которые начинаются в позиции i+1. Аналогично обозначим и максимумы: *maxSumLeft*[i] и *maxSumRight*[i+1]. Тогда для позиции i максимально возможная разность сумм отрезков равна:

max(maxSumRight[i+1] - minSumLeft[i], maxSumLeft[i] - minSumRight[i+1])

Перебрав все позиции i, выбираем ту, где данное значение максимально.

Осталось понять, как быстро вычислить массивы *minSumLeft*, *minSumRight*, *maxSumLeft* и *maxSumRight*. Разберём один из способов вычисления массива maxSumLeft (остальные массивы заполняются аналогично).

Пусть индексация идёт с нуля. Очевидно, что *maxSumLeft*[0] = *a*[0]. Остальные элементы вычислим в цикле. Пусть сейчас нужно вычислить *maxSumLeft*[*j*]. Посмотрим на позицию *j*-1. Если *maxSumLeft*[*j*-1] < 0, то это значит, что нам невыгодно начинать отрезок левее позиции *j*, чтобы не прибавлять отрицательное значение. Выгоднее всего сделать отрезок из одного элемента a[j], и тогда maxSumLeft[j] = a[j].

В противном случае мы берём отрезок, кончающийся в позиции *j*-1, и удлиняем его ещё на один элемент: *maxSumLeft*[*j*] = *maxSumLeft*[*j* - 1] + *a*[*j*].

В результате получаем решение, работающее за линейное время. Пример такого решения на Python:

```
n = int(input())
a = [None] * n
for i in range(n):
   a[i] = int(input())
Inf = int(1e18)

minSumLeft = [None] * n
maxSumLeft = [None] * n
minSumLeft[0] = maxSumLeft[0] = a[0]
for j in range(1, n):
  if maxSumLeft[j - 1] < 0:
     maxSumLeft[j] = a[j]
  else:
     maxSumLeft[j] = maxSumLeft[j - 1] + a[j]
  if minSumLeft[j - 1] > 0:
     minSumLeft[j] = a[j]
  else:
     minSumLeft[j] = minSumLeft[j - 1] + a[j]

minSumRight = [None] * n
maxSumRight = [None] * n
minSumRight[n - 1] = maxSumRight[n - 1] = a[n - 1]
for j in range(n - 2, -1, -1):
  if maxSumRight[j + 1] < 0:
     maxSumRight[j] = a[j]
  else:
     maxSumRight[j] = maxSumRight[j + 1] + a[j]
  if minSumRight[j + 1] > 0:
     minSumRight[j] = a[j]
  else:
     minSumRight[j] = minSumRight[j + 1] + a[j]

ans = -Inf
for i in range(n - 1):
   ans = max(ans, maxSumRight[i + 1] - minSumLeft[i],
maxSumLeft[i] - minSumRight[i + 1])
print(ans)
```

Рассмотрим возможные частичные решения.

Первую подзадачу можно решить совсем «в лоб» четырьмя вложенными циклами.

Для второй подзадачи можно написать простое решение за куб. Первый цикл перебирает начало первой группы участков, второй цикл - конец первой группы и одновременно начало второй, третий – конец второй группы участков. Их суммарные стоимости можно вычислять по ходу, чтобы не вводить четвёртый цикл.

В третьей подзадаче можно было написать решение за квадрат. Внешним циклом ищем позицию, в которой группы участков встречаются. Циклом влево перебираем начало первого отрезка, считаем текущую сумму этого отрезка и запоминаем её минимальное и максимальное значение *MinLeft* и *MaxLeft*. Аналогично, циклом вправо перебираем конец второго отрезка, считаем сумму и запоминаем её минимальное и максимальное значение *MinRight* и *MaxRight*. Осталось взять max(MaxRight - MinLeft, MaxLeft - MinRight), и для всех позиций выбрать ту, где это значение максимально.

Задача 5. Дима на рыбалке

Задача решается методом динамического программирования. Заведем массивы: *center*[*t*] - сколько максимум рыбы может выловить Дима, если в момент t он будет находиться в центре острова; *rods*[*i*] - сколько максимум рыбы может выловить Дима, если в текущий момент времени *timer* он будет находиться у удочки *i*.

Заметим, что в массиве *rods* для каждого момента времени *timer* у нас может поменяться только один элемент – *rods*[*rod*], где rod=*a*[*timer*]. Чтобы вычислить его значение, заметим, что у нас есть два варианта: мы либо минуту назад находились тоже у этой удочки, либо только что пришли из центра. Выбираем максимум:

rods[*rod*] = *max*(*rods*[*rod*], *center*[*timer* - *d*[*rod*]]) + 1

Поймав рыбину, мы можем пойти в центр, поэтому обновим значение в массиве *center* в будущем (в тот момент, когда туда доберёмся):

center[*timer* + *d*[*rod*]] = *max*(*center*[*timer* + *d*[*rod*]], *rods*[*rod*])

Ещё заметим, что если мы в момент *timer* находимся в центре, то мы могли там оказаться, не только придя откуда-то, а просто простояв в центре минуту, поэтому:

center[*timer*] = *max*(*center*[*timer*], *center*[*timer* - 1])

Пример решения на Python:

```
Inf = int(1e9)
t = int(input())
n = int(input())
a = [None] * (t + 1)
for i in range(1, t + 1):
   a[i] = int(input())
d = [None] * (n + 1)
for i in range(1, n + 1):
   d[i] = int(input())

center = [0] * (2 * t + 1)
rods = [0] * (n + 1)
for timer in range(1, t + 1):
   center[timer] = max(center[timer], center[timer - 1])
   rod = a[timer]
   if timer - d[rod] >= 0:
      rods[rod] = max(rods[rod], center[timer - d[rod]]) + 1
   center[timer + d[rod]] = max(center[timer + d[rod]],
rods[rod])

print(max(rods))
```

Описанное полное решение является относительно непростым. Однако, в задаче выделен целый ряд более простых подзадач. Опишем кратко подходы к их решению.

Первая подзадача решается очень просто. Если к моменту времени *d* Дима дойдет до единственной удочки и каждую минуту будет вытаскивать по рыбине до момента *t*, то всего он выловит *t*-*d*+1 рыб.

Во второй подзадаче можно применить динамическое программирование следующим образом. Пусть f1[i] – сколько максимум рыб можно выловить на первую удочку к моменту времени i, f2[i] – на вторую удочку.

Пусть d = d1 + d2 – время перехода от одной удочки к другой. Тогда:

f1[i] = max(f1[i-d], f2[i-d]). При этом если a[i]=1, то ещё добавить 1.

f2[i] = max(f1[i-d], f2[i-d]). При этом если a[i]=2, то ещё добавить 1.

Ответом будет max(f1[t], f2[t]).

В третьей подзадаче можно было обобщить решение второй подзадачи и написать решение за куб либо более оптимальную версию за квадрат.

Четвёртую подзадачу можно было решить так. Идём по увеличению времени и каждый раз рассматриваем удочку, на которую клюёт в этот момент. Рыбак мог быть возле этой удочки ещё минуту назад либо мог только что прийти от другой удочки, потратив на это две минуты. Понятно, что приходить ему выгодней всего от той удочки, где он мог получить максимум две минуты назад. Отсюда идея: хранить этот максимум, но с двухминутным отставанием от текущего времени, а также хранить ещё кандидата в максимумы – уже с одноминутным отставанием.