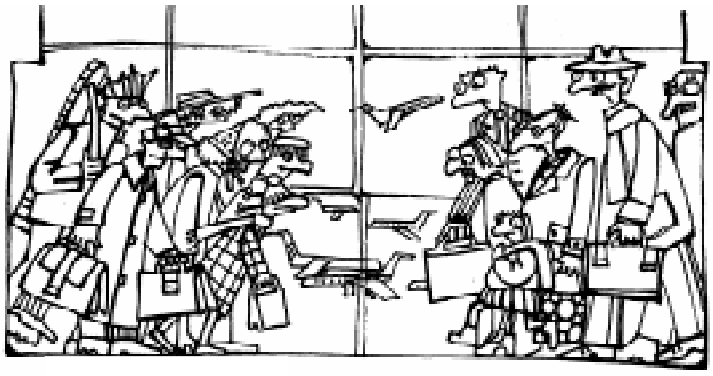
Елизаров Роман Анатольевич

РАЗБОР ЗАДАЧ 25-ГО ФИНАЛА КОМАНДНОГО ЧЕМПИОНАТА МИРА ACM ПО ПРОГРАММИРОВАНИЮ

В прошлом номере журнала был опубликован анализ шести задач чемпионата мира, решенных командами, получившими золотые медали. В этом номере мы публикуем анализ трех задач чемпионата мира, вызвавших наибольшие трудности у участников соревнований.

Задача A. Конфигурация аэропорта.



«Авиалинии ACM» – это региональная авиакомпания, главным аэропортом которой является аэропорт Фон Неймана, через который проходит множество транзитных пассажиров. Аэропорт Фон Неймана представляет собой коридор. Входы для прибывающих пассажиров расположены на равном расстоянии друг от друга на северной стороне коридора. Входы для отбывающих пассажиров расположены на южной стороне коридора, также на равном расстоянии друг от друга. Расстояние между входами равно ширине коридора. Каждый вход для прибывающих пассажиров назначен в точности одному городу. То же верно и для отбывающих пассажиров. Каждый транзитный пассажир попадает в коридор через вход, назначенный для его города прибытия, и следует ко входу, назначенному для его города назначения. В этой задаче рассматриваются только транзитные пассажиры.

Из-за наличия магазинов и садов коридор нельзя пересечь по диагонали. Таким образом, расстояние между входом для прибывающих пассажиров G1 и входом для отбывающих пассажиров G3 (рисунок 1) равно 1 + 2 = 3.

Среднее число пассажиров, путешествующих между любой парой городов, известно заранее. Вам необходимо оценить общую транспортную нагрузку аэропорта для нескольких различных конфигураций аэропорта. Нагрузка между входом для прибывающих пассажиров и входом для отбывающих пассажиров определяется как количество пассажиров, перемещающихся между этими входами, умноженное на расстояние между ними. Общая транспортная нагрузка – это сумма нагрузки по всем прибывающим-отбывающим парам входов.

Входные данные (файл airport.in).

Входной файл состоит из нескольких наборов тестовых данных. После

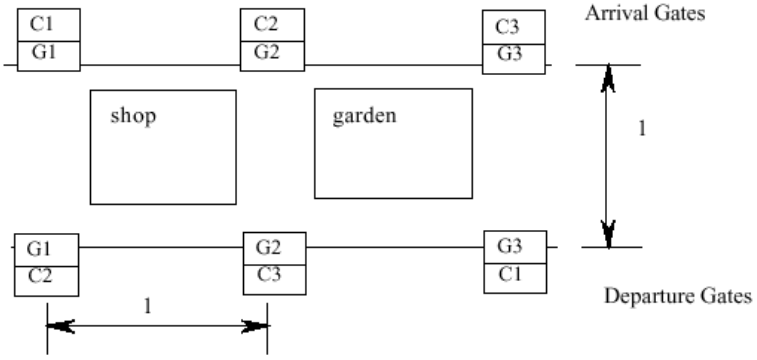


Рисунок 1

последнего набора тестовых данных следует строка, содержащая число 0. Каждый набор тестовых данных состоит из двух частей: информация о транспортной нагрузке и конфигурационная секция.

Информация о нагрузке начинается с целого числа N ($1 < N < 25$), представляющего собой число городов. Далее следуют N строчек, представляющих собой информацию о нагрузке для каждого города. Каждая строка информации о нагрузке начинается целым числом из диапазона 1..N, которое обозначает город отправления. После него следует целое число k и k пар целых чисел, по одной паре для каждого города назначения. Каждая пара определяет город назначения и число путешествующих по этому маршруту пассажиров (не более 500).

Конфигурационная секция состоит из некоторого числа (не более 20) конфигураций аэропорта и заканчивается строкой, содержащей число 0. Каждая конфигурация состоит из 3-х строк. Первая строка содержит положительное целое число, идентифицирующее конфигурацию. Следующая строка задает перестановку городов, как они назначены входам для прибывающих пассажиров: первое число представляет собой номер города, назначенного первому входу и т. д. Следующая строка таким же образом задает города, которые назначены входам для отбывающих пассажиров.

Выходные данные.

Для каждого набора входных данных выход представляет собой таблицу с номерами конфигураций и общей транспортной нагрузкой в порядке ее возрастания. Если две конфигурации имеют одинаковую нагрузку, то первой должна идти та, у которой меньший номер конфигурации. Придерживайтесь формата вывода, указанного в примере.

Пример.

Входной файл.

```
3
1 2 2 10 3 15
2 1 3 10
3 2 1 12 2 20
1
1 2 3
2 3 1
2
2 3 1
3 2 1
0
2
1 1 2 100
2 1 1 200
1
1 2
1 2
2
1 2
2 1
0
0
```

Вывод.

```
Configuration  Load
      2         119
      1         122
Configuration  Load
      2         300
      1         600
```

Решение.

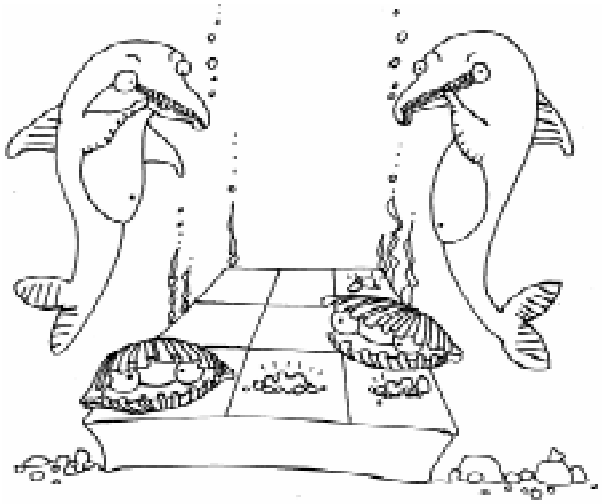
Решение этой задачи не представляет алгоритмической сложности, но требует некоторой аккуратности (программа представлена в приложении 1). На соревнованиях многие участники не могут решить задачи подобного рода с первой попытки из-за неверно прочитанного условия или пропуска каких-либо важных мелочей.

Будем хранить информацию о транспортной нагрузке в двухмерном массиве load (строка 14), в котором будет записано число путешествующих пассажиров между каждой парой городов (или 0, если оно не задано). Конфигурационную секцию будем хранить в двух одномерных массивах arrival и departure (строки 18–19). Обратите внимание на необходимость очистки массива load перед загрузкой каждого набора данных (строка 28).

Общую нагрузку для каждой конфигурационной секции придется запоминать в отдельной структуре данных totals (строки 15–16), чтобы можно было выполнить сортировку перед выводом ответа на экран.

Обсчет каждой конфигурации и запоминание результатов в массиве totals производится в соответствии с условием задачи путем перебора всех пар входов для прибывающих и отбывающих пассажиров (строки 50–58). Вся процедура обсчета потребует время порядка $O(C \cdot N^2)$, где C – количество конфигураций.

После обсчета всех конфигураций для текущего набора входных данных остается только отсортировать массив totals в соответствии с условием задачи и вывести ответ. Для такого небольшого количества элементов (по условию задачи есть не более 20 конфигураций в каждом наборе входных данных) прекрасно подойдет простейшая сортировка выбором (строки 60–70), что даст алгоритм с общим временем выполнения порядка $O(C \cdot N^2 + C^2)$.



Задача Е. Моллюски.

Исследователи запускают пару специально тренированных моллюсков в разных углах прямоугольного поля, разбитого на одинаковые квадратные клетки. Они ползут, разбрасывая светящиеся химикаты, присоединенные к их раковинам, в те клетки, через которые они проползают. Моллюски натренированы так, чтобы перемещаться на одну клетку за единицу времени, аппроксимируя заданный вектор. Если перемещение выводит моллюска за пределы поля, то тренированный дельфин мгновенно переносит его на противоположный край, обеспечивая механизм «заворачивания». Перемещения моллюсков синхронизированы, однако моллюск останавливается после попадания в клетку, которую он уже посетил. Если два моллюска попадают в одну и ту же клетку, то они там останавливаются. Если два моллюска пытаются переместиться в клетки друг друга, то они также останавливаются.

Начальное положение моллюска таково, чтобы вектор его движения был направлен «внутрь» поля. Оба моллюска начинают движение в момент времени $t = 1$ в различных углах поля. Моллюск следует своему вектору так, как будто начальная точка вектора закреплена в центре клетки, из которой моллюск начал свое движение. Моллюск всегда передвигается в следующую клетку, которую вектор (или его продолжение) разбивает на части, за одним исключением: если вектор проходит через угол клетки, то моллюск перемещается сначала горизонтально, а затем вертикально для того, чтобы достичь следующей клетки, разделяемой вектором на части. На рисунках 2а, 2б показаны два примера движения моллюска (номера в клетках указывают прошедшее время). Координаты клеток отсчитываются, начиная с нуля, по осям x и y.

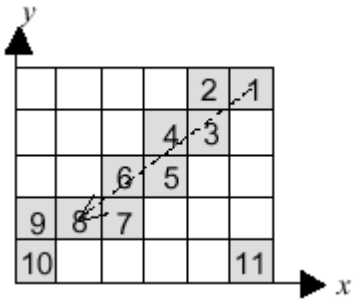


Рисунок 2а. *Моллюск с вектором (-4,-3) перемещается по полю 6 на 5, начиная с клетки (5,4). В момент времени 12 он останавливается, так как заново попадает в клетку (5,4).*

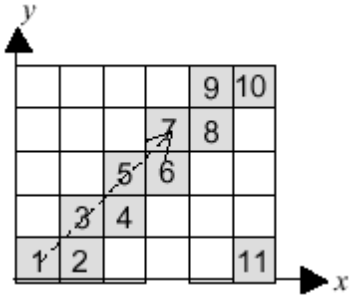


Рисунок 2б. *Моллюск с вектором (1,1) перемещается по полю 6 на 5, начиная с клетки (0,0). В момент времени 12 он останавливается, так как заново попадает в клетку (0,0).*

Если бы оба моллюска на рисунке начали движение на одном и том же поле 6 на 5, то они остановились бы в момент времени 5, осветив в сумме 10 клеток.

Вы должны написать программу, которая выберет пару моллюсков, которые осветят наибольшее число клеток за минимальное время. Повторите вычисления для полей разных размеров и разных наборов моллюсков.

Входные данные (файл geoduck.in).

Входной файл состоит из набора тестовых данных, каждый из которых начинается со строки, содержащей два целых числа m и n – размеры поля по осям x и y, соответственно ($1 \le m, n \le 50$), где m и n не могут быть оба равны 1. Вторая строка каждого набора содержит целое число k – количество моллюсков ($2 \le k \le 10$). Как минимум, одна пара моллюсков будет иметь различные начальные клетки. Следующие k строк содержат пары ненулевых целых чисел, представляющие собой компоненты x и y векторов направлений движения моллюсков. После последнего набора тестовых данных идет строка с двумя нулями.

Выходные данные.

Для каждого набора тестовых данных выведите его номер по порядку, максимальное количество клеток, которые можно осветить, минимально возможное время и номера всех пар моллюсков, которые достигают этих значений (максимального числа клеток за минимально возможное время). Порядок вывода не имеет значения, однако печатать одну и ту же пару два раза не нужно. Придерживайтесь формата вывода, указанного в примере.

Пример.

Входной файл.

```
6 5
3
-4 -3
1 1
1 -1
0 0
```

Вывод.

```
Case 1
Cells Illuminated: 10
Minimum Time:  5
     Geoduck IDs:  1  2
     Geoduck IDs:  1  3
```

Решение.

В этой задаче необходимо правильно промоделировать описанный процесс перемещения моллюсков по полю, однако в этом процессе содержится несколько подвохов. Обратите внимание на то, что моллюск останавливается после попадания в клетку, которую *он* уже посетил. Это условие подразумевает запоминание посещенных клеток *для каждого моллюска в отдельности*. Второй подвох заключается в требовании вычислить минимальное время, за которое будет закрашено максимальное число клеток, а это может произойти до того, как оба моллюска остановятся.

Для моделирования перемещения каждого моллюска в отдельности удобно запоминать не только текущую координату моллюска на поле, но и вектор (D_x, D_y) на который моллюск переместился бы, если бы не было механизма «заворачивания». Рассмотрим моллюска, обе компоненты скорости (V_x, V_y) которого положительны, и решим задачу определения следующего перемещения этого моллюска – на одну клетку вверх или на одну клетку вправо. Прямая линия, которую пытается аппроксимировать моллюск, может быть задана следующим параметрическим уравнением:

$$\begin{cases} x = \frac{1}{2} + V_x \cdot t \\ y = \frac{1}{2} + V_y \cdot t \end{cases}$$

где t возрастает с удалением моллюска от начальной точки. Таким образом, для принятия решения о том, в какую клетку перемещаться, необходимо лишь выяснить, какое условие наступит «раньше» (то есть при меньшем значении t): $x = D_x + 1$ или $y = D_y + 1$. Выписав соот-

ветствующие уравнения для переменной t, найдем, что моллюск перемещается вправо, если:

$$\frac{D_x + \frac{1}{2}}{V_x} \le \frac{D_y + \frac{1}{2}}{V_y}.$$

Здесь стоит знак «меньше или равно», так как, в соответствии с условием задачи, при прочих равных условиях, моллюск перемещается по горизонтали, то есть по оси *X*. Обратите внимание, что после приведения к общему знаменателю вышеприведенное условие может быть проверено в целых числах, без привлечения вещественной арифметики.

Используя эти рассуждения, несложно написать процедуру, которая моделирует перемещения двух моллюсков до их останова *или* до закраски определенного числа клеток. Моделирование перемещения всех пар моллюсков до их останова позволит вычислить максимально возможное число закрашенных клеток. Повторное моделирование перемещений всех пар моллюсков, вплоть до закраски этого числа клеток, позволит вычислить минимально необходимое время. После чего можно вывести ответ, например, снова промоделировав перемещение всех пар моллюсков и выводя номера тех из них, которые достигают найденных параметров.

Каждое моделирование перемещения пары моллюсков занимает время порядка O(mn). А весь алгоритм решения задачи (выполняющий моделирование перемещения для всех пар моллюсков) будет работать время порядка O(k^2mn).

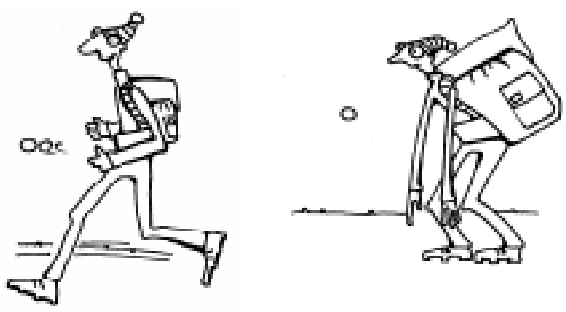
Задача G. Распределение памяти.

В ранних многозадачных системах оперативная память делилась на несколько областей фиксированного, но, возможно, различного размера. Сумма размеров всех областей равнялась размеру всей оперативной памяти. Получив набор программ, операционная система должна была так распределить их по областям памяти, чтобы они могли выполняться одновременно. Это не просто, так как время, которое занимает выполнение программы, может зависеть от количества выделенной ей памяти. Каждая программа имеет минимальные требования к памяти, но если ей выделена область памяти большего размера, то ее время выполнения может увеличиться или уменьшиться.

В данной задаче вы должны определить оптимальное распределение задач по областям памяти. Вашей программе даны размеры всех областей памяти и для каждой программы описание того, как ее время выполнения зависит от количества доступной ей памяти. Вы должны найти план выполнения программ, который минимизирует среднее время цикла обработки программы. План выполнения – это назначение каждой программе области памяти и времени таким образом, чтобы две программы не использовали одну и ту же область памяти в одно и то же время и чтобы ни одной программе не была назначена область памяти меньшего размера, чем минимально ей необходимая. Цикл обработки программы – это время, прошедшее с момента подачи заявки на ее запуск (оно равняется нулю для всех программ в этой задаче) до того момента, когда программа закончит свое выполнение.

Входные данные (файл memory.in).

Входной файл состоит из нескольких наборов тестовых данных. Каждый набор тестовых данных начинается со строки с двумя целыми числами m и n. Число m задает количество областей, на которые поделена память ($1 \le m \le 10$), а n – количество программ ($1 \le n \le 50$). Следующая строка содержит m положительных целых чисел, задающих размер m



областей памяти. Далее следуют n строк, задающих соответствие между количеством памяти и временем выполнения для каждой из n программ. Каждая строка начинается с положительного целого k ($k \le 10$), за которым следует k пар положительных целых чисел s_1, t_1, s_2, t_2, ..., s_k, t_k, удовлетворяющих условию $s_i < s_{i+1}$ для $1 \le i < k$. Минимальное количество памяти, которое необходимо программе, s_1. Если программа работает в области памяти s, где $s_i \le s < s_{i+1}$ для некоторого i, то ее время выполнения будет t_i. Если программа работает в области памяти размера s_k или больше, то ее время выполнения будет t_k.

После последнего набора тестовых данных идет пара нулей.

Вы можете предполагать, что выполнение каждой программы займет указанное время для данного размера области, вне зависимости от количества других программ в системе. Никакая программа не будет требовать размеров памяти больше, чем размер самой большой области памяти.

Выходные данные.

Для каждого набора тестовых данных сначала выведите его номер (начинайте с 1 и последовательно его увеличивайте). Затем выведите минимальное среднее время цикла обработки программы с двумя знаками после запятой. Далее выведите план выполнения, на котором достигается этот минимум. Выведите по одной строке на каждую программу в том порядке, в котором они даны во входном файле, что определит номер программы, область, в которой программа выполняется (нумерация в порядке входного файла), время начала выполнения программы и время, когда программа заканчивает свое выполнение. Придерживайтесь формата вывода, указанного в примере, и выводите пустую строку после каждого набора тестовых данных.

Если существует несколько планов выполнения, на которых достигается минимальное среднее время цикла обработки программы, то выведите любой из них.

Пример.

Входной файл.

```
2 4
40 60
1 35 4
1 20 3
1 40 10
1 60 7
3 5
10 20 30
2 10 50 12 30
2 10 100 20 25
1 25 19
1 19 41
2 10 18 30 42
0 0
```

Вывод.

```
Case 1
Average turnaround time = 7.75
Program 1 runs in region 1 from 0 to 4
Program 2 runs in region 2 from 0 to 3
Program 3 runs in region 1 from 4 to 14
Program 4 runs in region 2 from 3 to 10

Case 2
Average turnaround time = 35.40
Program 1 runs in region 2 from 25 to 55
Program 2 runs in region 2 from 0 to 25
Program 3 runs in region 3 from 0 to 19
Program 4 runs in region 3 from 19 to 60
Program 5 runs in region 1 from 0 to 18
```

Решение.

Заметим, что среднее время цикла обработки – это сумма времени обработки по всем программам, поделенное на количество программ n, которое постоянно. Таким образом, нахождение минимального среднего времени цикла обработки программы эквивалентно нахождению минимального суммарного времени цикла обработки программы.

Проанализировав соответствие между количеством памяти и временем выполнения для каждой из n программ, составим таблицу (a_{ij}) для $(1 \le i \le n, 1 \le j \le m)$, где a_{ij} – это время, за которое будет выполняться программа i, если ее запустить в регионе памяти j. Положим $a_{ij} = \infty$, если программа i не может выполняться в области памяти j. Например, вто-

рому тестовому набору из примера будет соответствовать таблица 1 (строки – программы, столбцы – области памяти).

Представим план выполнения в виде соответствия каждой программы некоторому положению p_{jk} в плане. Будем говорить, что программа i выполняется в положении p_{jk} ($1 \le j \le m$, $1 \le k \le n$), если она выполняется k-ой с конца в j-ой области памяти. Например, программа в положении $p_{2,1}$ выполняется последней во втором регионе памяти, а программа в положении $p_{5,2}$ выполняется предпоследней в пятом регионе памяти. Вклад в суммарное время цикла обработки программы, вносимый программой i, выполняемой в положении p_{jk}, равен $k \cdot a_{ij}$, что учитывает собственно время выполнения программы и то, что ее выполнение увеличит время окончания k–1 программ, которые будут выполняться в этой области памяти после нее. Суммарное время цикла обработки программы будет суммой этих вкладов для каждой программы. Таким образом, задача сводится к известной задаче «о распределении» – нахождению паросочетания с оптимальным (в данном случае минимальным) весом во взвешенном двудольном графе. Например, для второго тестового набора из примера необходимо решить задачу распределения для матрицы, представленной в таблице 2 (строки – программы, столбцы – положения).

Выделенные ячейки таблицы соответствуют оптимальному распределению программ по положениям. Среднее время цикла обработки в этом случае равно (30+50+38+41+18)/5 = 35.4.

Нахождение оптимального распределения программ по положениям удобно производить за n шагов, на i-ом шаге ($1 \le i \le n$) добавляя новую программу и находя чередующуюся цепь с наименьшим весом, начинающуюся во вновь добавленной программе. Нахождение чередующейся цепи удобно производить с помощью алгоритма Форда-Беллмана, которому для этого потребуется O(i) итераций нахождения. При этом полезно заметить, что в наилучшей чередующейся цепи могут быть задействованы не более m+i–1 положений (i–1 уже занятых и m свободных), что позволяет построить алгоритм, время выполнения которого порядка O($n^2(m+n)^2$).

Более подробно о графах и алгоритмах на них смотри, например, Кормен, Лейзерсон, Ривест «Алгоритмы: построение и анализ», МЦНМО, Москва, 2000.

регион / программа	1	2	3
1	50	30	30
2	100	25	25
3	∞	∞	19
4	∞	41	41
5	18	18	42

Таблица 1

положение / программа	1,1	1,2	1,3	1,4	1,5	2,1	2,2	2,3	2,4	2,5	3,1	3,2	3,3	3,4	3,5
1	50	100	150	200	250	<u>30</u>	60	90	120	150	30	60	90	120	150
2	100	200	300	400	500	25	<u>50</u>	75	100	125	25	50	75	100	125
3	8	8	8	8	8	8	8	8	8	8	19	<u>38</u>	57	76	95
4	8	8	8	8	8	41	82	123	164	205	<u>41</u>	82	123	164	205
5	<u>18</u>	36	54	72	90	18	36	54	72	90	42	84	126	168	210

Таблица 2

```
1  program AIRPORT;
2  const
3    MAX_N = 24; { Макс. число городов }
4    MAX_C = 20; { Макс. число конфигураций }
5
6  type
7    TOutputLine = record { Строка выходной таблицы }
8      c: integer; { Идент. конфигурации }
9      t: longint; { Общая нагрузка }
10   end;
11
12 var
13   n: integer; { количество городов }
14   load: array[1..MAX_N, 1..MAX_N] of integer;
15   cnt: integer; { количество конфигураций }
16   totals: array[1..MAX_C] of TOutputLine;
17   sw: TOutputLine;
18   arrival: array[1..MAX_N] of integer;
19   departure: array[1..MAX_N] of integer;
20   k, a, b, c, i, j: integer;
21   t: longint;
22
23 begin
24   assign(input, "airport.in");
25   reset(input);
26   while true do begin { Цикл по всем наборам данных }
27     { Читаем транспортную нагрузку }
28     fillchar(load, sizeof(load), 0); { Очистка массива }
29     read(n);
30     if n = 0 then
31       break;
32     for i := 1 to n do begin
33       read(a, k);
34       for j := 1 to k do begin
35         read(b);
36         read(load[a, b]);
37       end;
38     end;
39     { Читаем и обсчитываем каждую конфигурацию }
40     cnt := 0; { Будем считать количество кофигураций }
41     while true do begin { Цикл по конфигурациям }
42       { Читаем конфигурацию }
43       read(c);
44       if c = 0 then
45         break;
46       for i := 1 to n do
47         read(arrival[i]);
48       for i := 1 to n do
49         read(departure[i]);
50       { Обсчитываем конфигурацию }
51       t := 0; { Будем считам общую нагрузку }
52       for i := 1 to n do { Номера приб. входов }
53         for j := 1 to n do { Номера отб. входов }
```

```
54            inc(t, (abs(i - j) + 1) * load[arrival[i], departure[j]]);
55        { Запоминаем результаты в массиве totals }
56        inc(cnt);
57        totals[cnt].c := c;
58        totals[cnt].t := t;
59      end;
60      { Сортируем результаты }
61      for i := 1 to cnt - 1 do
62        for j := i + 1 to cnt do
63          if (totals[i].t > totals[j].t) or
64          ((totals[i].t = totals[j].t) and (totals[i].c > totals[j].c))
65            then begin
66            { Поменяем местами totals[i] и totals[j] }
67            sw := totals[i];
68            totals[i] := totals[j];
69            totals[j] := sw;
70          end;
71      { Выводим результаты }
72      writeln('Configuration  Load');
73      for i := 1 to cnt do
74        writeln(totals[i].c:5, ' ':10, totals[i].t);
75    end;
76  end.
```

Елизаров Роман Анатольевич,
председатель жюри
Северо-Восточного Европейского
региона Всемирных студенческих
соревнований по программированию
(ACM).