Школа современного программирования

Станкевич Андрей Сергеевич

РАЗБОР ЗАДАЧ ФИНАЛА МЕЖДУНАРОДНОГО СТУДЕНЧЕСКОГО ЧЕМПИОНАТА МИРА 2001–2002 гг. ПО ПРОГРАММИРОВАНИЮ ACM

Стиль формулировок условий задач может показаться непривычным читателю, однако он был сохранен при переводе с английского языка. В частности, авторы задач на финале ставят основной целью не художественную целостность задачи, а ее максимальную прозрачность и понятность, в том числе для участников, для которых английский язык не является родным. Этим объясняется слегка «рваный» стиль формулировок.

Одной особенностью данного разбора является полное отсутствие текстов программ (за исключением одной, самой сложной задачи – задачи F). Самостоятельное написание программы по разбору задачи является чрезвычайно полезным упражнением по программированию. Рекомендуем всем увлекающимся олимпиадными задачами выполнить его.

ЗАДАЧА A. ШАРИКИ В КОРОБКЕ.

Вам требуется написать программу, которая будет симулировать процесс упаковки воздушных шариков в коробку.

Опишем этот процесс. Представьте себе, что вам дана коробка в виде прямоугольного параллелепипеда и набор точек внутри нее. Каждая точка представляет собой место, в которое вы можете поместить воздушный шарик. Чтобы сделать это, поместите в нее центр воздушного шарика и надувайте его до тех пор, пока он не коснется коробки или одного из уже надутых воздушных шариков. Вы можете использовать данные вам точки в произвольном порядке и не обязаны использовать все точки. Ваша цель, таким образом разместить шары в коробке, чтобы максимизировать суммарный объем, занимаемый воздушными шарами, и рассчитать, какой объем останется в коробке после такой оптимальной упаковки воздушных шаров.

Входные данные

Входной файл содержит описания нескольких тестовых примеров. Первая строка каждого тестового примера содержит число n ($1 \le n \le 6$) – количество точек. Следующие две строки содержат по три целых числа (x, y, z) – координаты двух противоположных углов коробки. Затем следуют n строк, содержащих по 3 целых числа – (x_i, y_i, z_i) – координаты i-ой точки. Все ребра коробки имеют ненулевую длину и параллельны осям координат.

Завершает входной файл строка, содержащая единственное число 0.

Выходные данные

Для каждого тестового примера выведите одну строку, содержащую номер тестового примера,

после чего следует объем, который остается в коробке после оптимальной упаковки воздушных шаров.

После каждого тестового примера выводите пустую строку.

Пример

Пример входного файла	Вывод для данного входного файла
2 0 0 0 10 10 10 3 3 3 7 7 7 0	Box 1: 774

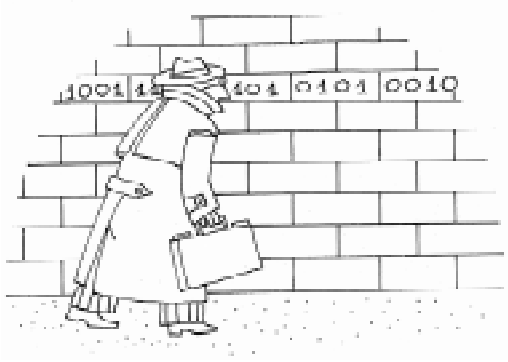
Решение.

Эта задача имеет несложное переборное решение. А именно, будем перебирать все перестановки точек, соответственно тому, в каком порядке мы будем добавлять шары в точки. Если очередная точка находится внутри шара, то надувать в ней шар, очевидно, бессмысленно. Пусть в некотором оптимальном решении некоторые точки не используются. Расположим точки следующим образом: сначала точки, в которых следует надуть шары в том же порядке, что и в оптимальном решении, затем точки, которые не были использованы, – в произвольном порядке. Ясно, что при этом первые шары будут такими же, как и в оптимальном решении, а если в точках, которые мы расположили в конце, также можно надуть шары ненулевого радиуса, то мы получим решение лучше нашего исходного, что противоречит его оптимальности. Значит, можно надувать шары во всех возможных точках, не пропуская их, кроме случаев, когда точка уже находится в шаре.

Посмотрим теперь, как выяснить, какого размера шар можно надуть в данной точке. Прежде всего, вычислим минимальные и максимальные координаты коробки по каждой из осей. Если точка находится за пределами коробки (хотя бы одна из ее координат не находится между соответствующей минимальной и максимальной координатой), то надувать шар в этой точке вообще бессмысленно. В противном случае, радиус шара есть минимум среди следующих величин: $x_i - x_{min}$, $x_{max} - x_i$, $y_i - y_{min}$, $y_{max} - y_i$, $z_i - z_{min}$, $z_{max} - z_i$, $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} - r_j$, где (x_i, y_i, z_i) – координаты текущей точки, а j пробегает множество номеров точек, в которые мы уже добавили шары. Если получившийся радиус меньше нуля, то точка лежит либо вне коробки, либо внутри уже надутого шара, следовательно, ее мы пропускаем.

Организация перебора с возвратом не представляет проблемы. Чтобы сосчитать ответ, надо сложить объемы всех получившихся шаров и вычесть их сумму из объема коробки. Напомним, что объем коробки есть $(x_{max} - x_{min})(y_{max} - y_{min})(z_{max} - z_{min})$, а объем шара $\frac{4}{3}\pi r_i^3$.

ЗАДАЧА B. НЕПРЕФИКСНЫЕ КОДЫ



Фил Оракул обладает уникальной способностью, благодаря которой его до сих пор не уволили из Центрального Разведывательного Управления. Его коллеги могут принести ему любой двоичный код, и он мгновенно может сказать им, является ли этот код однозначно декодируемым. *Код* – это сопоставление некоторой уникальной последовательности символов (*кодовой последовательности*) каждому символу в заданном *алфавите*. Двоичный код – это код, в котором кодовые последовательности состоят только из нулей и единиц. Приведем в качестве примера два двоичных кода для алфавита {a, c, j, l, p, s, v}:

	Код 1	Код 2
a	1	010
c	01	01
j	001	001
l	0001	10
p	00001	0
s	000001	1
v	0000001	101

Для строки (*сообщения*), состоящей из символов нашего алфавита, *закодированным сообщением* называется конкатенация кодовых последовательностей символов, из которых она состоит, по порядку слева направо. Код называется *однозначно декодируемым*, если для произвольных двух различных сообщений соответствующие им закодированные сообщения также являются различными. Так, в рассмотренном нами примере первый код является однозначно декодируемым, а второй – нет, так как в нем, к примеру, слова pascal и java кодируются одной и той же последовательностью символов – 001010101010. И даже такие короткие закодированные сообщения, как 01 или 10, не являются однозначно декодируемыми.

И хотя управление очень сильно гордится Филом, он дает исключительно ответы «да» и «нет». В то же время некоторые члены управления хотели бы увидеть некоторое подтверждение слов Фила, особенно для кодов, которые не являются однозначно декодируемыми. Так, в этой задаче вы будете иметь дело *только* с кодами, которые не являются однозначно декодируемыми. Для каждого такого кода вам следует найти закодированное сообщение, имеющее минимальную длину (в битах), которое не является однозначно декодируемым, поскольку могло получиться в результате кодирования, по крайней мере, двух различных сообщений. Если существует несколько таких закодированных сообщений, следует вывести то, которое идет раньше в лексикографическом порядке.

Входные данные

Входной файл содержит несколько кодов, которые следует протестировать. Сначала на отдельной строке следует число m ($1 \le m \le 20$) – количество символов в алфавите, для которых задан код (сами символы алфавита в этой задаче интереса не представляют). Затем следуют m строк, каждая из которых содержит кодовую последовательность, перед и после которой могут идти пробелы. Ни одно кодовое слово не содержит более 20 бит.

После последнего тестового примера идет строка, содержащая единственное число 0.

Выходные данные

Для каждого кода выведите сначала номер тестового примера, затем длину кратчайшей неоднозначно декодируемой кодовой последовательности и, наконец, саму последовательность. Выводите кодовую последовательность по 20 бит на строке, кроме последней, которая может содержать менее 20 бит. После каждого кода выводите пустую строку.

Пример

Пример входного файла	Вывод для данного входного файла
3 0 01 10 5 0110 00 111 001100 110 5 1 001 0001 00000000000000000001 10000000000000000000 0	Code 1: 3 bits 010 Code 2: 9 bits 001100110 Code 3: 21 bits 10000000000000000000 1

Решение.

Для решения этой задачи применим динамическое программирование. Пусть $a[i][j]$ – длина наиболее короткой битовой последовательности, которая может быть получена двумя следующими способами: либо в виде конкатенации некоторой последовательности кодовых последовательностей символов (то есть является некоторым закодированным сообщением), либо удалением первых j бит у некоторого закодированного сообщения, для которого исходное сообщение начинается с i-го символа алфавита.

Чтобы понять смысл $a[i][j]$, обратимся к рисунку 1.

Снизу изображено некоторое закодированное сообщение. Оно совпадает с закодированным сообщением, изображенным сверху, если у него удалить первые j бит. Минимально возможная длина такого сообщения и есть $a[i][j]$. При этом сообщение сверху начинается с i-го символа алфавита. Единственное дополнение – если $j = 0$, то потребуем, чтобы нижнее сообщение не начиналось с i-го символа.

Пусть также $b[i][j]$ – первая в лексикографическом порядке битовая строка, которая удовлетворяет такому условию (то есть $b[i][j]$ – наименьшая возможная в лексикографическом порядке для данных i и j нижняя строка на нашем рисунке).

Если таких строк не существует, то положим $a[i][j] = +\infty$. Ясно, что ответ на нашу задачу представляет $b[i][0]$ для такого i, для которого $a[i][0]$ – минимальное возможное, а если таких несколько, то наименьшее в лексикографическом порядке из подходящих $b[i][0]$.

Посмотрим теперь, как получить $a[i][j]$ и $b[i][j]$ для всех необходимых нам i и j. Используем для их вычисления динамическое программирование «сверху вниз» или так называемый метод «ленивых вычислений». Его суть заключается в том, что мы создаем искомую таблицу $a[i][j]$ и доступ к ней обеспечиваем с помощью специальной рекурсивной функции, назовем ее $calca(i, j)$. Эта функция сначала проверяет, не было ли уже вы-

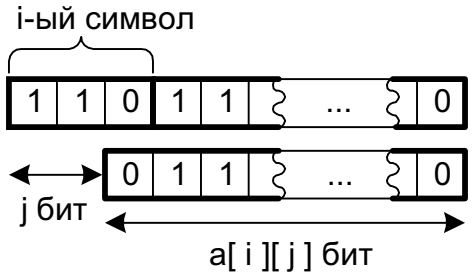


Рисунок 1.

числено значение $a[i][j]$. Если оно уже известно, то она просто возвращает готовое значение. В противном случае она вычисляет $a[i][j]$ по известному рекуррентному соотношению (которое мы сейчас выведем), сохраняет его в массиве и возвращает его. В следующий раз, когда нам потребуется значение $a[i][j]$, оно не будет повторно вычисляться, и будет возвращено уже готовое значение. Отметим, что, в отличие от динамического программирования «снизу вверх», при котором каждое значение вычисляется после того, как все необходимые для него элементы уже вычислены, здесь обращение к элементу может произойти до того, как все необходимые элементы будут вычислены, в таком случае они будут вычислены рекурсивно. Это особенно полезно в таких задачах, как наша, в которых порядок вычисления элементов заранее предсказать сложно. Свое название «ленивые вычисления» этот метод получил по той причине, что ни одно значение не будет вычислено до тех пор, пока оно не понадобится.

Итак, все что осталось, – это записать рекуррентное соотношение для $a[i][j]$ и указать метод нахождения $b[i][j]$. Обозначим кодовую последовательность для k-го символа за $s[k]$, а ее длину за $l[k]$.

Тогда $a[i][j] = \min_k f(i, j, k)$, где k пробегает номера таких символов, что либо кодовая последовательность соответствующего символа является префиксом $s[i][j+1 .. l[i]]$, либо $s[i][j+1 .. l[i]]$ является ее префиксом. В случае $j = 0$ дополнительно потребуем $k \neq i$. Функция $f(i, j, k)$ возвращает следующее значение, в зависимости от длин строк:

если $l[i] - j = l[k]$, то $f(i, j, k) = l[k]$;

если $l[i] - j > l[k]$, то

$$f(i, j, k) = a[i][j + l[k]] + l[k];$$

если $l[i] - j < l[k]$, то

$$f(i, j, k) = a[k][l[i] - j] + l[i] - j.$$

Соответственно, в качестве $b[i][j]$ следует среди всех вариантов, для которых реализуется минимум, выбрать первое в лексикографическом порядке значение из соответствующих $g(i, j, k)$, где

если $l[i] - j = l[k]$, то $g(i, j, k) = s[k]$;

если $l[i] - j > l[k]$, то

$$g(i, j, k) = s[k] + b[i][j + l[k]];$$

если $l[i] - j < l[k]$, то

$$g(i, j, k) = s[i][j + 1..l[i]] + b[k][l[i] - j].$$

Отметим один интересный момент, связанный с этой задачей. Условие однозначного декодирования битовых последовательностей тесно связано с понятием так называемых *префиксных* кодов. Код называется префиксным, если для любых двух различных кодовых последовательностей ни одна из них не является префиксом другой. Хорошо известен факт, что префиксный код всегда является однозначно декодируемым. Следовательно, чтобы код не был однозначно декодируемым, необходимо, чтобы он не был префиксным. Заметим, однако, что код не обязан быть префиксным, чтобы быть однозначно декодируемым. Тривиальный пример представляет собой следующий код для двухсимвольного алфавита: s[1] = 1, s[2] = 10. Закодированные сообщения для любых двух исходных сообщений различаются. Более подробно о префиксных кодах можно прочитать в [3] и в [4].

ЗАДАЧА C. ЧЕРЕЗ ПУСТЫНЮ

В этой задаче вам предстоит рассчитать, каким количеством провизии следует запастись, чтобы пересечь пустыню пешком.

В вашем стартовом пункте вы можете купить провизию в магазине и набрать неограниченное количество воды. В пустыне в различных точках могут располагаться оазисы. В каждом оазисе вы можете набрать сколько угодно воды, а также можете оставить запас еды на будущее, но не можете докупить еды. Вы также можете оставить еду на будущее в вашем стартовом пункте. Вам даны координаты на плоскости вашего стартового пункта, всех оазисов и вашего пункта назначения. Единицей измерения является миля.

За каждую пройденную милю вы должны съесть одну условную единицу провизии и выпить одну условную единицу воды. Будем предполагать, что запасы расходуются непрерывно, то есть, пройдя часть мили, вы должны съесть соответствующую часть провизии и выпить соответствующую часть воды. Если же у вас кончается запас провизии или воды, то вы не можете больше никуда идти. Во время отдыха в оазисе вы не расходуете ваши запасы вообще. Разумеется, имеется максимальное суммарное количество провизии и воды, которое вы можете нести. Это максимальное количество измеряется в условных единицах, которые будем считать одинаковыми для пищи и воды. Ни в какой момент вы не можете пройти какое бы то ни было расстояние, перенося суммарно провизии и воды больше, чем это количество.

Вам следует решить, какое минимальное количество провизии следует купить в стартовом пункте, чтобы добраться до конечного пункта. Не требуется, чтобы у вас осталась провизия или вода, когда вы доберетесь до конечного пункта. Поскольку магазин продает провизию только в целых условных единицах и поскольку на складе в магазине имеется только один миллион условных единиц провизии, то вам

следует купить количество провизии, которое было бы целым числом от единицы до одного миллиона.

Входные данные

Входной файл содержит несколько тестовых примеров.

Первая строка каждого примера содержит n ($1 \le n \le 20$) – общее количество особых точек в пустыне, после которого следует максимальное суммарное количество провизии и воды, которое вы можете нести. Следующие n строк содержат координаты особых точек в пустыне – пары целых чисел. Первая особая точка – это ваш стартовый пункт, последняя – ваш пункт назначения, остальные (если они есть) представляют собой оазисы. Вам не обязательно посещать каждый оазис, вы должны посещать оазисы, только если это помогает вам добраться до пункта назначения с меньшим количеством провизии.

Последняя строка входного файла содержит два нуля.

Выходные данные

Для каждого примера выведите его номер и затем количество условных единиц провизии, которое необходимо закупить, чтобы осуществить путешествие. Если при заданных условиях осуществить путешествие нельзя, выведите, вместо количества провизии, слово Impossible.

После каждого тестового примера выводите пустую строку.

Пример

Пример входного файла	Вывод для данного входного файла
4 100 10 -20 -10 5 30 15 15 35 2 100 0 0 100 100 0 0	Trial 1: 136 units of food Trial 2: Impossible

Решение.

Приведем решение этой задачи без доказательства, оставив его читателю как неплохое упражнение. Доказательство аналогично доказательству корректности работы алгоритма Дейкстры.

Рассмотрим граф, вершинами которого будут особые точки в пустыне (оазисы и пункты старта и назначения). Ребрами графа будут переходы между соответствующими точками. Введем функцию $d(v)$ на множестве вершин нашего графа. Пусть $d(v)$ – минимальное количество провизии, которое надо иметь в точке v, чтобы добраться до пункта назначения. Тогда $d(n) = 0$, а $d(1)$ – ответ на нашу задачу.

Как и в алгоритме Дейкстры, разделим вершины на три множества. В множестве U будут вершины, для которых $d(v)$ уже вычислено. В множестве P – вершины, для которых известно некоторое приближение $d(v)$, и в множестве W – вершины, для которых $d(v)$ неизвестно.

Шаг алгоритма будет переводить вершину из множества P в множество U. Если множество P пусто, то алгоритм заканчивает свою работу. Выберем в множестве P вершину v, для которой текущее значение $d(v)$ (напомним, что, вообще говоря, оно может быть еще не окончательным) минимально. Переместим эту вершину в множество U. Рассмотрим теперь все вершины графа из множеств W и P. Для каждой вершины u вычислим $d_v(u)$ – какое количество провизии должно быть у нас в вершине u, чтобы, переместившись из нее в вершину v, добраться затем до пункта назначения. Напомним, что, поскольку вершина v уже находится в множестве U, то для нее величина $d(v)$ окончательно известна.

Обозначим за $f(u, v)$ расстояние в милях от u до v. Пусть также c – максимальное суммарное количество воды и провизии, которое мы можем нести. Возможны два варианта. Если $2f(u, v) + d(v) \le c$, то можно выйти из точки u с необходимым количеством провизии и воды, чтобы добраться до v и затем до пункта назначения. В этом случае $d_v(u) = d(v) + f(u, v)$.

Если $2f(u, v) + d(v) > c$, то мы не можем сразу перенести всю провизию из u в v. Придется совершить несколько рейдов туда-обратно. Посмотрим, какое количество провизии мы можем перенести. Нам надо дойти из u до v и затем обратно. Кроме того, нам надо взять воды на дорогу в один конец. Значит, на «накладные расходы» уйдет $2f(u, v)$ провизии на каждый поход туда-обратно. Кроме того, каждый раз нам надо нести воду для пути туда (воду для пути обратно мы наберем в v). Значит, остаток нашей «нагрузочной способности» есть $c - 3f(u, v)$. Заметим также, что в последнем случае мы можем нести дополнительно $f(u, v)$ провизии, поскольку нам не требуется питание на обратную дорогу.

Если $3f(u, v) > c$, то полагаем $d_v(u) = +\infty$, поскольку осуществить требуемое нам не удастся. Иначе нам придется совершить k переходов туда–обратно, где k – минимальное число, такое, что $k(c - 3f(u, v)) + (c - 2f(u, v)) \geq d(v)$, то есть

$$k = \left\lceil \frac{d(v) - c + 2f(u,v)}{c - 3f(u,v)} \right\rceil .$$

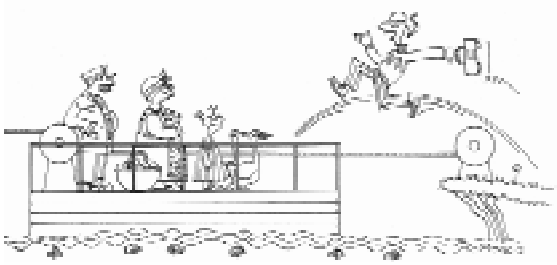
Тогда полагаем $d_v(u) = (2k + 1) f(u, v) + d(v)$.

После того, как $d_v(u)$ вычислено, полагаем $d(u) = \min\{d(u), d_v(u)\}$. Если при этом $d(u)$ стало меньше $+\infty$, то переводим вершину из множества W в множество P.

Отметим, что если некоторое $d(v)$ стало больше 10^6, то можно считать его равным $+\infty$, поскольку столько провизии все равно на складе нет. Итак, если в конце $d(1) < +\infty$, то это и есть ответ, в противном случае добраться нельзя.

Что касается доказательства корректности данного алгоритма, то оно, как уже упоминалось, остается в качестве упражнения. Доказать, что в рамках данной стратегии перемещений полученный ответ является верным, несложно. При доказательстве оптимальности приведенной стратегии следует воспользоваться неравенством треугольника и свести любую другую стратегию к предложенной с неувеличением количества требуемой провизии.

ЗАДАЧА D. ПАРОМЫ



Миллионы лет назад огромные ледяные поля покрывали горы Норвегии, и под тяжестью льда в земле образовались глубокие пещеры. Море наполнило их водой. Так появились заливы, которые норвежцы называют фьордами. Великолепные ландшафты поражают своей красотой, но фьорды мешают передвижению местных жителей. Наиболее типичная схема перемещения местных жителей из одного места в другое выглядит так: ехать на максимальной скорости (80 км/ч) до ближайшего парома, подождать пока он подойдет, переправиться. Повторять, пока не достигнешь пункта назначения.

Поскольку езда на максимальной скорости требует больше топлива, чем более медленная езда, такая стратегия, кроме всего прочего, оказывается дорогой и опасной для окружающей среды. Так что власти решили установить для помощи местным жителям специальную систему помощи в перемещениях. Зная, каким путем вы собираетесь ехать, система собирает информацию о расписании паромов, вычисляет минимальное время, которое необходимо для того, чтобы добраться до цели, и предлагает такую скорость передвижений, чтобы не приходилось ехать быстрее, чем требуется. Так, система рассчитывает скорость, исходя из того, что следует заехать на паром ровно в тот момент, когда он отходит.

Вам дан путь (последовательность дорог и паромов). Напишите программу, которая вычислит минимальное время, необходимое, чтобы проехать по этому пути. Кроме того, ваша программа должна найти такой способ проезда по этому пути, чтобы максимальная скорость вашего передвижения по пути была как можно меньше.

Входные данные

Входной файл содержит несколько тестовых примеров. Каждый тестовый пример описывает один путь. Путь состоит из нескольких участков, каждый из которых – это либо отрезок дороги, либо переправа на пароме. Первая строка тестового примера содержит число s ($s > 0$) – количество участков пути. Следующие s строк содержат описание участков. Каждая строка начинается с двух названий – пункт отправления и пункт назначения участка, после которых идет либо слово **road** либо слово **ferry**, что означает *дорога* или *паром*, соответственно. Если участок представляет собой дорогу, то после этого следует длина дороги в километрах (положительное целое число). Например,

```
Dryna Solholmen road 32
```

Строки, которые описывают паромы, содержат больше информации. После слова ferry следует время переправы в минутах (положительное целое число), затем идет f – количество отправлений парома в час и, наконец, f чисел – время в минутах от начала часа, в которое отправляется паром, в возрастающем порядке, например:

```
Manhiller Fodnes ferry 20 2 15 35
```

Паром идет из Манхиллера в Фоднес 20 минут, отправляется два раза в час (в 0ч15м, 0ч35м, 1ч15м, 1ч35м, ...). Начало вашего пути происходит в некоторый полный час. Участки пути даны последовательно, то есть если участок идет из A в B, то следующий участок идет из B. По каждому пути можно проехать за 10 часов.

Входной файл завершается строкой, содержащей единственное число – 0.

Выходные данные

Для каждого тестового примера выведите одну строку, содержащую три элемента. Сначала выведите номер тестового примера, затем – общее время путешествия в формате hh:mm:ss и максимальную скорость, которая требуется в некоторый момент в оптимальной поездке (там, где эта величина минимальна), округленную до 2 знаков после запятой.

После каждого тестового примера выводите пустую строку.

Пример

Пример входного файла
1 Bygd Bomvei road 7 2 Ferje Overfarten ferry 20 2 5 25 Overfarten Havneby ferry 30 3 10 30 50 5 Begynnelse Brygge road 30 Brygge Bestemmelse ferry 15 4 10 25 40 55 Veiskillet Grusvei road 25 Grusvei Slutt ferry 50 1 10 0
Вывод для данного входного файла
Test Case 1: 00:05:15 80.00 Test Case 2: 01:00:00 0.00 Test Case 3: 03:00:00 45.00

Решение.

Интересно, но эта несложная задача имеет два принципиально различных решения, основанных на совершенно различных идеях. Одно из них использует стандартное динамическое программирование, а другое – двоичный поиск по ответу.

Рассмотрим оба этих решения.

Итак, начнем с динамического решения. Обозначим за $a[i][t]$ минимально возможную максимальную скорость, необходимую для достижения i-го промежуточного пункта за t минут (занумеруем пункты от 0 до s). Отметим, что нам достаточно варьировать t от 0 до 600. Тогда $a[0][t] = 0$ для всех t. Пусть теперь $i > 0$. Рассмотрим два варианта. Пусть i-ый участок пути – дорога. Тогда $a[i][t] = \min_{t'<t}\left(\max(a[i-1][t'], \frac{l[i]}{t-t'})\right)$, где $l[i]$ – длина нашей дороги. Если минимум берется по пустому множеству, то положим $a[i][t] = +\infty$. Если же i-ый участок пути –

переправа, то $a[i][t] = \min_{t \le f(t)} a[i-1][t']$, где функция $f(t)$ определена следующим образом: $f(t)$ – максимальный момент времени, в который можно подъехать к переправе, чтобы успеть на паром таким образом, чтобы оказаться на другой стороне в момент t. Написание функции f по описанию паромов не представляет особого труда, следует найти последний по времени момент прибытия парома до времени t и вычесть из этого значения время переправы.

Заметим, что функция $a[i][t]$ является невозрастающей по второму аргументу (так как всегда можно приехать позже и не ехать быстрее). Поэтому можно отбросить минимум в последней формуле, получив $a[i][t] = a[i - 1][f(t)]$.

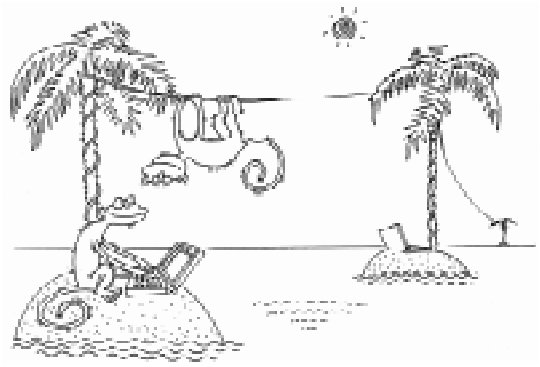
Посмотрим теперь, как извлечь из массива ответ на наши два вопроса – о минимальном времени, за которое можно добраться до пункта назначения, и минимально возможной максимальной скорости для этого. Необходимое время – это $t_0 = \min\{t : a[s][t] < +\infty\}$. Соответственно, максимальная скорость, по определению нашего массива, есть $a[s][t_0]$.

Рассмотрим теперь еще один элегантный способ решения нашей задачи. Пусть мы знаем максимальную скорость движения. Ясно, что при ее увеличении минимальное время нашего прибытия не увеличится, а при уменьшении – не уменьшится. Значит, функция $t_0(v)$ – минимальное время прибытия в зависимости от максимальной скорости на пути является невозрастающей.

Научимся считать эту функцию. Это несложно – в силу того, что все паромы на одной переправе имеют одно и то же время в пути, всегда выгодно садиться на первый же паром. Значит, оптимальной будет следующая стратегия: ехать на скорости v до ближайшей переправы, ждать парома, переправляться Повторять, пока не доберемся до пункта назначения.

Итак, теперь нам надо найти минимальное значение аргумента функции $t_0(v)$, при котором она достигает своего минимального значения. Отметим, что по условию $0 < v \le 80$. Ясно, что в силу своего невозрастания $t_0(80) \le t_0(v)$ для всех $v \le 80$, то есть нам известно минимальное значение t_0 – это $t_0(80)$. Теперь осталось с помощью двоичного поиска найти минимальное значение v, при котором достигается это значение.

ЗАДАЧА Е.
ИНТЕРНЕТ НА КАЖДЫЙ ОСТРОВ!



Компания «Тихоокеанские Островные Сети» (ТОС) недавно обнаружила в Тихом океане группу островов, которые до сих пор не подключены к Интернет по выделенному каналу. ТОС планирует захватить этот потенциальный рынок сбыта, предложив островитянам свои услуги. Главный остров архипелага уже имеет выход в Интернет с помощью спутников ТОС, так что все, что требуется, – это соединить острова в группе с главным островом с помощью кабелей. Ваша цель – помочь им разработать план по соединению. Для каждого острова известна позиция его роутера – точки подключения к кабелю и количество жителей острова – будущих потребителей Интернет.

Так, на рисунке 2 черные точки – роутеры, а числа рядом с островами – количество жителей острова. ТОС планирует проложить кабели между некоторыми роутерами таким образом, чтобы от каждого роутера был путь до главного острова. ТОС планирует проложить кабели так, чтобы их суммарная длина была минимальна. Несмотря на это ограничение, может

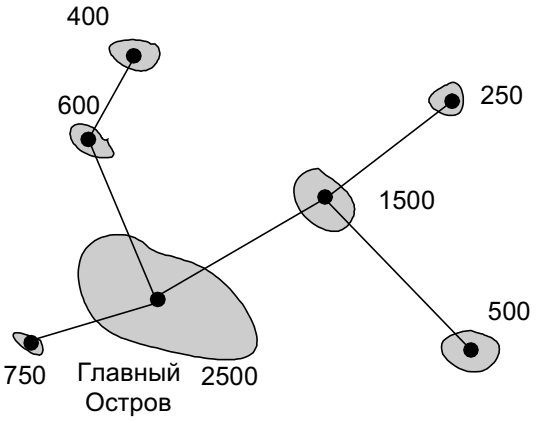


Рисунок 2.

быть несколько возможных планов соединения островов. В этом случае ТОС не важно, какой из них реализовать.

ТОС интересует среднее время, в течение которого ее новым клиентам придется ждать подключения, исходя из предположения, что прокладка всех кабелей из плана начнется одновременно. Кабели будут прокладываться со скоростью 1 километр за день. В результате более короткие кабели будут проложены раньше, чем более длинные. Остров получает доступ к Интернет, как только появляется путь по кабелям от него до главного острова. Если m_i – количество жителей i-го острова и t_i – время, которое требуется, чтобы подключить остров к Интернет, то среднее время подключения есть $\frac{\sum t_i m_i}{\sum m_i}$.

Входные данные

Входной файл содержит несколько тестовых примеров. Первая строка каждого описания группы островов содержит целое положительное число n – количество островов в группе ($n \le 50$). Каждая из следующих n строк содержит по три целых числа – x_i, y_i и m_i – координаты роутера и количество жителей i-го острова ($m_i > 0$). Координаты измеряются в километрах. Первый остров в группе – это главный остров.

Последняя строка входного файла содержит единственное число 0.

Выходные данные

Для каждой группы островов выведите порядковый номер группы и среднее количество дней, которое островитянам придется ждать подключения к Интертнет. Количество дней должно быть выведено с двумя знаками после запятой.

После каждого тестового примера выводите пустую строку.

Пример

Пример входного файла
7 11 12 2500 14 17 2500 9 9 750 7 15 600 19 16 500 8 18 400 15 21 250 0
Вывод для данного входного файла
Island Group: 1 Average 3.20

Решение.

Прежде всего заметим, что нам достаточно найти $\sum t_i m_i$, поскольку подсчитать $\sum m_i$ не представляет никакого труда. Итак, будем искать эту сумму.

На первый взгляд кажется, что ответ определен неоднозначно. Действительно, требование, предъявляемое к плану соединений, есть ни что иное, как требование построить минимальный остов в полном евклидовом графе, вершинами которого являются роутеры на островах.

Напомним, что взвешенный граф называется евклидовым, если его вершины можно разместить на плоскости так, чтобы вес любого его ребра был равен расстоянию между вершинами, которые оно соединяет как точки плоскости. В нашем случае граф уже размещен на плоскости. Остовом графа называется минимальный по включению его связный подграф. Для связного исходного графа (а полный граф, безусловно, связен) остов является деревом (для несвязного – лесом) –

в этом случае его обычно называют остовным деревом. Остов взвешенного графа называется минимальным, если сумма весов входящих в него ребер минимальна среди всех возможных остовов графа (ибо остовов, разумеется, может быть много; так, в полном графе существует N^{N-2} различных остова – эту формулу часто называют теоремой Кэли, ее доказательство можно найти, например, в [2]).

В дальнейшем нам также понадобится понятие разреза. Разрезом называют такое множество C ребер, что вершины графа можно разбить на два непересекающихся множества S и T таких, что ребро принадлежит C тогда и только тогда, когда один его конец принадлежит S, а другой – T (то есть в C вошли все ребра, соединяющие S и T, и никакие другие).

Существует большое количество алгоритмов построения минимального остова, наиболее известными из которых являются алгоритмы Прима и Краскала (мы рассмотрим их далее в разборе). Однако известно, что если в графе есть ребра одинакового веса, то у него может быть несколько минимальных остовов. Примером может служить граф в виде правильного треугольника – все три его остова имеют равный вес и потому являются минимальными.

Таким образом, на первый взгляд кажется, что сумма, которую нам требуется найти, может зависеть от конкретного остова, ведь время подключения островов может зависеть от того, какой остров мы выбрали. Однако оказывается, что это не так, – время подключения не зависит от конкретного минимального остова. Заметим, что время подключения острова есть вес максимального ребра на пути по построенному остовному дереву от рассматриваемого острова до главного. Для каждой пары u и v вершин графа и минимального остовного дерева T определим величину $f(u, v, T)$ как вес наибольшего ребра на простом пути из u в v по дереву T. Докажем, что величина $f(u, v, T)$ не зависит от T, и для любых двух вершин u и v и любых двух минимальных остовных деревьев T_1 и T_2 выполняется равенство $f(u, v, T_1) = f(u, v, T_2)$.

Действительно, рассмотрим две вершины u и v и какое-нибудь минимальное остовное дерево T. Докажем, что не существует другого минимального остовного дерева T^*, для которого $f(u, v, T^*) < f(u, v, T)$. Рассмотрим максимальное ребро e на пути из u в v по T, пусть его вес w. Удалим его из дерева. Дерево распадется на две компоненты связности, одна из которых содержит u, а другая v, обозначим их как T_u и T_v, соответственно. Если мы теперь удалим в исходном графе все ребра, которые соединяли вершины из T_u и T_v, то и исходный граф распадется на две компоненты связности. Обозначим их за U и V, соответственно (отметим, что, поскольку дерево распалось на две компоненты, граф не может распасться более чем на две компоненты – все ребра, которые остались в дереве, остались и в графе). Рассмотрим теперь все ребра, идущие из U в V. Заметим, что они формируют разрез нашего графа, причем все пути из u в v пересекают этот разрез.

Но в нем не может быть ребра с весом меньше w, поскольку в таком случае, заменив в дереве T ребро e на ребро меньшего веса, мы получили бы остовное дерево меньшего веса, что противоречит минимальности T. Значит, наибольшее ребро на пути из u в v вообще не может быть меньше w.

Осталось показать, что не существует также минимального остовного дерева T^*, для которого $f(u, v, T^*) > f(u, v, T)$. Но это непосредственно следует из доказанного – ведь для такого дерева наше дерево T было бы деревом, в котором наибольший вес ребра на пути между u и v меньше, а мы уже доказали, что такого не бывает.

Итак, мы выяснили, что компании действительно все равно, какой способ оптимальной прокладки кабелей выбрать, – ведь среднее время подключения клиентов не зависит от конкретного минимального остовного дерева.

Таким образом, достаточно построить произвольное минимальное остовное дерево в нашем графе. После этого вычисление искомого результата по данной формуле не представляет особого труда. Из формулы Кэли следует, что общее их количество может при наших ограничениях достигать 50^{48}, так что полным перебором всех деревьев искать минимальное дерево нерационально. К счастью, существует большое количество полиномиальных алгоритмов построения минимального остовного дерева, два наиболее распространенных из которых мы сейчас рассмотрим.

Практически все алгоритмы построения минимального остовного дерева основаны на применении следующей теоремы, которую часто называют теоремой о разрезе и минимальном остовном дереве.

Пусть выбрано некоторое множество S ребер графа, которое является подмножеством множества ребер некоторого минимального остовного дерева (то есть множество S можно дополнить до некоторого минимального остовного дерева). Назовем ребро $e \notin S$ безопасным для множества S, если $S \cup \{e\}$ также является подмножеством множества ребер некоторого минимального остовного дерева нашего графа (то есть можно добавить e в множество S, и полученное множество также можно будет дополнить до некоторого минимального остовного дерева). Пусть C – разрез, такой, что $C \cap S = \varnothing$ и e – одно из минимальных по весу ребер в C. Тогда e безопасно для S.

Доказательство этой теоремы можно найти, к примеру, в [1].

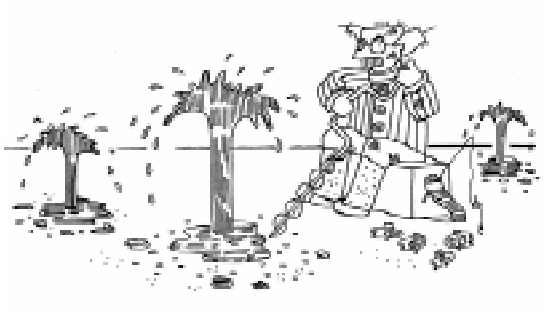
Алгоритмы построения остовного дерева действуют в соответствии с этой теоремой – они начинают с пустого множества, которое без сомнения можно дополнить до минимального остовного дерева, и поочередно добавляют в текущее множество минимальное ребро из некоторого разреза. Различие в алгоритмах заключается в том, какой разрез используется.

Алгоритм Прима в качестве разреза использует множество ребер, один из концов которого принадлежит множеству вершин, достижимых по текущему остовному дереву из первой вершины, а другой – нет. Так, сначала это множество ребер, инцидентных первой вершине. Минимальное из них по теореме безопасно для пустого множества, добавим его в наше остовное дерево. Теперь множество вершин, достижимых из первой вершины, состоит из двух вершин. Наш разрез – это множество ребер, инцидентных одной из них, но не инцидентных другой. Найдем теперь в нем минимальное ребро. Продолжая таким образом, постепенно построим минимальное остовное дерево. Действительно, по приведенной теореме наше множество ребер в любой момент является подмножеством ребер некоторого минимального остовного дерева. Каждый шаг добавляет в него по ребру. Когда ребер будет $n - 1$, где n – количество вершин в графе, они будут стягивать n вершин, то есть будут остовным деревом нашего графа.

Алгоритм Краскала внешне использует более нетривиальный разрез, но при этом оказывается даже проще алгоритма Прима. Пусть S – множество ребер, которое мы дополняем до минимального остовного дерева. Найдем минимальное ребро e в графе, концы которого не соединены путем в S. Тогда существует разрез, содержащий e и не пересекающийся с S (достаточно взять все ребра, не входящие в S, и найти минимальное по включению множество, содержащее e, такое, что при его удалении граф распадается на компоненты связности). Ясно, что в этот разрез не входит множество ребер, концы которых соединены путем в S, значит, e минимально и в этом разрезе, следовательно, оно безопасно для S. Отметим, что нам не требуется искать сам этот разрез, достаточно его существования. Отсюда получаем следующий алгоритм. Начнем с пустого множества. Найдем минимальное ребро в графе. Добавим его в S. Найдем минимальное ребро в графе, концы которого не соединены путем в S. Добавим его в S. Продолжаем так, пока не получится минимальное остовное дерево.

Отметим, что оба алгоритма являются по сути жадными (они делают на каждом шаге выбор локального минимума, то есть берут минимальное ребро в некотором классе, которое не нарушает ациклличности текущего предостова). Возможность применения жадных алгоритмов для решения этой задачи не случайна. Дело в том, что ациклические подмножества ребер неориентированного графа образуют семейство независимых множеств матроида, который называют также графовым или циклическим матроидом. Остовные же деревья являются базами этого матроида. Более подробно о матроидах и применимости для них жадных алгоритмов можно прочитать в [1] или в [2].

ЗАДАЧА F. ВСЕ РАДИ НЕФТИ



Исследование новых месторождений нефти превратилось в высокотехнологичную отрасль промышленности. С новыми технологиями бурения становится экономически выгодно добывать нефть из все меньших и более глубоко залегающих месторождений. Однако исследование таких месторождений с помощью пробного бурения неэффективно, поэтому исследователи разработали методы косвенного определения наличия нефти.

Один из таких методов обнаружения нефти – ультразвук, волны которого отражаются от стен подземных пещер. Определить, сколько нефти содержится в такой пещере – сложная проблема.

В этой задаче вам будет дано некоторое сечение подземной пещеры, представленное в виде многоугольника, наподобие тех, что показаны на рисунке 3. Некоторые точки на границе многоугольника могут быть дырками, через которые нефть уходит в окружающую породу (на рисунке они изображены как черные точки). Имея информацию о многоугольнике и таких точках, вы должны определить, какое количество нефти может оставаться в пещере (она показана серым цветом на рисунке). Это количество ограничено тем фактом, что для любого связного участка нефти ее уровень не может быть выше дырки, которой он касается, ибо в этом случае избыток нефти уйдет в окружающую породу.

Входные данные

Входной файл содержит описания нескольких тестовых примеров. Первая строка каждого тестового примера содержит число n ($3 \le n \le 100$) – количество заданных точек на многоугольнике (его вершин и дырок).

Каждая из следующих n строк содержит по три целых числа – x_i, y_i, h_i. Пара (x_i, y_i) задает координаты точки, которые перечислены в порядке передвижения по границе многоугольника против часовой стрелки. Многоугольник не пересекает и не касается себя. Число же h_i равно 1 для дырки, через которую нефть может уходить, и 0 – в противном случае. Направление «вверх» – это положительное направление оси y.

Последняя строка входного файла содержит число 0.

Выходные данные

Для каждого описания выведите номер тестового примера и затем максимальное количество нефти, которое может быть

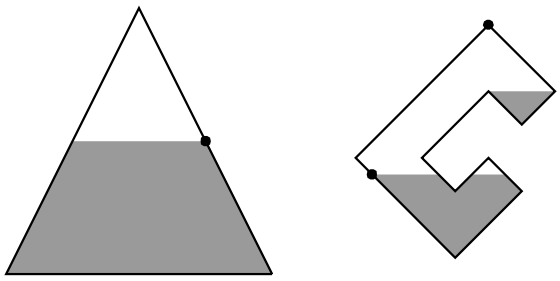


Рисунок 3.

в пещере. Округлите количество до ближайшего целого числа.

После каждого тестового примера выводите пустую строку.

Пример

Пример входного файла	Вывод для данного входного файла
4 10 0 0 5 10 1 0 20 0 -10 0 0 11 0 6 0 1 5 1 6 0 0 10 4 0 8 6 0 6 4 0 4 6 0 8 10 0 10 8 0 12 10 0 8 14 1 0	Cave 1: Oil capacity = 150 Cave 2: Oil capacity = 27

Решение.

Это, пожалуй, самая сложная задача на соревновании и вообще одна из наиболее сложных геометрических задач, предлагавшихся на чемпионате мира. По сложности с ней может сравниться, к примеру, задача про вырубку леса с предыдущего финала или задача из нашего рассматриваемого набора про качение колеса (задача I). Отметим, что ни одна из двух сложных геометрических задач так и не была решена ни одной из команд на соревновании.

Решение этой задачи представляет сложность как с идейной, так и с технической стороны. Рассмотрим основные идеи, которые необходимы для решения этой задачи. Поскольку техническая реализация решения этой задачи также представляет высокую сложность, приведем по мере изложения решения ключевые моменты программы.

Вспомним сначала способ вычисления площади многоугольника, называемый методом трапеций. Он заключается в следующем: площадь многоугольника, заданного последовательностью своих вершин в порядке их обхода против часовой стрелки, есть

$$\frac{1}{2}\sum_{i=1}^{n}(x_{i+1}+x_i)(y_{i+1}-y_i),$$

где предполагается $x_{n+1} = x_1$, $y_{n+1} = y_1$.

В справедливости этой формулы можно убедится, если посмотреть на следующий рисунок 4.

Каждое слагаемое в приведенной формуле есть площадь трапеции, подобной той, которая затенена на рисунке. Площадь берется с отрицательным знаком, если трапеция лежит снаружи от многоугольника, и с положительным – если ее часть, прилегающая к наклонной стороне, лежит внутри многоугольника. Отметим, что горизонтальные стороны многоугольника можно игнорировать при вычислениях по этой формуле.

Проведем горизонтальные линии через все вершины нашего многоугольника и добавим в точках их пересечения с его сторонами дополнительные вершины (см. рисунок 5). Если теперь рассмотреть новый многоугольник, получившийся в результате этой операции, то его площадь можно по-прежнему вычислить по приведенной формуле.

Рассмотрим теперь ту часть многоугольника, которая заполнена нефтью. Заметим, что каждая часть, получившаяся в результате проведенного нами разрезания

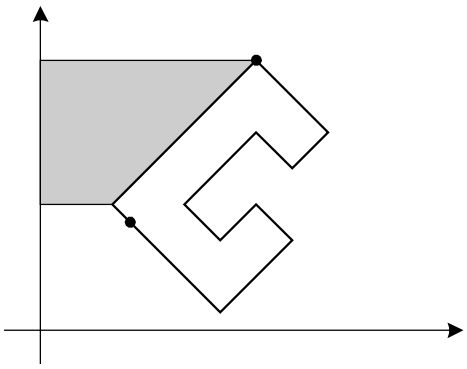


Рисунок 4.

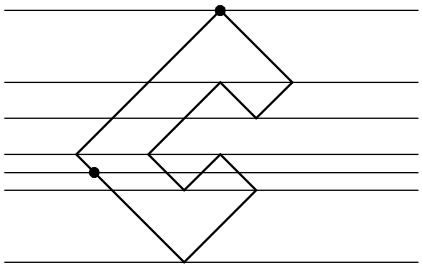


Рисунок 5.

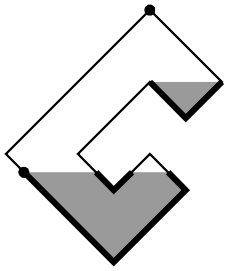


Рисунок 6.

многоугольника горизонтальными линиями, либо полностью заполнена нефтью, либо в ней нефти нет. Если мы научимся определять для заданной точки на границе многоугольника, находится ли нефть в этой точке, то мы сможем определить площадь покрытого нефтью участка. Для этого в формуле вычисления площади достаточно оставить только слагаемые, которые соответствуют ребрам многоугольника, окрестность которых заполнена нефтью. Рисунок 6 демонстрирует такие ребра.

Чтобы проверить, что в окрестности ребра находится нефть, в соответствии со сделанным выше замечанием достаточно проверить, что нефть находится в его середине.

Рассмотрим фрагмент программы, реализующий вычисление искомой площади, исходя из того, что у нас имеется функция *wet*(*x*, *y*), выполняющая необходимую проверку (см. фрагмент 1). Нам также понадобится процедура *ihor*(y, i, x_{res}), которая пересекает i-ую сторону многоугольника с горизонтальной прямой, находящейся на высоте y, и возвращает в переменной x_{res} абсциссу точки пересечения.

Итак, оставшаяся проблема – проверить, что в точке имеется нефть. В соответствии с законом сообщающихся сосудов это означает, что нет пути внутри многоугольника от данной точки до дырки, не поднимающегося выше положения рассматриваемой точки. Как же проверить существование такого пути? Сделаем это следующим образом. Очертим мысленно максимальную связную фигуру, лежащую внутри многоугольника и содержащую данную точку, никакая точка которой не лежит выше данной. Тогда такой путь существует тогда и только тогда, когда эта фигура не содержит дырок, кроме, возможно, дырок на верхней грани. Поскольку дырки расположены только в вершинах нашего многоугольника, а мы проверяем исключительно точки, которые лежат по вертикали между двумя несовпадающими ординатами каких-либо вершин многоугольника, то ситуация, когда дырка оказалась на верхней грани нашей фигуры, исключена, так что можно просто проверять, что фигура не содержит дырки.

Обойдем эту фигуру в нашей программе по контуру. Для этого реализуем процедуру «шага вправо» – для данной точки найдем точку, в которую мы попадем, если будем смещаться направо, пока не окажемся на границе многоугольника. Для этого последовательно пересечем горизонтальную прямую со всеми сторонами многоугольника, лежащими справа от точки, и выберем среди точек пересечения ближайшую к нашей. Следует отметить, что надо быть аккуратными, ибо есть случай, когда наша точка уже лежит на стороне многоугольника и вправо из нее пойти нельзя. Будем считать, что процедура *ihor* была преобразована в функцию, которая возвращает, есть ли пересечение стороны с горизонтальной линией (см. фрагмент 2).

Проверка что $y_i < y_{i+1}$ необходима именно для решения указанной проблемы с точкой, лежащей на стороне многоугольника. Нам не требуется пересекать нашу горизонтальную прямую с участками, которые идут вниз – при обходе против часовой стрелки такой участок не может быть

Фрагмент 1.

```
{ Входные данные находятся в массивах x и y }
{ Создаем копию массива y и выполняем его сортировку }
y2 := y;
for i := 1 to n do
  for j := i + 1 to n do
    if y2[i] < y2[j] then
    begin
       t := y2[i]; y2[i] := y2[j]; y2[j] := t;
    end;

{ Находим верхнюю левую точку многоугольника }
top := 1;
for i := 2 to n do
  if (y[i] > y[top]) or
    ((y[top] = y[i]) and (x[top] > x[i])) then top := i;

i := top; { Текущая вершина }
j := 1;   { Текущая горизонтальная черта }
s := 0;   { Текущая площадь }
repeat
  { Очередная сторона разбита на части горизонтальными прямыми, }
  { аккуратно перемещаемся по ней. }
  { Инвариант в этом месте - y[i] = y2[j] }
  xprev := x[i];
  yprev := y[i];
  repeat
    { Находим следующую горизонтальную прямую }
    while (y2[j] = yprev) do
        if y[i + 1] < y[i] then inc(j) else dec(j);
    { Пересекаем с ней нашу сторону }
    ihor(y2[j], i, xp);

    { Если середина участка содержит нефть, то модифицируем площадь }
    if wet((xprev + xp) / 2, (yprev + y2[j]) / 2) then
        s := s + (xprev + xp) * (y2[j] - yprev);

    { Сдвигаем участок }
    xprev := xp;
    yprev := y2[j];
    { Если дошли до конца стороны, то переходим к следующей }
  until y2[j] = y[i + 1];

  i := i mod n + 1;
  while y[i] = y[i + 1] do
    i := i mod n + 1;
  { Если обошли многоугольник до конца, то выход }
until i = top;
```

ближайшим к нам справа, а если наша точка лежит на таком участке, то он находится слева.

Теперь реализовать обход нашей фигуры не представляет особого труда. Идем из нашей точки вправо до упора с помощью *stepright*. Дальше идем по границе. Когда пытаемся подняться выше уровня нашей точки, «перепрыгиваем» вправо. Если встречаем дырку, то это означает, что наша точка не содержит нефти. Если возвращаемся на исходное место и не встретили дырку, то наша точка содержит нефть (см. фрагмент 3).

Фрагмент 2.

```
procedure stepright(xp, yp: real; var iright: longint);
var
  xright, xb: real;
  i: longint;
begin
  xright := 0;
  iright := 0;
  for i := 1 to n do
    if ihor(yp, i, xb) then
    begin
      if (xb >= xp - eps) and (y[i] < y[i + 1]) then
          if (iright = 0) or (xb < xright) then
          begin
              xright := xb;
              iright := i;
          end;
    end;
end;
```

Фрагмент 3.

```
function wet(xp, yp: real): boolean;
var
  i: longint;
  xb: real;
  iright: longint;
begin
  wet := false;

  stepright(xp, yp, iright);
  i := iright; { Текущая сторона, по которой мы идем }
  while true do
  begin
    if h[i] then { Нашли дырку }
          exit;
    if (y[i] < y[i - 1]) and (y[i - 1] >= yp) then
      begin
        { Пытаемся подняться выше нашей точки }
        ihor(yp, i - 1, xb);
        stepright(xb, yp, i);
        if i = iright then { Полный круг }
            break;
     end else begin
       { Перемещаемся вдоль границы }
        dec(i);
        if i < 1 then
           i := n;
      end;
  end;

  wet := true;
end;
```

Фрагмент 4.

```
function ihor(yp: real; i: longint; var xp: real): boolean;
begin
    ihor := false;
    if y[i] = y[i + 1] then
        exit;
    if (y[i] - yp) * (y[i + 1] - yp) > eps then
        exit;
    ihor := true;
    xp := x[i] + (x[i+1] - x[i]) * (yp - y[i]) / (y[i+1] - y[i]);
end;
```

Отметим, что для того, чтобы не происходило выхода за границы массива, следует организовать массивы от 0 и скопировать в нулевую вершину – n-ую, а в $(n + 1)$-ую – первую.

Итак, построение нашего решения завершено. Приведем для полноты картины также текст функции *ihor* (см. фрагмент 4).

ЗАДАЧА G. РАЗБИЕНИЯ

Разбиение прямоугольника – это представление его в виде объединения множества меньших непересекающихся прямоугольников. На рисунке 7 приведено несколько вариантов разбиения.

На рисунке 8 изображены три одинаковых прямоугольника, разбитые на меньшие прямоугольники. Разбиение B получено из разбиения A разделением на части двух прямоугольников из A. Вообще, если разбиение B получено из разбиения A разделением одного или нескольких прямоугольников из A, то говорят, что B *мельче,* чем A, и A, в свою очередь, *грубее,* чем B. Это отношение порядка является лишь частичным – разбиение C не мельче и не грубее ни чем A, ни чем B.

Если даны два разбиения D и E одного и того же прямоугольника, то существует бесконечно много его разбиений, которые одновременно мельче и D и E. Так, на рисунке 9 F и G мельче, чем и D и E. Однако известно, что среди разбиений, которые одновременно мельче и D и E, существует одно *наиболее грубое*. Оно называется *точной нижней гранью* D и E. На рисунке 9 разбиение F – точная нижняя грань D и E.

На рисунке 10 разбиения H и J грубее, чем и D и E. Здесь J – *наиболее мелкое* разбиение, которое грубее и D и E. Оно, таким образом, является *точной верхней гранью* D и E.

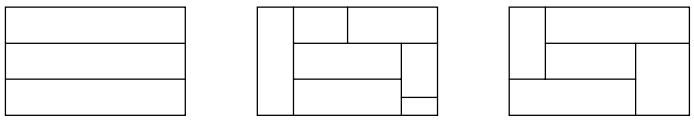


Рисунок 7.

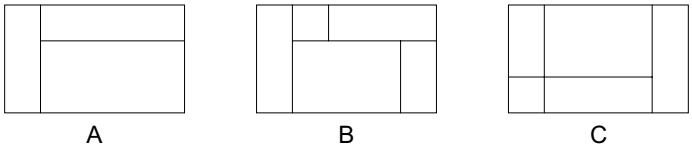


Рисунок 8.

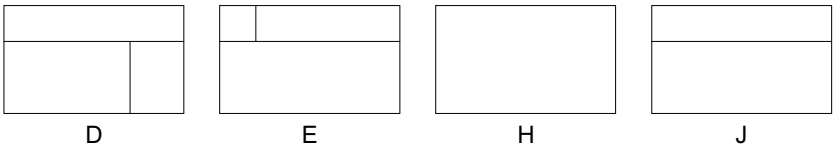


Рисунок 9.

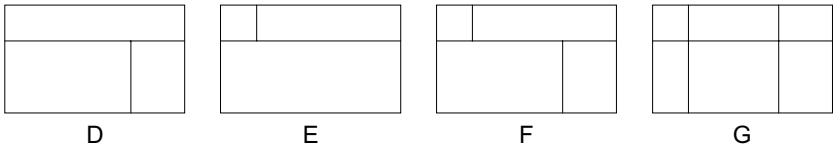


Рисунок 10.

Напишите программу, которая по двум заданным разбиениям одного и того же прямоугольника найдет их точную нижнюю и верхнюю грани.

Входные данные

Входной файл содержит описания нескольких тестовых примеров. Первая строка каждого тестового примера содержит числа w и h – ширину и высоту прямоугольника ($0 < w, h \le 20$). На следующих $h + 1$ строке содержатся два разбиения, каждая из этих строк содержит $4w + 3$ символа. Первые $2w + 1$ символ относятся к первому разбиению, а следующие $2w + 1$ – ко второму. Два разбиения разделены пробелом в каждой строке. Горизонтальные линии нарисованы с помощью знаков подчеркивания «_», а вертикальные – с помощью вертикальной черты «|».

Последняя строка входного файла содержит два нуля.

Выходные данные

Для каждого тестового примера выведите сначала строку, содержащую номер тестового примера. Затем выведите точную нижнюю и точную верхнюю грани, следуя тому же формату, что и во вводе.

После каждого тестового примера выводите пустую строку.

Пример

Пример входного файла	Вывод для данного входного файла
4 3 _ _ _ _ _ _ _ _ |_ _ _ _| |_|_ _ _| | | | | | |_ _|_ _| |_ _ _ _| 3 4 _ _ _ _ _ _ | | | | | | | | | |_|_ _| |_|_ _| | | | |_ _|_| |_ _|_| 0 0	Case 1: _ _ _ _ _ _ _ _ |_|_ _ _| |_ _ _ _| | | | | | |_ _|_ _| |_ _ _ _| Case 2: _ _ _ _ _ _ | | | | | | |_|_ _| | | |_|_|_| | | |_ _|_| |_ _ _|

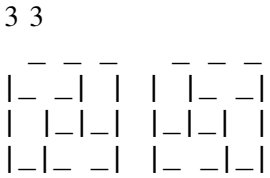
Решение.

Эта несложная задача вызвала на удивление много проблем у участников соревнования. Интересно, что за приведенными в условии задачи определениями стоит довольно обширная теория. Множество, на котором введен частичный порядок, в котором любое двухэлементное множество имеет точную верхнюю и нижнюю грани, называется *решеткой*. Заинтересовавшимся этой темой можно порекомендовать прочесть какую-либо книгу по общей алгебре. Мы же подойдем к поставленной задаче с более практической точки зрения.

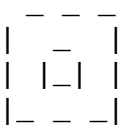
Итак, начнем с более простого – нахождения точной нижней грани. Для этого просто объединим все разрезы. Действительно, поскольку пересечение двух пря-

моугольников снова оказывается прямоугольником, то полученное разделение прямоугольника на части будет разбиением.

Более сложным является нахождение точной верхней грани. Первой идеей, которая приходит в голову, является выделение прямоугольников, которые есть в обоих разбиениях. Однако гипотеза, что множество этих прямоугольников составит разбиение нашего прямоугольника, оказывается неверна. Чтобы продемонстрировать это, посмотрим на следующий пример:



Несложно видеть, что квадратик в центре есть в обоих разбиениях и является единственным прямоугольником, обладающим таким свойством. Однако следующее деление прямоугольника на части,



очевидно, не является разбиением.

Удивительно, но более простая идея оказывается более плодотворной. Рассмотрим некоторое разбиение прямоугольника. Посмотрим внимательно на все внутренние точки прямоугольника. Несложно видеть, что из 16 возможных способов выхода линий разбиения из такой точки, допустимыми являются лишь 8 (см. рисунок 11).

Оказывается, что верно и обратное утверждение. Таким образом, разделение прямоугольника линиями на части является разбиением его на прямоугольники тогда и только тогда, когда через все внутренние точки линии разбиения проходят одним из восьми допустимых способов. Доказательство этого утверждения оставим в виде несложного упражнения. Отметим, что в нашей задаче достаточно смотреть на «точки с целыми координатами», поскольку только через них проходят разрезы.

Кроме того, ситуация сложилась для нас очень удачно – любое подмножество сегментов, исходящих из точки в недопустимом случае, также является недопустимым. Поэтому если из некоторой точки исходит недопустимая комбинация, то можно просто удалить все сегменты, из нее исходящие. Отметим, кстати, что именно выполнение этого свойства гарантирует единственность точной верхней грани и, как следствие, то, что множество разбиений является решеткой.

Теперь реализовать построение точной верхней грани не представляет никакого труда. Составим деление прямоугольника на части, взяв из разбиений те сегменты, которые входят в оба разбиения. Затем будем последовательно просматривать все внутренние точки нашего предразбиения и удалять те сегменты, которые выходят из точек, имеющих недопустимую комбинацию выходящих сегментов. Будем повторять эту операцию, пока остаются такие точки. Ясно, что, поскольку каждый раз удаляется хотя бы один сегмент, процесс рано или поздно завершится. Более того, поскольку количество внутренних точек имеет порядок $O(wh)$ и количество сегментов также порядка $O(wh)$, то время работы нашего алгоритма есть $O(w^2h^2)$.

ЗАДАЧА H. ГЛУПАЯ СОРТИРОВКА

Ваш младший брат недавно получил домашнее задание, и ему нужна ваша помощь. Учитель дал ему последовательность

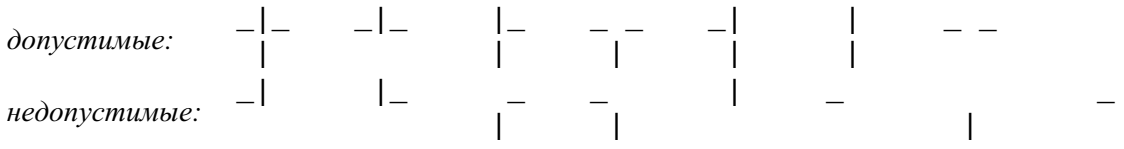


Рисунок 11.

чисел, которую требуется отсортировать в возрастающем порядке. Во время сортировки можно менять местами два любых числа. Каждый обмен имеет стоимость, равную сумме чисел, которые в него входят.

Напишите программу, которая найдет минимальную стоимость такой сортировки заданной последовательности.

Входные данные

Входной файл содержит описания нескольких тестовых примеров. Каждый тестовый пример состоит из двух строк. Первая строка содержит положительное целое число n ($n > 1$) – количество чисел, которые требуется отсортировать. Вторая строка содержит n различных чисел (каждое число положительное и не больше 1000), которые надо отсортировать.

Последняя строка входного файла содержит число 0.

Выходные данные

Для каждого тестового примера выведите одну строку, содержащую номер тестового примера и минимальную стоимость сортировки чисел в этом тестовом примере.

После каждого тестового примера выводите пустую строку.

Пример

Пример входного файла	Вывод для данного входного файла
3 3 2 1 4 8 1 2 4 5 1 8 9 7 6 6 8 4 5 3 2 7 0	Case 1: 4 Case 2: 17 Case 3: 41 Case 4: 34

Решение.

Рассмотрим ориентированный граф, вершинами которого будут элементы нашего массива, который требуется отсортировать. Из каждой вершины графа проведем одно ребро, которое будет вести в вершину, соответствующую месту, на которое должен встать рассматриваемый элемент в отсортированном массиве.

Поскольку в этом графе полустепени захода и исхода каждой вершины равны 1, то он представляет собой набор циклов. Заметим, что суммарный штраф за сортировку наших чисел равен $\sum_i c_i v_i$, где v_i – i-ое число в нашем наборе, а c_i – количество перемещений этого числа. Один цикл можно отсортировать так, чтобы только одно число перемещалось более одного раза. Для этого следует поставить на свое место сначала число, которое стоит на месте наименьшего числа в цикле. Цикл сократится на 1. Повторяя так далее, получим стоимость такой сортировки цикла $\sum_{i \in C} v_i + m(|C|-2)$, где за C обозначено множество вершин нашего цикла, а за m – минимальный элемент в нашем цикле. Заметим также, что мы можем объединить два цикла в один, поменяв местами их минимальные элементы. Тем самым, для любого цикла, кроме цикла, содержащего минимальный элемент массива, можно выполнить сортировку за $\sum_{i \in C} v_i + m + g(|C|-1)$, где за g обозначен минимальный элемент всего массива (после того, как все элементы рассматриваемого цикла встанут на свои места, цикл, с которым мы объединили рассматриваемый, примет свой первоначальный вид). Выбрав меньшую из этих двух величин, получим оптимальную стратегию для данного цикла.

Доказательство корректности приведенного алгоритма оставим в качестве интересного упражнения. Приведем лишь некоторые рекомендации. Рассмотрите возможные последовательности обменов и

докажите, что в каждом обмене должен участвовать минимальный элемент какого-либо цикла. Затем докажите, что в этом классе стратегий приведенная стратегия является оптимальной.

ЗАДАЧА I. КАЧЕНИЕ КОЛЕСА

Один из способов измерения длины некоторого пути – это прокатить по нему колесо (подобное колесу от велосипеда). Если мы знаем радиус колеса и количество оборотов, которое оно совершит, то мы сможем подсчитать длину пути.

Этот метод работает неплохо когда путь представляет собой ровную поверхность. Однако когда на пути встречаются кочки и ямы, которые представляют собой резкие перепады высоты, то длина пути может быть подсчитана неточно, поскольку колесо может совершить поворот вокруг точки (вокруг какого-нибудь угла) или скатиться по вертикальной плоскости. В этой задаче вам требуется подсчитать расстояние, пройденное центром колеса, когда оно катится по поверхности, содержащей только горизонтальные и вертикальные участки.

Чтобы измерить длину, колесо изначально помещают центром точно над началом пути. Затем колесо катят вперед, постоянно сохраняя его контакт с поверхностью, пока оно не окажется центром точно над концом пути.

Предположим в качестве примера, что задан путь, изображенный слева на иллюстрации внизу, и колесо имеет радиус 2. Путь начинается и заканчивается горизонтальными участками длины 2, находящимися на одной высоте. Между ними находится участок длины 2.828427, лежащий на 2 ниже крайних участков. Чтобы измерить путь, колесо изначально помещают в положение 1. Затем оно катится до позиции 2, поворачивается на 45 градусов до позиции 3, снова поворачивается на 45 градусов до позиции 4 и, наконец, катится по горизонтали до своего конечного положения 5. Центр колеса, таким образом, прошел путь 7.1416, а не 6.8284.

На рисунке 12 справа путь начинается и заканчивается горизонтальными сегментами длины 3, между которыми находится горизонтальный сегмент длины 7, расположенный на глубине 7. При этом колесо радиуса 1 пройдет путь 26.142, прежде чем достигнет конца пути.

Входные данные

Входной файл содержит описания нескольких тестовых примеров. Каждый тестовый пример начинается с положительного вещественного числа – радиуса колеса и целого числа n – которое лежит в пределах от 1 до 50. Затем следуют n пар вещественных чисел. Первое число есть расстояние по горизонтали до следующей вертикальной поверхности. Второе число в каждой паре есть изменение высоты после этого участка, приведенное со знаком. Положительное число означает увеличение высоты, в то время как отрицательное – уменьшение. Вертикальные участки всегда перпендикулярны горизонтальным. В n-ой паре изменение высоты есть 0.

Входной файл завершается двумя нулями.

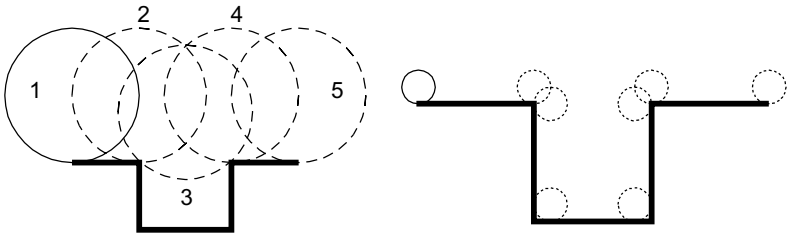


Рисунок 12.

Выходные данные

Для каждого тестового примера выведите одну строку, содержащую номер тестового примера и расстояние, пройденное центром колеса с точностью до 3 знака после запятой.

После каждого тестового примера выводите пустую строку.

Пример

Пример входного файла	Вывод для данного входного файла
2.0 3 2.0 -2.0 2.828427 2.0 2.0 0.0 1.0 3 3.0 -7.0 7.0 7.0 3.0 0.0 1.0 3 1.0 -4.0 2.0 4.0 1.0 0.0 0 0	Case 1: Distance = 7.142 Case 2: Distance = 26.142 Case 3: Distance = 5.142

Решение.

Решение этой задачи ненамного проще решения задачи про нефть (задачи F). Дадим на этот раз лишь набросок основной идеи. Путь нашего колеса состоит из отрезков прямой и дуг. Каждый раз, когда мы должны спуститься или подняться, для всех точек нашего пути проверяем, на какой угол мы должны повернуть колесо, чтобы опереться на эту точку в дополнение к точке поворота. Среди всех таких углов выбираем минимальный. Если таких углов нет, то поворачиваем на 90 градусов и «падаем», как во втором примере в условии.

Литература

1. *Кормен Т.*, *Лейзерсон Ч.*, *Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 2000.

2. *Асанов М.О.*, *Баранский В.А.*, *Расин В.В.* Дискретная математика: графы, матроиды, алгоритмы. – Ижевск: НИЦ «Регулярная и хаотическая динамика», 2001.

3. *Семенюк В.В.* Экономное кодирование дискретной информации. СПб: СПб - ГИТМО (ТУ), 2001.

4. *Романовский И.В.* Дискретный анализ. СПб.: Невский диалект, 2000.

Станкевич Андрей Сергеевич,
член оргкомитета
Всероссийской интернет-олимпиады
по информатике и
программированию.