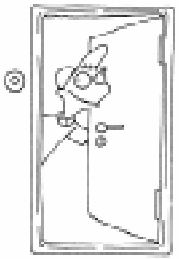
Ивановский Сергей Алексеевич,
Преображенский Алексей Семенович,
Симончик Сергей Константинович

АЛГОРИТМЫ ВЫЧИСЛИТЕЛЬНОЙ ГЕОМЕТРИИ. ВЫПУКЛЫЕ ОБОЛОЧКИ: ПРОСТЫЕ АЛГОРИТМЫ



1. ВВЕДЕНИЕ

Вычислительная геометрия – это очень интересная и сравнительно молодая область компьютерной науки, находящаяся на стыке информатики и геометрии и сочетающая геометрические понятия, идеи и модели с понятиями, идеями и методами разработки алгоритмов и структур данных.

Со времен Евклида (III век до нашей эры) *геометрические построения* были существенной частью и эффективным средством как геометрических исследований, так и геометрической педагогики. Классический пример представляют собой задачи на построение: построить треугольник по трем данным его сторонам, разделить данный угол пополам, из данной точки опустить перпендикуляр на данную прямую, разделить пополам данный отрезок [4].

Геометрические построения, по сути дела, являются зачатками алгоритмической науки. Действительно, в задачах на построение по заданным геометрическим объектам (точкам, прямым, отрезкам и окружностям), то есть *по исходным данным*, требуется построить другие геометрические объекты, то есть *выходные данные*, используя для этого фиксированный набор инструментов (базовых операций или примитивов), которыми могут быть, например, циркуль и линейка, или только циркуль [3], или только линейка. У геометрических построений существует также понятие *сложности*, введенное Эмилем Лемуаном в 1902 году и означающее минимальное количество применений базовых операций, необходимое для данного построения.

Появление компьютеров и затем бурное развитие компьютерной науки (информатики) открыло возможность симбиоза чисто геометрической природы задач на построение с аналитическим (вычислительным) подходом к их решению, что, в свою очередь, привело к возможности решения нового класса задач, которыми и занимается вычислительная геометрия.

В этих задачах *исходными данными* являются геометрические объекты (например, множество точек, набор отрезков, многоугольник), а *результатом* может являться:

– ответ на вопрос о некоторых отношениях между заданными объектами (например, пересекаются ли заданные отрезки);

– перечисление фактов относительно этих отношений (например, перечисление всех пар пересекающихся отрезков);

– построение нового геометрического объекта (например, наименьшего выпуклого многоугольника, содержащего заданные точки).

Причем важно, что как размер исходных данных (число заданных геометрических объектов), так и размер полученного ответа (перечисления или нового геометрического объекта) могут быть произвольными и, как правило, достаточно большими.

Вычислительная геометрия находит применение во многих областях науки и техники, таких как [5]:

– компьютерная графика и визуализация;

– компьютерное зрение и робототехника;

– географические информационные системы (ГИС);

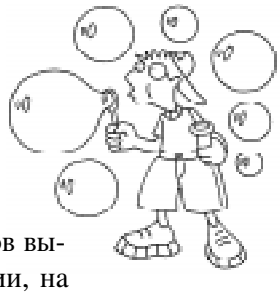
– системы автоматизированного проектирования (САПР);

– компьютерный анализ и интерпретация данных (например, компьютерная томография);

– молекулярная биология;

– астрофизика.

2.1. ВЫПУКЛАЯ ОБОЛОЧКА И ВЫПУКЛОЕ МНОЖЕСТВО



Задача построения выпуклой оболочки не только является центральной в целом ряде приложений, но и позволяет разрешить ряд вопросов вычислительной геометрии, на первый взгляд не связанных с ней. Построение выпуклой оболочки конечного множества точек, особенно в случае точек на плоскости, уже довольно широко и глубоко исследовано и имеет приложения, например в распознавании образов, обработке изображений, а также при раскрое и компоновке материала. Мы будем рассматривать эту задачу на плоскости.

Если нам задано множество точек Q, то его *выпуклая оболочка* $CH(Q)$ – это наименьшее выпуклое множество, включающее в себя все эти точки. *Выпуклым* называется множество точек, в котором выполняется следующее правило: если какие-то две точки принадлежат множеству, то и соединяющий их отрезок принадлежит этому же множеству (рис. 2.1).

Чтобы наглядно представить это понятие в случае, когда Q – конечное множество точек на плоскости, предположим, что это множество охвачено большой растянутой резиновой лентой. Когда лента освобождается, то она принимает форму границы выпуклой оболочки.

Еще одним понятием, которое нам понадобится, является понятие *крайней точки*. Точка p выпуклого множества R называется *крайней*, если не существует пары точек $a, b \in R$ таких, что p лежит на открытом отрезке (a, b). Несложно догадаться, что выпуклая оболочка конечного множества точек на плоскости является выпуклым многоугольником, все вершины которого принадлежат исходному множеству точек (и являются крайними точками выпуклой оболочки). В свою очередь, стороны этого многоугольника (называемые также *ребрами* многоугольника) образуют границу выпуклой оболочки (рис. 2.2).

Заметим, что для любого ребра выпуклой оболочки (многоугольника) часть точек исходного множества будет лежать на данном ребре (например, концы этого ребра), а оставшиеся точки по одну сторону от содержащей ребро прямой (попробуйте доказать это утверждение самостоятельно).

Верно и обратное утверждение если для отрезка, соединяющего две точки исходного множества, часть точек исходного множества лежит на нем, а оставшиеся точки по одну сторону от содержащей отрезок прямой, то данный отрезок является ребром выпуклой оболочки (рис. 2.3).

Выпуклые многоугольники принято задавать последовательностью вершин в порядке обхода по часовой стрелке или про-

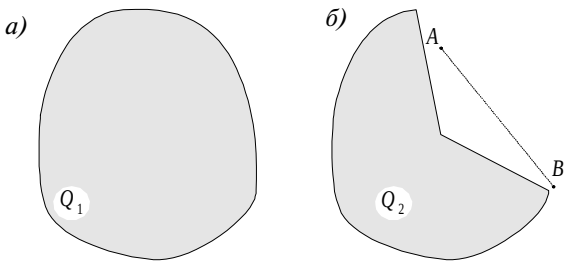


Рис. 2.1. Пример выпуклого (*а*) и невыпуклого (*б*) множества

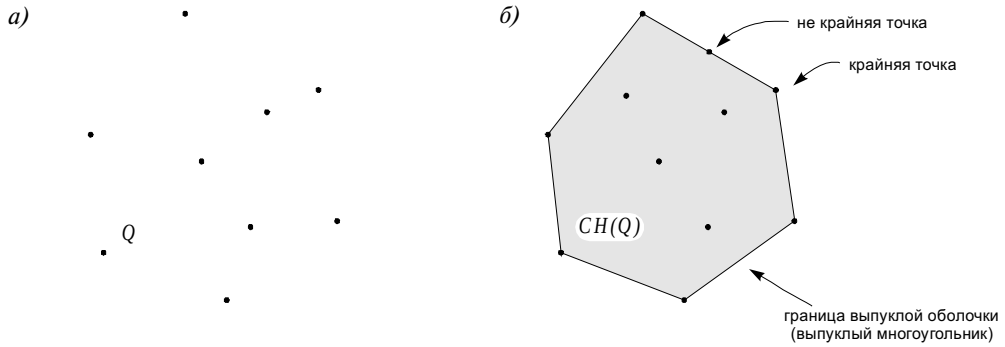


Рис. 2.2. Исходное множество точек (*а*) и его выпуклая оболочка (*б*)

тив часовой стрелки. Например, для выпуклого многоугольника, приведенного на рисунке 2.4, последовательность вершин *FEDCBA* является его обходом по часовой стрелке, а последовательность *ABCDEF* обходом против часовой стрелки.

Далее будет рассмотрено несколько алгоритмов построения выпуклой оболочки. Для всех этих алгоритмов искомой выпуклой оболочкой будем считать выпуклый многоугольник, заданный последовательностью своих вершин в порядке обхода против часовой стрелки.

2.2. АЛГОРИТМ ЗАВОРАЧИВАНИЯ ПОДАРКА

В некотором царстве, в некотором государстве король приказал своему садовнику оградить свой сад от посяганий учеников близлежащей школы. У садовника есть достаточно длинная веревка, и он намеревается использовать ее в качестве ограждения. Садовнику нужно построить изгородь так, чтобы как можно большую площадь огородить и как можно меньше веревки при этом потратить. Веревку можно привязывать только к стволам деревьев сада.

Несложно догадаться, что садовник должен построить не что иное как выпуклую оболочку множества точек. Точками в данном случае являются деревья (если смотреть на них с высоты птичьего полета).

Чтобы решить эту задачу, садовник будет действовать в соответствии со следующей стратегией: сначала он найдет такое дерево, которое точно будет включено в изгородь (оно будет являться крайней точкой выпуклой оболочки), и закрепит на нем веревку, а затем начнет обходить сад кругом (например, против часовой стрелки), держа в руках натянутую веревку. Когда он сделает полный круг, сад уже будет огорожен. Этот процесс проиллюстрирован на рисунке 2.5.

Оказывается, идея, использованная садовником для решения данной задачи, лежит в основе *алгоритма заворачивания подарка,* или *обхода Джарвиса* [2] (алгоритм называется так в честь своего автора Рэя Джарвиса, который придумал его в 1973 году).

На самом деле, для того чтобы описанный способ действия стал алгоритмом, его надо формализовать, то есть записать так, чтобы его мог выполнить робот, а не только человек.

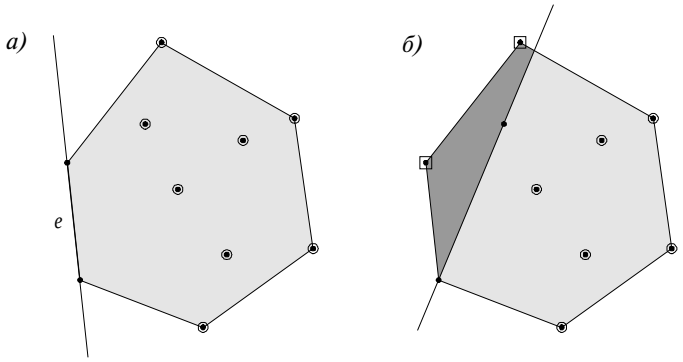


Рис. 2.3. Свойство ребер выпуклой оболочки

Пусть в памяти робота задана карта сада, где деревья обозначены точками на плоскости и каждая точка имеет пару координат. Обозначим общее количество деревьев за n, тогда координатами дерева с номером i будет пара чисел $p_i = (x_i, y_i)$, где x_i – абсцисса точки, соответствующей дереву, а y_i – ордината этой точки, при этом i может принимать значения от 1 до n.

Тогда наш (пока еще неформальный) алгоритм выглядит так (см. листинг 1).

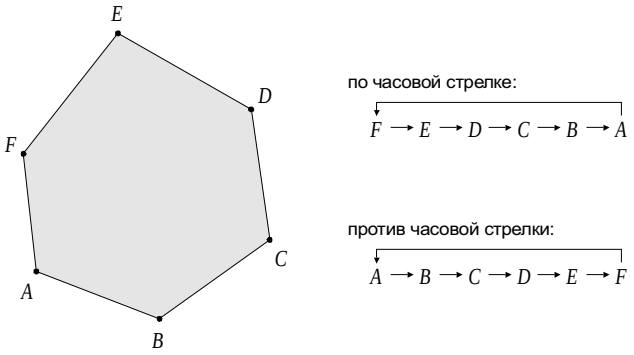


Рис. 2.4. Обходы выпуклого многоугольника

Листинг 1. algorithm JARVIS-MARCH(Q) $\rightarrow$ $CH(Q)$

Вход. Исходное множество точек $Q = \{p_i \mid p_i = (x_i, y_i),\ i \in [1..n]\}$.

Выход. Выпуклая оболочка множества точек $CH(Q)$, заданная последовательностью вершин (перечисленных в порядке обхода против часовой стрелки).

1 | Инициализировать выпуклую оболочку пустой последовательностью точек
2 | Выбрать первую точку, которая будет являться крайней точкой выпуклой оболочки
3 | Сделать полный круг вокруг множества точек, «заворачивая» вокруг него ребра выпуклой оболочки

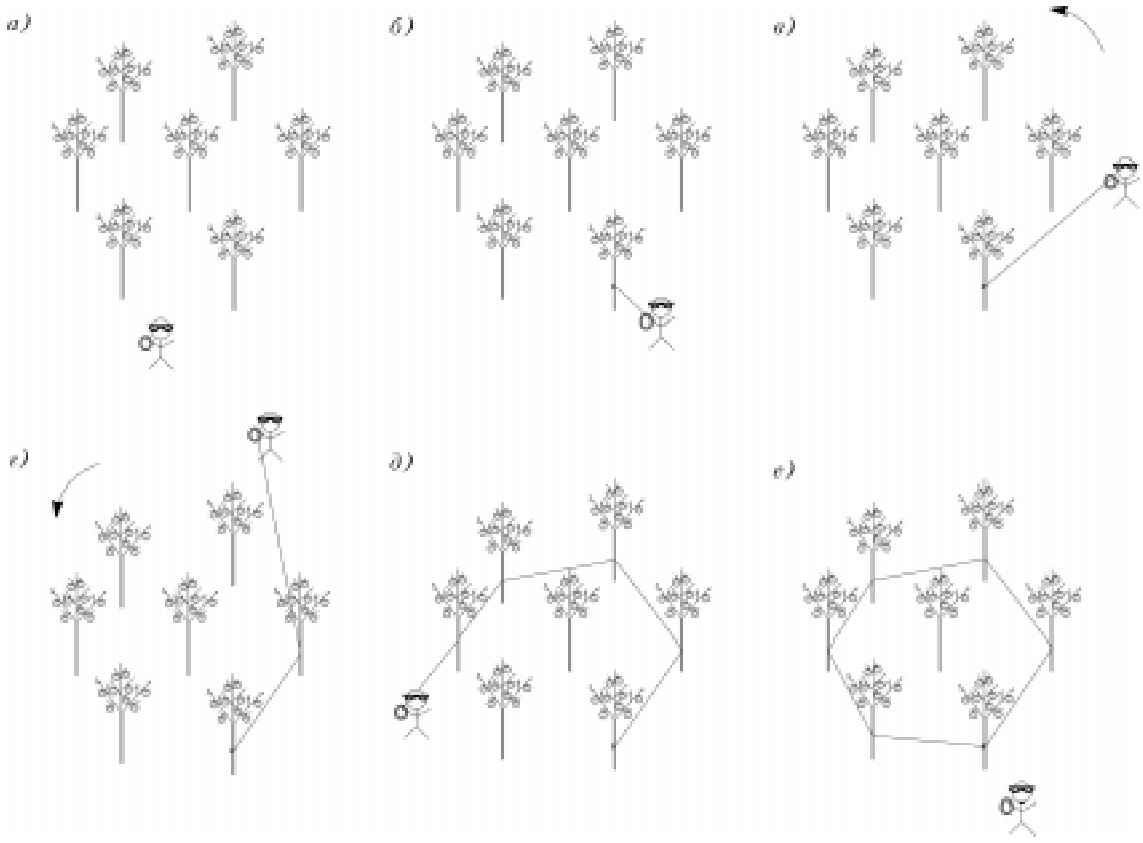


Рис. 2.5. Процесс «заворачивания»

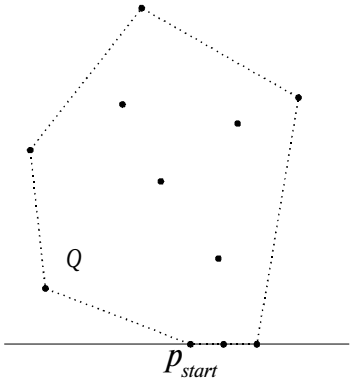


Рис. 2.6. Начальная точка p_{start}

Листинг 2.

```
2.1 | p_start ← p_1
2.2 | for ∀p_i = (x_i, y_i) ∈ Q do
2.3 |   if (y_i < y_start) or ((y_i = y_start) and (x_i < x_start)) then
2.4 |     p_start ← p_i
2.5 |   end-if
2.6 | end-do
```

Рассмотрим вторую строку алгоритма JARVIS-MARCH. В ней говорится, что мы должны выбрать первую точку так, чтобы она являлась вершиной выпуклой оболочки. Оказывается, что точка, имеющая наименьшую абсциссу из всех точек, которые имеют наименьшую ординату, является крайней точкой выпуклой оболочки (попробуйте доказать это утверждение самостоятельно) (рис. 2.6).

Учитывая этот факт, можно детализировать вторую строку алгоритма JARVIS-MARCH следующим образом (листинг 2).

Теперь детализируем процесс «заворачивания» ребер выпуклой оболочки вокруг множества точек. Будем использовать запись $[p_i, p_j]$ для обозначения ориентированного отрезка, направленного из точки p_i (называемой началом ориентированного отрезка) в точку p_j (называемую концом ориентированного отрезка) (листинг 3).

Здесь $p_{current}$ – текущая (последняя найденная) вершина выпуклой оболочки, а p_{next} – следующая за ней вершина выпуклой оболочки (в направлении обхода против часовой стрелки). Запись ADDVERTEX($p_{current}$) добавляет точку $p_{current} = (x_{current}, y_{current})$ в конец последовательности точек, представляющей собой обход выпуклой оболочки против часовой стрелки.

Пусть $p_{current}$ является последней найденной вершиной выпуклой оболочки, а $p_{previous}$ – предпоследней найденной. Рассмотрим луч l, исходящий из вершины $p_{current}$ в том же направлении, что и ориентированный отрезок $[p_{previous}, p_{current}]$ (в случае, когда мы находимся на первом шаге алгоритма, рассмотрим в качестве l горизонтальный луч, исходящий из вершины p_{start} и направленный в сторону увеличения абсцисс). Заметим, что все остальные точки лежат либо на прямой, содержащей луч, либо по левую сторону от него. Зафиксируем произвольную точку t на этом луче, не совпадающую с $p_{current}$. Тогда любой точке p_i множества $Q \setminus \{p_{current}\}$ будет соответствовать угол $\angle p_i p_{current} t$, лежащий в диапазоне $(0; \pi]$. Этот угол называется *полярным углом* точки p_i относительно луча l.

Теперь поставленную перед нами задача в строке 3.4 алгоритма JARVIS-MARCH можно переформулировать следующим образом:

Найти такую точку p_{next}, чтобы угол $\angle p_{next} p_{current} t$ был минимальным.

Листинг 3.

```
3.1 | p_current ← p_start
3.2 | do
3.3 |   ADDVERTEX(p_current)
3.4 |   Найти такую вершину p_next, чтобы она являлась следующей после p_current вершиной
    |     выпуклой оболочки (в направлении обхода против часовой стрелки), то есть любая
    |     точка исходного множества лежала бы по левую сторону от ориентированного
    |     отрезка [p_current, p_next] или на нем
3.5 | p_current ← p_next
3.6 | while p_current ≠ p_start
```

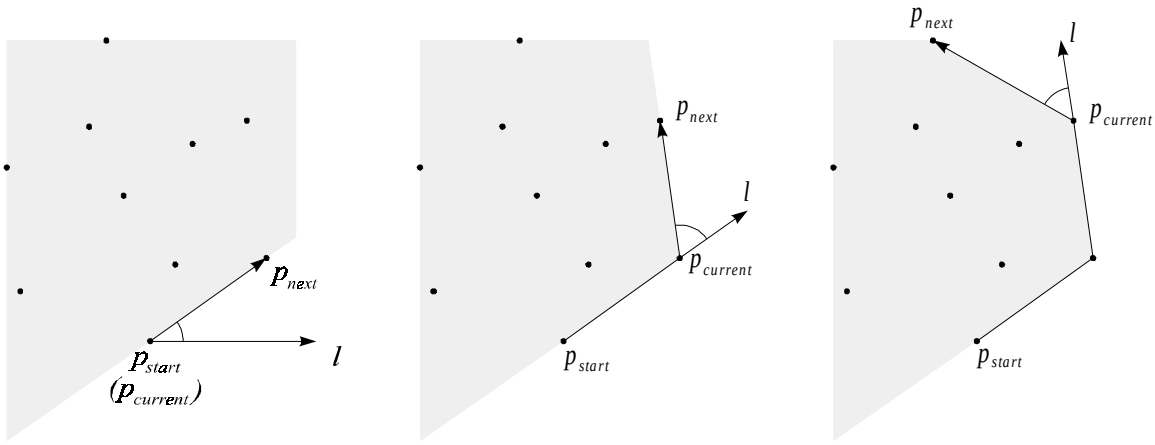


Рис. 2.7. Несколько шагов алгоритма JARVIS-MARCH

Несколько шагов описанного алгоритма проиллюстрированы на рисунке 2.7.

Можно реализовывать данный алгоритм, используя непосредственное вычисление указанных углов с целью их последующего сравнения для нахождения наименьшего. Однако непосредственное вычисление углов сразу же подразумевает использование вещественных чисел и обратных тригонометрических операций. На самом деле, эту задачу можно решить, используя только операции умножения, сложения и сравнения. При этом, если исходные данные задачи целочисленные, все производимые вычисления не выведут за рамки операций с целыми числами. Для этого, нужно использовать понятие *ориентированной площади* треугольника. Рассмотрим следующую подзадачу:

Для ориентированного отрезка $[p_a, p_b]$ *и точки* p_c *определить, лежит ли точка* p_c *на прямой, содержащей данный отрезок, и если нет, то по левую или по правую сторону от него она лежит.*

Чтобы ее решить, рассмотрим выражение:

$$S(p_a, p_b, p_c) = \frac{1}{2}[(x_b - x_a)(y_c - y_a) - (y_b - y_a)(x_c - x_a)].$$

В дальнейшем от данного выражения потребуется только его знак, поэтому множитель 1/2 можно опустить (рис. 2.8).

Оказывается (см. приложение), выражение $S(p_a, p_b, p_c)$ представляет собой так называемую *ориентированную площадь* треугольника $\Delta p_a p_b p_c$. Абсолютная величина $S(p_a, p_b, p_c)$ равна площади треугольника $\Delta p_a p_b p_c$. В случае, когда точка p_c лежит по левую сторону от ориентированного отрезка $[p_a, p_b]$, площадь имеет положительный знак, в случае, когда точка p_c лежит по правую сторону от ориентированного отрезка $[p_a, p_b]$, – знак площади отрицательный, если же точки p_a, p_b и p_c лежат на одной прямой, площадь равна нулю.

Для того чтобы в строке 3.4 алгоритма JARVIS-MARCH найти точку p_{next}, имеющую минимальный угол, то есть точку, которая бы являлась вершиной выпуклой оболочки, и любая точка исходного множества лежала по левую сторону от ориентированного отрезка $[p_{current}, p_{next}]$ или на нем, будем действовать в соответствии со следующим алгоритмом.

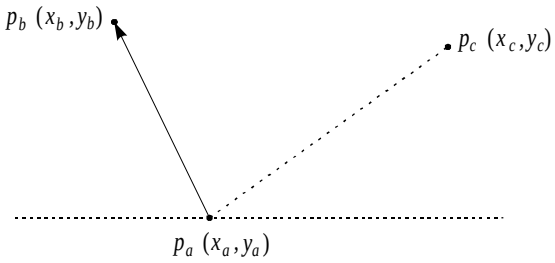


Рис. 2.8. Для такого случая $S(p_a, p_b, p_c)$ будет иметь отрицательный знак

Будем постепенно подбирать искомую точку p_{next} из точек исходного множества Q, отбрасывая на каждом шаге один заведомо неподходящий для нее вариант. Сначала положим p_{next} равной любой точке исходного множества. Будем перебирать все точки исходного множества. Пусть на каком-то шаге перебора мы рассматриваем точку p_i. Тогда должно выполниться одно из трех условий:

1) p_i лежит левее чем $[p_{current}, p_{next}]$. Это означает, что точка p_i заведомо не является той точкой, которую мы ищем.

2) p_i лежит правее чем $[p_{current}, p_{next}]$. Это означает, что текущая точка p_{next} на самом деле не является той точкой, которую мы ищем. Однако мы пока не можем сказать такого о p_i, поскольку все рассмотренные ранее точки лежат или левее ориентированного отрезка $[p_{current}, p_i]$, или на нем. Поэтому на этом шаге положим $p_{next} \leftarrow p_i$.

3) Три точки $p_{current}$, p_i и p_{next} лежат на одной прямой. Причем точки p_i и p_{next} гарантированно лежат по одну сторону от точки $p_{current}$ на этой прямой, поскольку в противном случае она не была бы крайней точкой выпуклой оболочки. Тогда выполняется одно из двух:

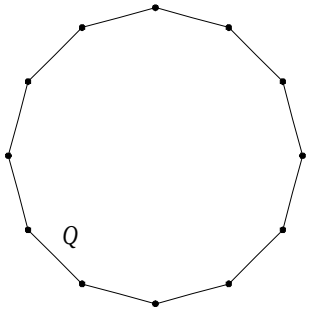


Рис. 2.9. Один из худших случаев для алгоритма Джарвиса

а) точка p_i лежит на открытом отрезке $(p_{current}, p_{next})$ и, следовательно, она не может быть крайней точкой выпуклой оболочки;

б) точка p_{next} лежит на открытом отрезке $(p_{current}, p_i)$. Это означает, что текущая точка *pnext* на самом деле не является той точкой, которую мы ищем. Однако мы пока не можем сказать такого о p_i, поскольку все рассмотренные ранее точки лежат или левее ориентированного отрезка $[p_{current}, p_i]$, или на нем. Поэтому на этом шаге положим $p_{next} \leftarrow p_i$.

После того как в качестве p_i мы переберем все возможные точки с помощью приведенного алгоритма, мы обнаружим p_{next} следующую за $p_{current}$ вершину выпуклой оболочки (в направлении обхода против часовой стрелки). Алгоритмическая запись этого процесса приведена в листинге 4.

Запись AREA($p_{current}$, p_i, p_j) означает вычисление ориентированной площади треугольника $\Delta p_{current} p_i p_j$.

Запись LENGTH(p_i, $p_{current}$) означает вычисление длины отрезка $[p_i, p_{current}]$.

Теперь проанализируем сложность работы алгоритма Джарвиса. Обозначим количество точек исходного множества Q, являющихся вершинами выпуклой оболочки, через h. Заметим, что цикл **while** в строках 3.2–3.6 алгоритма JARVIS-MARCH выполняется в точности h раз, а одна итерация такого цикла требует $O(n)$ операций. Это означает, что время выполнения алгоритма Джарвиса равно $O(hn)$. Любопытный факт заключается в том, что время выполнения данного алгоритма зависит не только от n (размера входных данных), но и от h (размера выходных данных). Если мы заранее знаем, что число h невелико, то этот алгоритм будет очень эффективен (рис. 2.9).

Листинг 4.

```
3.4.1 | p_next ← p_current
3.4.2 | for ∀p_i ∈ Q | i ∈ [1..n] do
3.4.3 |   if (AREA(p_current, p_i, p_next) > 0) or
      |   ((AREA(p_current, p_i, p_next) = 0) and (LENGTH(p_current, p_next) < LENGTH(p_current, p_i))) then
3.4.4 |     p_next ← p_i
3.4.5 |   end-if
3.4.6 | end-do
```

Но так как в некоторых случаях все n точек исходного множества могут быть вершинами выпуклой оболочки ($h = n$), то время выполнения данного алгоритма в худшем случае равно $O(n^2)$. Особо стоит отметить, что существуют алгоритмы для построения выпуклой оболочки в пространствах размерности больше двух (например, в трехмерном случае), использующие ту же идею «заворачивания».

2.3. АЛГОРИТМ ГРЭХЕМА

В 1972 году Рональд Грэхем в одной из первых работ, специально посвященных вопросу разработки эффективных геометрических алгоритмов, показал, что, выполнив предварительно сортировку точек, крайние точки можно найти за линейное время [2]. Использованный им метод стал очень мощным средством в области вычислительной геометрии.

Алгоритм Грэхема использует стек, в котором хранятся точки, являющиеся кандидатами в выпуклую оболочку. Как известно [1], *стек* это динамическая последовательная структура данных, в которой непосредственно доступен только тот элемент, который был добавлен в него последним. В данном алгоритме будет использоваться модификация стека, в которой доступны два последних элемента. Нам понадобятся следующие операции со стеком (стек обозначен за S):

1) PUSH(S; e) добавить элемент e в стек S;

2) POP(S) удалить верхний элемент стека S;

3) SIZE(S) количество элементов, находящихся в стеке S;

4) TOP(S) верхний элемент стека S;

5) NEXT-TO-TOP(S) элемент стека S, который следует за верхним в стеке;

Итак, предположим, что найдена точка p_{start}, заведомо являющаяся вершиной выпуклой оболочки множества точек Q (задача нахождения такой точки уже решалась в алгоритме Джарвиса). Упорядочим остальные точки в соответствии со значениями полярного угла относительно точки p_{start} как начала координат (осью в данной системе координат будет горизонтальный луч, исходящий из точки p_{start} в сторону увеличения абсцисс), а при равенстве полярных углов упорядочим точки по расстоянию от точки p_{start}.

Другими словами введем на оставшихся точках отношение строгого порядка $<$. Для двух точек p_a и p_b будем считать $p_a < p_b$, если точка p_a лежит по правую сторону от ориентированного отрезка $[p_{start}, p_b]$, либо точка p_a лежит на этом отрезке, и длина отрезка $[p_{start}, p_b]$ меньше, чем длина отрезка $[p_{start}, p_b]$. В данном случае использование определения взаимного расположения точки и ориентированного отрезка уместно, поскольку полярные углы точек из множества $Q \setminus \{p_{start}\}$ находятся в пределах $[0; \pi)$ относительно точки p_{start}.

Алгоритм Грэхема состоит из двух этапов: на первом этапе точки сортируются в соответствии с введенным отношением порядка $<$; на втором этапе в процессе обхода упорядоченной последовательности точек отбрасываются те точки, которые гарантированно не являются вершинами выпуклой оболочки. Второй этап алгоритма Грэхема также называется обходом Грэхема (рис. 2.10).

На каждом шаге обхода выпуклая оболочка уже пройденных точек хранится в стеке S, при этом верхний элемент соответ-

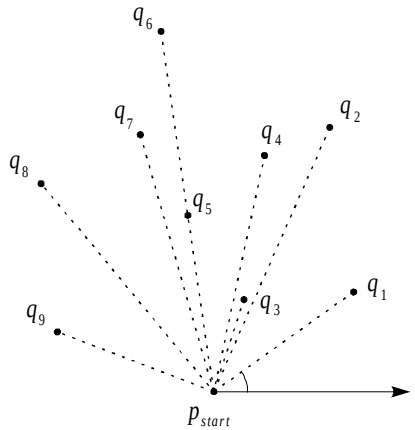


Рис. 2.10. Точки множества $Q \setminus \{p_{start}\}$, упорядоченные в соответствии с введенным отношением $<$ $(i < j \Rightarrow q_i < q_j\ ;\ \forall i; j)$

ствует последней обработанной точке. Когда все точки будут обработаны, в стеке будут находиться все точки выпуклой оболочки (в порядке обхода против часовой стрелки).

Для обхода Грэхема ключевым понятием является понятие *левого поворота*.

Тройка точек (p_a, p_b, p_c) называется *левым поворотом*, если точка p_c лежит строго слева от ориентированного отрезка $[p_a, p_b]$. Если бы мы прямолинейно перемещались от точки p_a до точки p_b, а затем от точки p_b до точки p_c, то в точке p_b нам бы пришлось *повернуть налево*, именно поэтому такой поворот называется левым.

Если обход многоугольника осуществляется против часовой стрелки, то при движении по ребру слева от ребра будет оставаться *внутренность* многоугольника. На рисунке 2.11 закрашенная область соответствует внутренности многоугольника при движении в направлении, указанном стрелкой. Так как выпуклая оболочка является выпуклым многоугольником с вершинами в крайних точках, то для нее допустимы только левые повороты (в случае на рисунке 2.11(*б*) точка p_b не является крайней, а в случае на рисунке 2.11(*в*) многоугольник не будет выпуклым).

Для того, что поддерживать в стеке S выпуклую оболочку уже обработанных точек, нужно поддерживать свойство, что любые три подряд идущих элемента стека образуют левый поворот. При обработке очередной точки q_i это свойство может нарушиться только для тройки точек (NEXT-TO-TOP(S), TOP(S), q_i). Если тройка точек (NEXT-TO-TOP(S), TOP(S), q_i) не образует левый поворот, то необходимо удалить вершину Top(S) из стека, поскольку она не может являться вершиной выпуклой оболочки (см. рисунок 2.11). Удаление необходимо выполнять до тех пор, пока в стеке не останется один элемент или пока тройка точек (Next-To-Top(S), Top(S), q_i) не образует левый поворот. В конце шага в стек добавляется обрабатываемая точка q_i.

Запишем алгоритм формально (листинг 5).

Во второй строке алгоритма мы находим точку p_{start}; все остальные точки множества Q находятся выше не (или на одном уровне, но правее), и потому она заведомо входит в $CH(Q)$. В третьей строке оставшиеся точки множества Q упорядочиваются в соответствии с введенным отношением строгого порядка $<$. Затем в четвертой строке мы помещаем в стек первую точку p_{start}, после чего можем утверждать, что стек содержит выпуклую оболочку множества $\{p_{start}\}$. Дальше на каждой итерации цикла **for**, записанного в строках 5–10, мы добавляем по одной точке к выпуклой оболочке и поддерживаем свойство, что в конце k-ой итерации стек содержит выпуклую оболочку множества точек $\{p_{start}, q_1, q_2, \ldots, q_k\}$ в виде последовательности вершин в порядке обхода против часовой стрелки. Заметим, что в направлении от дна стека S к его верхушке любые три подряд идущих элемента стека s_a, s_b и s_c образуют левый поворот, то есть точка s_c лежит левее ориентированного отрезка $[s_a, s_b]$. После добавления очередной точки в стек это свойство может нарушиться, поэтому может потребоваться удаление нескольких верхних элементов стека (см. рисунок 2.10), что и происходит в

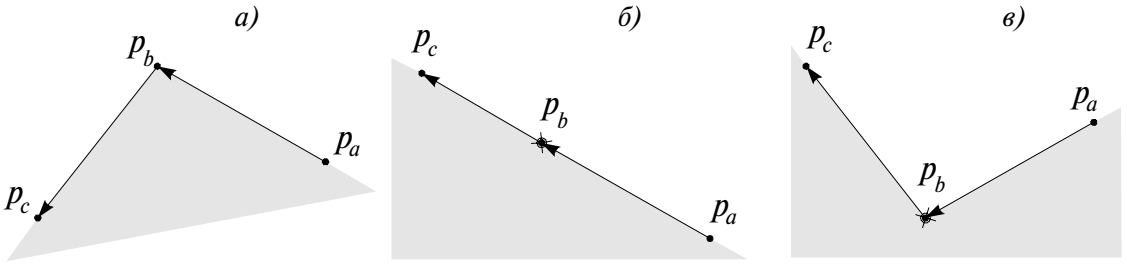


Рис. 2.11. Пример того, как тройка точек (p_a, p_b, p_c) образует (*а*) и не образует (*б*, *в*) левый поворот.

Листинг 5. algorithm GRAHAM-SCAN(Q) $\rightarrow$ $CH(Q)$

Вход. Исходное множество точек $Q = \{p_i \mid p_i = (x_i, y_i),\ i \in [1..n]\}$.

Выход. Выпуклая оболочка множества точек $CH(Q)$, заданная последовательностью вершин (перечисленных в порядке обхода против часовой стрелки).

```
1   Создать пустой стек S
2   Выбрать точку с наименьшей абсциссой среди всех точек имеющих минимальную
    ординату (p_start)
3   Отсортировать все точки множества Q = {p_start} в порядке отношения <.
      В результате получится последовательность точек {q_i}_1^{n-1}, такая что
    q_1 < q_2 < ... < q_{n-2} < q_{n-1}
4   PUSH(S, p_start)
5   for i ← 1 to n – 1 do
6     while SIZE(S) > 1 and три точки (NEXT-TO-TOP(S), TOP(S), q_i) не образуют левый
        поворот do
7       POP(S)
8     end-do
9     PUSH(S, q_i)
10  end-do
11  Вернуть в качестве ответа последовательность точек, содержащуюся в стеке S (дно
      стека будет первым элементом результирующей последовательности, а TOP(S)
      последним)
```

цикле **while** в строках 6–7 алгоритма GRAHAM-SCAN.

На рисунке 2.12 показан пример итерации алгоритма GRAHAM-SCAN. На данной итерации обрабатывается точка q_i. Сначала последовательность точек NEXT-TO-TOP(S) $\rightarrow$ TOP(S) $\rightarrow q_i$ не составляет левый поворот (см. рисунок 2.12 *а*). Следовательно, точка Top(S) не входит в выпуклую оболочку. После двух удалений (см. рисунок 2.12 *б–в*) тройка точек (NEXT-TO-TOP(S), TOP(S), q_i) составляет левый поворот и точка q_i добавляется в стек. В конце итерации стек содержит выпуклую оболочку $\{p_{start}, q_1, q_2, ..., q_i\}$.

Покажем теперь, что время работы алгоритма GRAHAM-SCAN есть $O(n \log n)$. Сортировка точек может быть выполнена за время $O(n \log n)$. Остальная часть алгоритма требует времени $O(n)$. Это не сразу очевидно, поскольку очередная итерация цикла **for** в строках 5–10, помимо добавления одной новой точки, может потребовать удаления нескольких старых. Тем не менее, любая точка q_i добавляется в стек S только один раз, а потому и удаляется не более

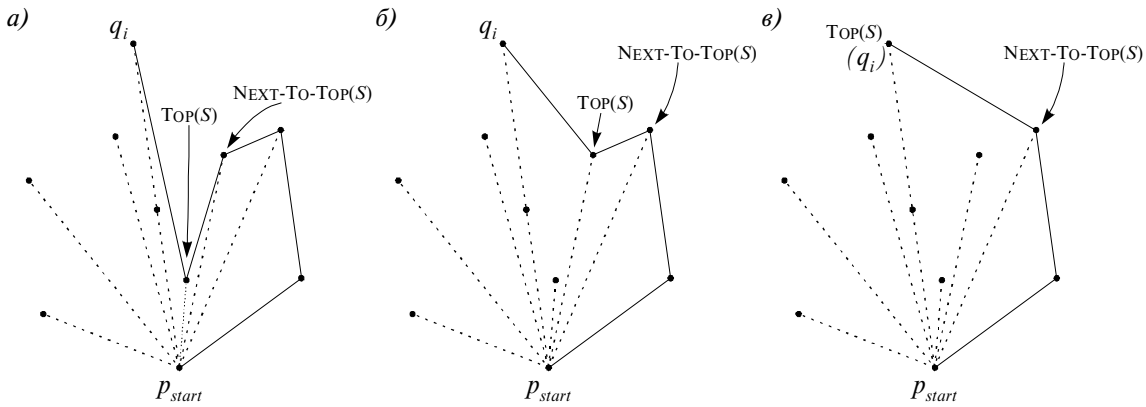
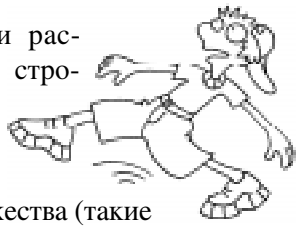


Рис. 2.12. Одна итерация алгоритма GRAHAM-SCAN

одного раза. Тем самым общее время и на добавление, и на удаление есть $O(n)$. Итак, мы приходим к следующему выводу: выпуклая оболочка n точек на плоскости может быть найдена за время $O(n \log n)$ при использовании памяти $O(n)$ с использованием только арифметических операций и сравнений.

2.4. ПОШАГОВЫЙ АЛГОРИТМ ПОСТРОЕНИЯ ВЫПУКЛОЙ ОБОЛОЧКИ



До сих пор были рассмотрены алгоритмы, строящие выпуклую оболочку только после того как известны все точки исходного множества (такие алгоритмы называются алгоритмами в режиме «offline»). В противоположность этому, может потребоваться алгоритм, выдающий ответ немедленно по поступлении очередной точки, не дожидаясь следующей (такие алгоритмы называются рекуррентными алгоритмами, или алгоритмами в режиме «online»). На вход такого алгоритма поступает последовательность из n точек; после поступления очередной точки (но до поступления следующей за ней) требуется указать выпуклую оболочку всех точек, полученных к данному шагу.

Можно применять обход Грэхема на каждом шаге заново, и тогда общее время работы алгоритма будет равно $O(n^2 \log n)$. Можно модифицировать такой алгоритм и не сортировать точки при добавлении новой точки p_i, а вставлять ее в отсортированный массив в соответствии с полярным углом за время $O(i)$, а после выполнить просмотр методом Грэхема также за $O(i)$. Асимптотическая сложность такого алгоритма будет $O(n^2)$. Рассмотрим пошаговый алгоритм, не использующий обход Грэхема.

Будем обрабатывать точки одну за другой по мере поступления, поддерживая на каждом шаге выпуклую оболочку. Обозначим за Q_r множество $\{p_1, ..., p_r\}$, где $r \geq 1$. Рассмотрим шаг алгоритма, на котором точка p_r ($r > 1$) добавляется к уже построенной выпуклой оболочке для множества точек Q_{r-1}. Иными словами, рассмотрим переход от $CH(Q_{r-1})$ к $CH(Q_r)$. Возможны два случая:

1) точка p_r лежит внутри $CH(Q_{r-1})$, либо на ее границе, тогда $CH(Q_r) = CH(Q_{r-1})$;

2) точка p_r лежит вне $CH(Q_{r-1})$. Предположим, что в точке p_r находится точечный источник света. Некоторые ребра $CH(Q_{r-1})$ будут «освещены», а остальные ребра будут находиться «в тени». Освещенные ребра выпуклой оболочки $CH(Q_{r-1})$ образуют цепь из подряд идущих ребер. Освещенная цепь ограничена двумя *опорными точками*, то есть такими точками, у которых одно из инцидентных ребер выпуклой оболочки будет освещено, а второе будет в тени. Через точку p_r и каждую из опорных точек могут быть проведены две вспомогательные прямые, являющиеся опорными к $CH(Q_{r-1})$. Прямая является опорной к выпуклому многоугольнику P, если она проходит через вершину P и внутренняя область многоугольника P полностью лежит по одну сторону от этой прямой. Будем называть опорную точку l левой опорной точкой относительно точки p_r, если слева от опорной прямой, проходящей через точки p_r и l, не лежит точек из $CH(Q_{r-1})$. Аналогично будем называть опорную точку r правой опорной точкой относительно точки p_r, если справа от опорной прямой, проходящей через p_r и r, не лежит точек из $CH(Q_{r-1})$. Опорные точки играют важную роль в преобразовании $CH(Q_{r-1})$ в $CH(Q_r)$: они представляют собой границу между частью выпуклой оболочки, которая должна быть сохранена (неосвещенные ребра), и частью выпуклой оболочки, которая должна быть удалена (освещенные ребра). Освещенные ребра должны быть замещены ребрами, образованными точкой p_r и двумя опорными точками (рис. 2.13).

Для каждой вершины p выпуклой оболочки определим NEXT(p) как следующую за ней вершину в порядке обхода выпуклой оболочки. Аналогично определим вершину PREVIOUS(p), предшествующую вершине p.

Ребром выпуклой оболочки с началом в точке p и концом в точке NEXT(p) называется ориентированный отрезок [p, NEXT(p)].

Освещенность ребра из точки необходимо определить более формально. Ребро $[p_a, p_b]$ освещается из точки p_c, если точка p_c лежит по правую сторону от ориентированного отрезка $[p_a, p_b]$, либо на прямой, содержащей этот отрезок.

Для определения опорных точек необходимо для каждой вершины текущей выпуклой оболочки узнать, является ли она опорной точкой. Вершина p является *левой опорной точкой* относительно точки p_r, если ребро [PREVIOUS(p), p] не освещено из точки p_r, а ребро [p, NEXT(p)] освещено из точки p_r. Аналогично точка p является *правой опорной точкой* относительно p_r, если ребро [PREVIOUS(p), p] освещено из точки p_r, а ребро [p, NEXT(p)] не освещено из точки p_r.

После нахождения опорных точек вершины освещенной цепи, заключенные между опорными точками, должны быть замещены точкой p_r.

Прежде чем углубляться дальше в детали алгоритма, необходимо описать используемые структуры данных. Вершины выпуклой оболочки будем хранить с помощью односвязного кольцевого списка L, поддерживающего операцию NEXT(L, p) для каждого

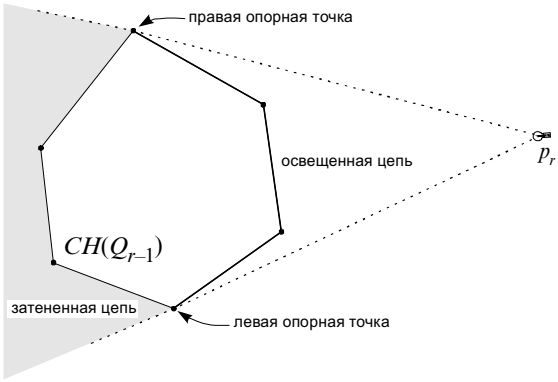


Рис. 2.13. Шаг последовательного алгоритма

своего элемента p. Эта операция возвращает для точки p точку NEXT(p). Чтобы работать со списком надо знать хотя бы одну вершину, принадлежащую ему, для этого вводится операция HEAD(L), возвращающая точку, являющуюся первой в списке.

В листинге 5 представлена алгоритмическая запись пошагового алгоритма.

Детализируем добавление точки p_1 в пустой список L (вторая строка алгоритма SEQUENTIAL) (листинг 6).

После этого шага HEAD(L) и NEXT(L, p_1) вернут p_1.

Листинг 5. algorithm SEQUENTIAL(Q) $\rightarrow [CH(Q_r)]_1^n$

Вход. Последовательность точек $Q = [p_i \mid p_i = (x_i, y_i),\ i \in [1..n]]$.

Выход. Последовательность выпуклых оболочек $\{CH(Q_r)\}_1^n$.

Каждая выпуклая оболочка $CH(Q_r)$ задана последовательностью вершин (перечисленных в порядке обхода против часовой стрелки).

```
1  | Создать пустой односвязный кольцевой список L
2  | Добавить в L точку p_1
3  | Зафиксировать в качестве CH(Q_1) последовательность точек, записанную в списке L
4  | for i ← 2 to n do
5  |   Найти вершины l и r текущей выпуклой оболочки, являющиеся левой и правой
   |     опорными точками относительно точки p_i
6  |   if Опорные точки l и r существуют then
7  |     Заменить элементы списка L в промежутке с l по r на вершину p_i
8  |   end-if
9  |   Зафиксировать в качестве CH(Q_i) последовательность точек, записанную в списке L
10 | end-do
```

Листинг 6.

```
2.1 | Сделать p_1 головой списка L
2.2 | Циклически замкнуть список L
```

Поиск опорных точек l и r относительно точки p_i будем осуществлять следующим образом (пятая строка алгоритма SEQUENTIAL) (листинг 7).

Замена вершин списка L в промежутке с l по r вершиной p_i можно осуществлять следующим образом (седьмая строка алгоритма SEQUENTIAL) (листинг 8).

Оценим асимптотическую сложность алгоритма SEQUENTIAL. Поиск опорных точек для точки p_i имеет сложность $O(i)$ (пятая строка), а замена в списке – сложность $O(1)$ (седьмая строка). Сложность поддержания выпуклой оболочки на i-ом шаге составляет $O(i)$. Таким образом, суммарная сложность алгоритма SEQUENTIAL составляет $O(n^2)$.

Если хранить последовательность вершин выпуклой оболочки в виде сбалансированного дерева поиска [1], то операции нахождения опорных точек, вставки и удаления цепочки вершин на i-ом шаге можно осуществлять за время $O(\log i)$; таким образом, можно достичь суммарной сложности алгоритма $O(n \log n)$.

ЗАКЛЮЧЕНИЕ

Рассмотренные алгоритмы построения выпуклой оболочки на плоскости являются в некотором смысле естественными и простыми для понимания. Существуют и другие алгоритмы решения этой задачи. В продолжение данной статьи будут рассмотрены некоторые из них, а также тесная связь задачи построения выпуклой оболочки с задачей *сортировки*. Это позволит определить так называемую нижнюю оценку сложности задачи построения выпуклой оболочки и оптимальные по сложности алгоритмы ее решения.

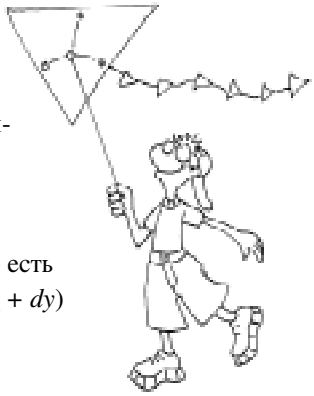
Листинг 7.

```
5.1  p ← Head(L)
5.2  do
5.3     nextp ← Next(L, p)
5.4     nextnextp ← Next(L, nextp)
5.5     b1 ← ребро [p, nextp] освещено из точки pi
5.6     b2 ← ребро [nextp, nextnextp] освещено из точки pi
5.7     if (not b1) and b2 then l ← nextp
5.8     if b1 and (not b2) then r ← nextp
5.9     p ← nextp
5.10 while p ≠ HEAD(L)
```

Листинг 8.

```
7.1  Для вершины l переставить ссылку, указывающую на следующий элемент,
       на вершину pi
7.2  Для вершины pi переставить ссылку, указывающую на следующий элемент,
       на вершину r
7.3  Сделать pi головой списка L
```

Приложение.



ОРИЕНТИРОВАННАЯ ПЛОЩАДЬ ТРЕУГОЛЬНИКА

Обратимся к выражению для ориентированной площади треугольника $\Delta p_a p_b p_c$:

$$S(p_a, p_b, p_c) = \frac{1}{2}[(x_b - x_a)(y_c - y_a) - (y_b - y_a)(x_c - x_a)]$$

Заметим, что если мы выполним параллельный перенос на (dx, dy), то есть заменим точки p_a, p_b и p_c на точки $p'_a = (x_a + dx, y_a + dy)$, $p'_b = (x_b + dx, y_b + dy)$ и $p'_c = (x_c + dx, y_c + dy)$, то для любых dx и dy будет верно равенство

$$S(p_a, p_b, p_c) = S(p'_a, p'_b, p'_c).$$

В этом можно легко убедиться, подставив соответствующие значения в выражение для вычисления ориентированной площади.

Вместо того, чтобы рассматривать точки $p_a = (x_a, y_a)$, $p_b = (x_b, y_b)$ и $p_c = (x_c, y_c)$, рассмотрим три точки $p_0 = (0, 0)$, $p_1 = (x_1, y_1)$ и $p_2 = (x_2, y_2)$. Достаточно доказать, что ориентированная площадь вычисляется верно для всех таких троек точек, поскольку параллельным переносом мы всегда можем совместить точку p_a с точкой p_0, а точки p_b и p_c перейдут в p_1 и p_2 соответственно ($x_1 = x_b - x_a$, $y_1 = y_b - y_a$, $x_2 = x_c - x_a$, $y_2 = y_c - y_a$).

Простой подстановкой получим:

$$S(p_0, p_1, p_2) = \frac{1}{2}(x_1 y_2 - x_2 y_1).$$

Заметим, что площадь треугольника равна половине площади параллелограмма, построенного на двух его сторонах $[p_0, p_1]$ и $[p_0, p_2]$. В дальнейшем мы рассмотрим три принципиально разных случая взаимного расположения точек p_0, p_1 и p_2. Достаточно будет рассмотреть только те случаи, в которых знак ориентированной площади отрицательный, поскольку при рассмотрении выражения очевидно, что

$$S(p_0, p_1, p_2) = -S(p_0, p_2, p_1).$$

Также очевидно, что $S(p_0, p_1, p_2) = 0$ только в том случае, когда все три точки p_0, p_1 и p_2 лежат на одной прямой.

1) Точки p_1 и p_2 лежат в одном квадранте. Не умаляя общности, будем считать, что точки p_1 и p_2 лежат в первом квадранте ($x_1 \geq 0$, $x_2 \geq 0$, $y_1 \geq 0$, $y_2 \geq 0$). Для случая других квадрантов вывод аналогичен.

Как показано на рис. 2.14 *а–в*, площадь параллелограмма можно выразить как разность площадей на рис. 2.14 *б* и 2.14 *в*:

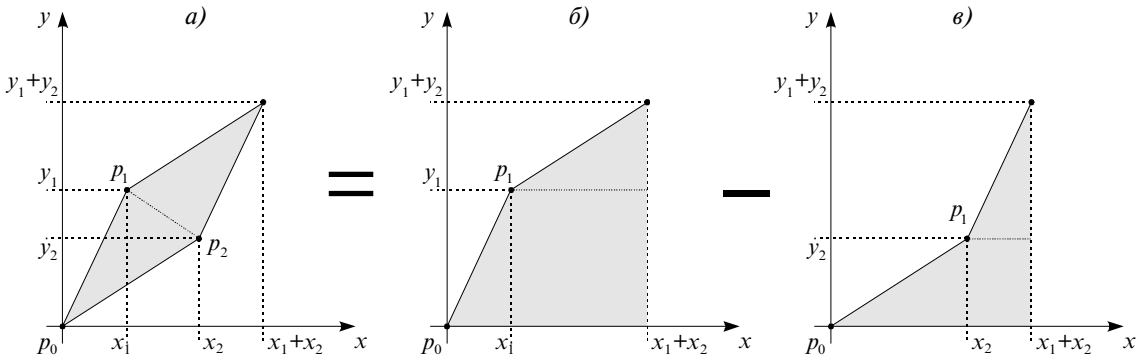


Рис. 2.14. Точки p_1 и p_2 лежат в одном квадранте.

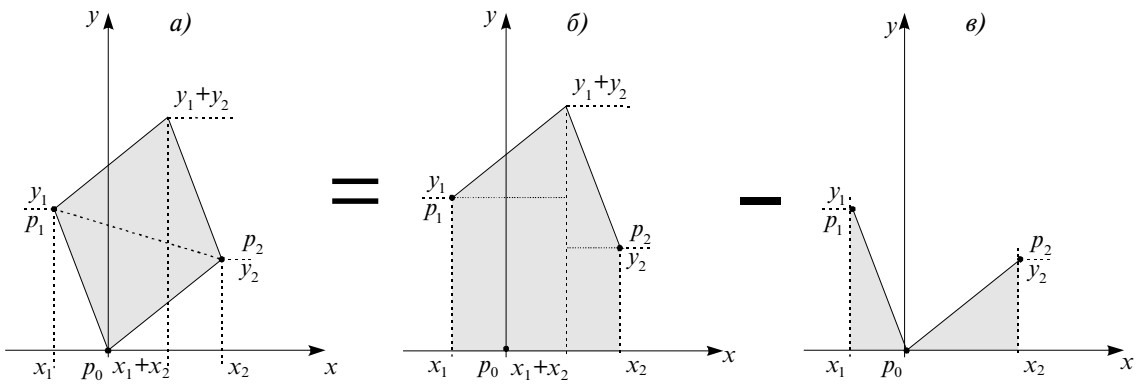


Рис. 2.15. Точки p_1 и p_2 лежат в соседних квадрантах.

$$S_\Diamond = (\frac{1}{2}x_1y_1 + \frac{1}{2}x_2y_2 + x_2y_1) - (\frac{1}{2}x_2y_2 + \frac{1}{2}x_1y_1 + x_1y_2) = -(x_1y_2 - x_2y_1).$$

Под $S_\Diamond$ подразумевается неориентированная площадь параллелограмма (то есть $S_\Diamond$ – неотрицательная величина).

$$S(p_0, p_1, p_2) = -\frac{1}{2}S_\Diamond,$$

так как точка p_2 лежит по правую сторону от ориентированного отрезка $[p_0, p_1]$.

2) Точки p_1 и p_2 лежат в соседних квадрантах. Не умаляя общности, будем считать, что точки p_1 и p_2 лежат во втором и первом квадрантах, соответственно ($x_1 \le 0$, $x_2 \ge 0$, $y_1 \ge 0$, $y_2 \ge 0$). Для случая других квадрантов вывод аналогичен.

Как показано на рис. 2.15 *а–в*, площадь параллелограмма можно выразить как разность площадей на рис. 2.15 *б* и 2.15 *в*. Заметим, что x_1 – отрицательное число:

$$S_\Diamond = (\frac{1}{2}x_2y_2 + x_2y_1 + \frac{1}{2}(-x_1)y_1 + (-x_1)y_2) - (\frac{1}{2}(-x_1)y_1 + \frac{1}{2}x_2y_2) = -(x_1y_2 - x_2y_1).$$

Под $S_\Diamond$ подразумевается неориентированная площадь параллелограмма (то есть $S_\Diamond$ – неотрицательная величина).

$$S(p_0, p_1, p_2) = -\frac{1}{2}S_\Diamond,$$

так как точка p_2 лежит по правую сторону от ориентированного отрезка $[p_0, p_1]$.

3) Точки p_1 и p_2 лежат в противоположных квадрантах. Не умаляя общности, будем считать, что точки p_1 и p_2 лежат в третьем и первом квадрантах, соответственно ($x_1 \le 0$, $x_2 \ge 0$, $y_1 \le 0$, $y_2 \ge 0$). Для случая других квадрантов вывод аналогичен.

Параллельным переносом сдвинем точки p_0, p_1 и p_2 так, чтобы точка p_1 совпала с началом координат.

Тогда, точки p_1 и p_2 будут находиться в одном квадранте.

Покажем, что

$$S(p_0, p_1, p_2) = -S(p_1, p_0, p_2).$$

По определению ориентированной площади, $|S(p_0, p_1, p_2)| = |S(p_1, p_0, p_2)|$. Если $S(p_0, p_1, p_2) = 0$, то, очевидно, $S(p_1, p_0, p_2) = 0$. Смена знака обусловлена тем, что, если точка p_2 лежит строго по левую сторону от ориентированного отрезка $[p_0, p_1]$, то она будет лежать строго по правую сторону от ориентированного отрезка $[p_1, p_0]$, и наоборот.

Выражение $S(p_1, p_0, p_2)$ вычисляется корректно, поскольку оно удовлетворяет первому рассмотренному случаю. Значит и $S(p_0, p_1, p_2)$ вычисляется корректно.

Мы получили формулу для ориентированной площади треугольника $\Delta p_a p_b p_c$ из сугубо геометрических соображений. Можно было бы сделать вывод короче, введя понятие векторного произве-

дения. Векторное произведение двумерных векторов $\vec{a}$ и $\vec{b}$ обозначается как $[\vec{a}, \vec{b}]$. Если известны координаты векторов $\vec{a} = (a_x, a_y)$ и $\vec{b} = (b_x, b_y)$, то $[\vec{a}, \vec{b}]$ можно вычислить через них: $[\vec{a}, \vec{b}] = a_x b_y - a_y b_x$.

В свою очередь, ориентированная площадь $S(p_a, p_b, p_c)$ может быть выражена через векторное произведение. Здесь за $\overrightarrow{p_a p_b}$ обозначен вектор, с началом в точке p_a и концом в точке p_b, имеющий координаты $\overrightarrow{p_a p_b} = (p_{b_x} - p_{a_x}, p_{b_y} - p_{a_y})$:

$$S(p_a, p_b, p_c) = \frac{1}{2}[\overrightarrow{p_a p_b}, \overrightarrow{p_a p_c}].$$

Литература

1. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: Построение и анализ, М.: МЦНМО, 2000. 960 с.

2. *Препарата Ф., Шеймос М.* Вычислительная геометрия: Введение, М.: Мир, 1989. 480 с.

3. *Костовский А.П.* Геометрические построения одним циркулем, М.: Наука, 1984. 80 с.

4. *Киселев А.П.* Элементарная геометрия. Для средних учебных заведений. 1914. 404 с.

5. *Amenta N, al.* Application Challenges for Computational Geometry. CG Impact Task Force Report., 1996.

Ивановский Сергей Алексеевич,
кандидат технических наук,
доцент кафедры Математического обеспечения и применения ЭВМ СПбГЭТУ «ЛЭТИ»,

Преображенский Алексей Семенович,
студент 6-го курса СПбГЭТУ «ЛЭТИ» (магистратура),

Симончик Сергей Константинович,
магистр, студент 6-го курса СПбГЭТУ «ЛЭТИ» (магистратура).