Для **быстрого знакомства с материалами** смотрите раздел 5.

Пылесосы

Сегодня нам предстоит заняться уборкой клеточных ковров с помощью современных пылесосов. За окном уже XXI век, а значит, и пылесосы будут высокотехнологичными: для уборки области ковра им не надо объезжать каждую клетку ковра, а достаточно объехать её по контуру, тогда вся область, оказавшаяся внутри контура, станет чистой. Ковры тоже будут необычными: вместо скучных прямоугольных ковров пылесосы будут работать на нескучных тороидальных коврах.

Ковёр представляет собой прямоугольное клеточное поле. Поле зациклено как сверху-вниз, так и слева-направо. Это означает, что при попытке переместиться вверх с самого верхнего ряда клеток пылесос попадёт на самый нижний ряд и наоборот, при этом вторая координата пылесоса не изменится. Аналогично работают перемещения влево и вправо, соответственно, с самого левого столбца и с самого правого столбца поля.

Для каждой клетки поля запоминается игрок, чей пылесос последний раз очистил её. Если такой игрок есть, то будем считать, что этот игрок *владеет* данной клеткой. Изначально все клетки поля считаютися ничейными, за исключением тех, на которых начинают игру пылесосы, и соседних с ними по стороне или углу.

У каждого игрока имеется по одному пылесосу, с помощью которого ему предстоит очистить как можно бóльшую площадь. В дальнейшем понятия «пылесос» и «игрок» будут порой отождествляться. Пылесос умеет перемещаться по клеткам поля и занимает на нём ровно одну клетку. За каждый такт пылесос должен переместиться в одну из четырёх соседних по стороне клеток.

Как только пылесос выходит за пределы клеток, которыми он владеет, он начинает оставлять за собой *след*. При этом запоминается клетка, на которой он находился за такт до этого: будем называть её *стартовой*. Когда пылесос приходит в клетку, из которой по клеткам игрока достижима стартовая клетка, след считается замкнутым. Как только след замыкается, окружённая им территория очищается тем игроком, чей пылесос замкнул контур. Этот игрок начинает владеть всеми этими клетками.

Когда след замыкается, он дополняется до замкнутого контура некоторым кратчайшим путем по клеткам во владении игрока. Если полученный контур проходит всё поле насквозь (т.е. проходит через все строки поля или через все столбцы поля), то такой контур игнорируется: весь след стирается, и никакая территорию во владение к игроку не переходит. В противном случае все клетки внутри контура и все клетки самого контура переходят во владение игроку.

Если на след игрока наезжает чей-либо пылесос, то игрок, на чей след наехали, погибает. Это происходит даже в том случае, если игрок наезжает на свой же след. После смерти игрока пылесос и его след стираются с поля, а все клетки в его владении становятся ничейными.

В архиве с игрой имеется визуализатор, позволяющий в процессе игры управлять пылесосом (в том числе, и несколькими пылесосами) с клавиатуры. Таким образом, визуализатор можно использовать для изучения механики игры. Подробнее о запуске визуализатора можно прочитать в разделе 8.

1 Моделирование игры

Игровое поле является таблицей, состоящей из 140×140 клеток. Каждая клетка имеет свои координаты. Координата Y обозначает номер строки, ось Y направлена сверху вниз. Координата X обозначает номер столбца, ось X направлена слева направо.

Игра состоит из 1 500 тактов. За каждый такт пылесос каждого игрока делает ход. Игроки ходят по очереди. Порядок игроков в каждой отдельной игре зафиксирован и **не** меняется по ходу игры. На сервере жюри порядок игроков определяется случайным образом в каждом игре (см. ключ `--shuffle`). На компьютере участника порядок игроков по умолчанию соответствует порядку аргументов командной строки.

Все генераторы (псевдо)случайных чисел инициализируются при помощи некоторого значения `<seed>` и ведут себя детерминированно. На сервере жюри `<seed>` задаётся автоматически, а локально его можно установить с помощью ключа `--seed`. Обратите внимание, что из-за наличия мягкого ограничения по времени даже запуски игры с одинаковым `<seed>` могут приводить к разным результатам.

2 Среда исполнения

Пылесосы программируются на языке Lua (версии 5.1). Программа запускается один раз и должна определить функцию `tick`, которая затем вызывается на каждом такте. Значения глобальных переменных сохраняются между тактами. Таким образом, наименьшее допустимое решение выглядит так:

```
function tick()
end
```

Вам доступны базовые функции стандартной библиотеки Lua, а также библиотеки `table`, `string`, `math`. В качестве исключения, вам недоступны функции `dofile`, `loadfile`, `pcall`, `xpcall`, а также `math.random`. При локальном тестировании без строгого режима эти исключения не действуют, и вам также доступны библиотеки `package`, `debug`, `io`, `os`, однако, использовать их не рекомендуется, поскольку на сервере жюри их нет.

При вызове функции `tick` можно использовать игровой API, записанный в глобальной таблице `game`. При изначальном запуске программы игровой API использовать нельзя.

Учтите, что на языке LUA нумерация начинается с единицы (как в Паскале). По умолчанию в игровом API все координаты клеток задаются относительно позиции пылесоса игрока в самом начале игры, но это можно отключить при помощи функции `game.coordmode`.

В игровом API есть следующие функции:

- `game.move(dy, dx)` — задаёт направление движения пылесоса как вектор. Параметры `dy` и `dx` определяют Y и X компоненты вектора соответственно. Одно из этих чисел должно быть нулевым, а второе должно равняться ± 1.
- `game.move(dir)` — задаёт направление движения пылесоса но номеру направления. Направление `dir` должно лежать в диапазоне от 0 до 3 включительно.
- `playerIdx, playerCnt, height, width, turns = game.params()`
  Возвращает постоянные, которые не изменяются в течение игры. Здесь `playerIdx` — номер вашего игрока (начиная с единицы), `playerCnt` — общее количество игроков,

`height` и `width` — размеры игрового поля по осям Y и X соответственно, `turns` — длительность игры в тактах.

- `turnIdx = game.status()` — возвращает номер текущего такта игры (начиная с единицы).

- `areaOf, tailOf, player = game.field(y, x)`
  Возвращает содержимое клетки поля с координатами `y`, `x`. Здесь `player` — номер пылесоса, который находится в клетке, `tailOf` — номер игрока, след которого находится в клетке, `areaOf` — номер игрока, который владеет клеткой. Каждое из этих чисел равно нулю, если такого игрока нет, то есть если в клетке нет пылесоса, нет следа, и ей никто не владеет соответственно.

- `alive, y, x, tailLen, area, killedCnt = game.player(idx)`
  Возвращает текущее состояние игрока (пылесоса) под номером `idx`. Если игрок мёртв, то все возвращённые числа равны нулю. Если игрок жив, то: `alive` равно 1, `y` и `x` — координаты Y и X клетки, в которой находится пылесос, `tailLen` — количество клеток в следе пылесоса плюс один, `area` — количество клеток во владении игрока, `killedCnt` — сколько пылесосов убил игрок (врезаясь в их хвосты).

- `y, x = game.tail(pIdx, tIdx)` — возвращает координаты клетки в следе игрока. Здесь `pIdx` — номер игрока, а `tIdx` — номер клетки в его следе (не должен превышать `tailLen` игрока). Функция возвращает `y` и `x` — координаты искомой клетки. Все клетки следа игрока пронумерованы в порядке их возникновения, начиная с двух. В качестве первой клетки эта функция возвращает стартовую клетку — ту клетку, в которой находился пылесос за такт до выхода за пределы своей территории. Обратите внимание, что первая клетка **не** является следом с точки зрения правил игры и **не** показывается функцией `game.field` как след.

- `game.coordmode(mode)` — переключает режим задания координат для клеток поля. Если `mode` равно 1 (по умолчанию), то в качестве координат клетки используется смещение относительно исходной позиции вашего пылесоса в самом начале игры. Например, координаты $y = -2$, $x = 0$ задают клетку, которая на две клетки выше стартовой. Если `mode` равно 0, то используются абсолютные координаты клетки на поле.

  Не стоит забывать, что поле циклически замкнуто по обеим координатам. Когда вы передаёте координаты клетки в API, они могут быть в любом диапазоне: игра трактует их по модулю размера игрового поля. Однако когда игра возвращает вам координаты, они всегда нормированы: в случае `mode` $= 0$ верно $1 \leqslant y \leqslant h$, $1 \leqslant x \leqslant w$, а в случае `mode` $= 1$ верно $-\lfloor \frac{h-1}{2} \rfloor \leqslant y \leqslant \lceil \frac{h-1}{2} \rceil$, $-\lfloor \frac{w-1}{2} \rfloor \leqslant x \leqslant \lceil \frac{w-1}{2} \rceil$, где h и w — размер поля по Y и X соответственно. Из-за цикличности поля могут возникать разные случаи, например соседние клетки следа пылесоса могут иметь сильно отличающееся значение по одной координате.

  Данная функция игрового API переключает режим нумерации на заданный параметром `mode`. После этого все функции игрового API начинают согласно новому режиму. **Рекомендуется выбрать режим нумерации клеток в первый такт игры и не переключать его в дальнейшем.**

Кроме того, вашему решению всегда доступны следующие функции:

- `log(...)` — печатает сообщение в текстовый лог игры. Все переданные параметры конвертируются в строки и печатаются подряд в виде одного сообщения. Текстовый лог прошедших на сервере игр можно скачивать и смотреть. В строгом режиме (в том числе на сервере жюри) есть ограничения на размер сообщений. Сообщения вашего игрока можно отфильтровать например по строке `p17:`, если ваш игрок имеет 17-ый номер в игре.

- `debugger()` — программный breakpoint для отладчика ZeroBrane. При вызове функции в режиме отладки (`@d` или `@D` — см. описание матариалов) программа приостанавливается, как будто в этом месте сработал breakpoint. Далее в среде разработки ZeroBrane можно смотреть значения переменных, отлаживать по шагам, продолжить выполнение, и т.д. При вызове функции без включения режима отладки ничего не происходит.

- `r = randomint(a, b)` — генерирует случайное целое число `r` в диапазоне от `a` до `b` включительно.

- `r = randomreal()` — генерирует случайное вещественное число `r` в диапазоне между 0 до 1 включительно.

3 Ограничения

Ограничение по памяти составляет 10 мегабайт. При превышении этого ограничения ваша программа будет отключена, а пылесос продолжит двигаться в последнем заданном направлении. После каждого вызова функции `tick` автоматически запускается полная сборка мусора. При желании вы можете управлять сборкой мусора с помощью стандартной функции `collectgarbage` языка Lua.

Каждый вызов функции `tick` и инициализация программы должны завершиться за 500 миллисекунд. Если программа не завершается в течение этого времени, то она будет принудительно отключена, а пылесос продолжит двигаться в последнем заданном направлении.

Кроме того, есть мягкое ограничение на суммарное время работы вашей программы за все прошедшие такты: это 500 миллисекунд, плюс количество прошедших тактов, умноженное на 5 миллисикунд. При превышении мягкого ограничения ваша программа будет пропускать дальнейшие такты, пока квота не вырастет достаточно, чтобы покрыть использованное время.

Общий размер всех сообщений, выданных программой в текстовый лог с помощью функции `log`, ограничен 1 мегабайтом на всю игру. При превышении этого размера дальнейшие вызовы `log` ничего не делают. Кроме того, каждое сообщение автоматически обрезается до 200 символов, а специальные символы заменяются на `?`.

Все описанные ограничения действуют только в строгом режиме. Строгий режим включается ключом `--strict` или `-s`. Строгий режим требуется явного включения, а на сервере жюри он включен всегда.

4 Отправка решений и система оценки

По результатам игры очки каждого решения определяются по формуле:

$$S_i = A_i \cdot \sqrt{1 + \frac{K_i}{2}}$$

Здесь S_i — очки решения за игру, K_i — сколько других игроков убило решение. Число A_i определяется как количество клеток во владении игрока — максимальное отмеченное в течение всей игры.

Участники могут отправлять свои решения в систему тестирования nsuts. На сервере жюри в бесконечном цикле запускаются игры, у каждой следующей игры порядковый номер на единицу больше предыдущей. В каждой игре на сервере жюри от каждого участника участвует последнее отправленное к этому моменту решение. Истории этих игр вы можете скачать и посмотреть (подробнее в разделе 9).

Итоговые очки участника в первой номинации вычисляются как взвешенное среднее его очков за все игры на сервере жюри. Если решение участника не участвует в игре, то считается, что он получает за игру ноль очков. Обратите внимание, что **прошедшие в течение тура игры также учитываются при подведении итогов**. Финальное тестирование проходит после конца тура. В нём участвует последнее отправленное решение от каждого участника.

Веса игр определяются таким образом, что:

1. В течение первого часа тура результаты игр **не** учитываются вообще.
2. Результаты игр в остальные четыре часа тура составляют 50% итоговых очков участника.
3. Результаты игр финального тестирования составляют 50% итоговых очков участника.
4. Вес каждой игры пропорционален её длительности в секундах. То есть каждый учитываемый час тура имеет примерно одинаковый вес.

В случае возникновение серьёзных проблем во время тура жюри имеет право изменить вес игр, произошедших в течение тура, в меньшую сторону.

Предполагается провести четыре часа финального тестирования. Тогда итоговые очки конкретного участника будут вычисляться по формуле:

$$R = \sum_{j=p}^{q} S_j \cdot T_j$$

Здесь S_j — очки участника за j-ую игру, T_j — длительность j-ой игры в секундах, а R — итоговые очки участника. При этом игры с номерами от p до q — это все игры финального тестирования и игры последних четырёх часов тура.

В течение тура вам доступен текущий рейтинг в системе nsuts. Данный рейтинг считается по приведённой выше формуле, однако, **в нём учитываются только последние 10 игр**.

5 Материалы

Архив с материалами можно скачать по ссылке на вкладке «Новости» в системе nsuts. Чтобы материалы работали, нужно положить их в одну директорию, и запускать все скрипты и исполняемые файлы из неё. Если вы хотите скопировать что-то в другое место, не забудьте перенести необходимые служебные файлы. Для работы рекомендуется использовать Far Manager или командную строку Windows.

Для быстрого знакомства с материалами можно запустить команды:

1. `conrun.exe sample.lua^49` — запустить игру с 49 копиями предоставленного примера игрока.

2. `visrun.exe -r --demo` — посмотреть историю игры в визуализаторе (быстрый демо-режим).
3. `visrun.exe sample.lua^9` — запустить игру с 9 копиями игрока-примера и с визуализатором.

Полное содержимое архива:

- `conrun.exe` — консольная запускалка игр (подробности в разделе 6)
- `visrun.exe` — визуализатор: может запускать новые игры и проигрывать истории старых игр (см. разделе 8)
- `sample.lua` — пример решения участника
- `download.bat` — скрипт для скачивания результатов игр с сервера жюри (раздел 9)
- `statement.pdf` — это условие задачи
- `docs\*` — документация по языку Lua (особенно рекомендуется `LuaIn15Minutes.txt`)
- `libs\*` — немного дополнительных библиотек на Lua (`printf`, сериализация, ...)
- `grep.exe` — утилита для поиска информации в тексте (см. раздел 7)
- `wget.exe` — служебный файл для скрипта `download.bat`
- `lua51.dll` — служебный файл для запускалок игры
- `*.dll` (остальные), `platforms` — служебные файлы для `visrun.exe`

6 Запуск решений

Локально запустить новую игру с заданными решениями можно при помощи `conrun.exe` или `visrun.exe`. Параметры запуска и поведение игры одинаковы для обеих программ.

Программа `conrun.exe` работает полностью в текстовом режиме и моделирует игру с максимально возможной скоростью.

Программа `visrun.exe` открывает окно визуализатора и всегда показывает в нём текущее состояние игры. Моделирование игры при этом идёт согласно командам пользователя, а скорость моделирования ограничена. Подробно визуализатор описан в разделе 8.

Для запуска игры нужно указать в параметрах командной строки все решения, которые будут запущены. Каждое решение задаётся именем файла с кодом, при этом файл должен лежать в рабочей директории. Также допускается задавать путь к файлу решения, если файл лежит в другой директории. Решения бывают двух типов:

1. Файл с расширением `.lua` — решение с исходным кодом на языке Lua. Рекомендуется запускать все ваши решения именно таким образом.
2. Файл с расширением `.sol` — скомпилированное решение в бинарном формате. Этот способ сделан в первую очередь для **локального запуска решений других участников**, подробнее о скачивании решений других участников написано в разделе 9. Скомпилированные решения всегда запускаются в строгом режиме и с отключением вывода в текстовый лог игры.

Например:

`conrun sample.lua other.sol backup\old.lua` — запускает игру с тремя решениями. Решения `sample.lua` и `other.sol` берутся из рабочей директории, а `old.lua` — из поддиректории `backup`. При этом файл `other.sol` содержит бинарный код решения, а остальные два файла — исходный текст решений.

При указании решения для запуска поддерживается специальный синтаксис:

1. Можно задавать сразу много решений с помощью шаблона, содержащего звёздочку (glob). Такой шаблон заменяется на список всех файлов, имя которых соответствует шаблону при замене каждой звёздочки на произвольную строку.
2. Можно запустить много копий одного решения, добавив сразу после имени файла крышку (ставится по Shift + 6) и количество копий.
3. Чтобы подключить отладчик ZeroBrane к решению, нужно дописать к имени файла `@D` или `@d`. При дописывании `@D` отладчик сразу подключается и ставит игру на паузу. При дописывании `@d` отладчик подключается только после первого вызова функции `debugger` (подробнее в разделе 2).

Например:

`conrun *.lua` — запускает все файлы с расширением `.lua` в рабочей директории.

`conrun sample.lua^25 me.lua^20` — запускает 25 копий решения `sample.lua` и 20 копий решения `me.lua`.

`conrun me.lua@D` — запускает решение `me.lua` с отладчиком.

`conrun me.lua@D game0123\*.sol` — запускает решение `me.lua` с отладчиком, а также все файлы с расширением `.sol` из поддиректории `game0123`.

Кроме того, можно указать в командной строке дополнительные ключи. Все ключи начинаются с символа минуса, например `-s` или `--strict`. Если нужно указать для ключа значение, то оно указывается после знака равенства, например `-t=100` или `--turns=100`.

Список ключей:

- `--help` — показать краткое описание параметров и ключей запуска.
- `--strict` или `-s` — запустить игру в строгом режиме. В этом режиме соблюдаются все ограничения из раздела 3.
- `--height=N` или `-h=N` — изменить высоту игрового поля.
- `--width=N` или `-w=N` — изменить ширину игрового поля.
- `--turns=N` или `-t=N` — изменить длительность игры в тактах.
- `--seed` — указать ключ инициализации для генераторов псевдослучайных чисел (по умолчанию равно нулю). При использованию одного и того же значения этого ключа моделирование игры происходит одинаково.
- `--shuffle` — переупорядочить заданные решения случайным образом, разместить игроков на поле случайно.
- `--save-binary` или `-b` — сгенерировать для всех `.lua`-решений бинарные `.sol`-файлы.
- `--names` или `-n` — переименовать игроков согласно заданному файлу (в каждом строке: имя файла решения, пробел, отображаемое имя игрока).

При запуске игры на сервере жюри будут использоваться ключи:

```
--strict --seed=? --shuffle --save-binary --names=?
```

7 Выходные файлы игры

После запуска игры в рабочей директории создаются файлы различных типов. Вы можете скачать все эти файлы для игр, проведённых на сервере жюри (подробности в разделе 9).

Файл `loggame.txt` содержит текстовый лог игры, в котором можно в частности увидеть:

- Ошибки компиляции и времени исполнения, превышение ограничений по времени и памяти.
- Сообщения, выданные игроком с помощью функцию `log(...)`.
- Затраты времени и памяти решения на каждый такт, сообщения о пропуске тактов.
- События замыкания следа и смерти игрока.

Для фильтрации текстового лога можнго использовать программу grep из материалов. Например:

`grep p15 loggame.txt` — напечатать все сообщения о 15-ом игроке.

`grep p15: loggame.txt` — напечатать всё, что вывел 15-ого игрок через `log(...)`.

`grep WARN loggame.txt` — напечатать все серьёзные сообщения, в частности всевозможные ошибки программ.

Следует заметить, что при локальном запуске игры все серьёзные сообщения (уровня `WARN` и `FATAL`) дублируются в консоль.

Файл `replay.bin` содержит историю игры, которую можно просмотреть в визуализаторе. Для просмотра достаточно запустить `visrun.exe` без параметров, если история содержится в `replay.bin` в рабочей директории. Подробнее визуализатор описан в разделе 8.

Файл `score.txt` содержит результаты игры. В каждой строке описан один игрок: записано имя файла с решением, очки игрока за игру, и отображаемое имя.

Файлы с расширением `.sol` содержат бинарные версии решений. Они создаются только при указании ключа `--save-binary`, в том числе на сервере жюри. Вы можете скачивать решения других участников в этом формате и запускать игру с ними локально, как с обычными решениями. Подробнее в разделах 9 и 6.

8 Визуализатор

С помощью визуализатора можно как запустить новую игру, так и воспроизвести запись (файл `replay.bin`) уже сыгранной игры.

Визулизатор позволяет:

- Проиграть одну итерацию игры (кнопка “Next Iteration”).
- Отмотать на одну итерацию назад (кнопка “Previous Iteration”).
- Перемотать состояние игры до определённой итерации (ввести номер итерации в поле “Iteration Number” и нажать Enter).
- Запустить моделирование игры (кнопка “Start Game”). При этом в поле “Delay” можно указать паузу между итерациями игры в миллисекундах.

Игровое поле поддерживает изменение масштаба (с помощью вращения колеса мыши) и перемещение (с помощью зажатой левой кнопки мыши). Для наблюдения за конкретным

игроком можно увеличить масштаб и в выпадающем поле “Player to Watch” выбрать имя наблюдаемого игрока.

Для воспроизведения записанной игры визуализатор требуется запускать с ключом `--replay`, в котором указать имя replay-файла (по умолчанию `replay.bin`):

```
visrun --replay
visrun --replay=game0123\replay.bin
```

О том, как воспроизвести игру из системы тестирования, написано в разделе 9. При запуске воспроизведения с ключом `--demo` визуализатор запускается в автоматическом режиме, значением ключа можно указать значение Delay.

При запуске новой игры визуализатор поддерживает все те же ключи, что и консольная версия игры `conrun` (описано в разделе 6). Например, `visrun -h=50 -w=100 sample.lua game0123\*.sol` запустит в визуализаторе новую игру на поле 50×100 с игроком `sample.lua` и со всеми скомпилированными игроками из директории `game0123`.

Для изучения механики игры визуализатор поддерживает игроков, управляемых с клавиатуры. Например, команда:

```
visrun -h=20 -w=20 keyboard^2
```

создаст игру на поле 20×20 с двумя игроками, управляемыми пользователем. Управлением осуществляется с помощью стрелок на клавиатуре.

9 Просмотр игр из системы тестирования

Скачать истории игр из системы тестирования можно в браузере из директории:

`http://parallels.nsu.ru/vacuumcleaner/active/`

Для каждой сыгранной игры создаётся директория с названием `gameXXXX`, где `XXXX` — порядковый номер игры. В каждой такой директории лежат все выходные файлы, описанные в разделе 7.

Для удобства скачивания истории, логов и решений можно использовать скрипт `download.bat`. Данный скрипт создаёт поддиректорию с указанным номером игры и скачивает в неё все выходные файлы этой игры. Если указать `last` вместо номера, то будет скачана последняя на текущий момент сыгранная игра.

Примеры:

`download.bat last play` — скачать последнюю игру и запустить её в визуализаторе.

`download.bat last` — просто скачать последнюю игру, визуализатор не запускать.

`download.bat 128` — скачать 128-ую игру (можно добавить `play` для проигрывания).

10 Отладка и ZeroBrane

Вы можете отлаживать решения на Lua, используя среду разработки ZeroBrane. Для запуска отладки (в примеру решения `my.lua`) нужно:

1. Запустить ZeroBrane Studio.
2. Запустить сервер отладки в меню: Project $\longrightarrow$ Start Debugging Server

3. Запустить новую игру, дописав `@D` в конец имени решения, например: `conrun *.sol my.lua@D`

Тогда в начале игры в ZeroBrane сработает точка останова, и можно будет отлаживать по шагам, смотреть значения переменных и т.д. Кроме того, вы можете ставить точки останова (breakpoint) в коде программы и продолжать исполнение.

Можно программно приостановить выполнение решения на Lua (т.е. сделать программный breakpoint), вызвав функцию `debugger()`. Программные точки останова работают, если решение было запущено с любым из суффиксов `@d` или `@D`. К сожалению, в случае запуска решения с суффиксом `@d` отладчик подключается к программе только после первого вызова функции `debugger()`.

В ZeroBrane рекомендуется использовать функцию анализа кода (Shift + F7) — это помогает быстро найти опечатки.