3

Строительные блоки алгоритмов

Мы создаем программное обеспечение для решения задач. Но программисты часто слишком сосредоточены на решении задачи, чтобы выяснять, нет ли уже готового ее решения. Даже если программист знает, что задача уже была решена, не очевидно, что существующий код будет соответствовать конкретной задаче, с которой работает программист. В конечном счете не так уж легко найти код на данном языке программирования, который мог бы быть легко изменен для решения конкретной задачи.

Размышлять об алгоритмах можно по-разному. Многие практики просто ищут алгоритм в книге или на веб-сайтах, копируют код, запускают его (а может быть, даже проверяют), а затем переходят к следующей задаче. По нашему мнению, такой процесс ничуть не улучшает понимание алгоритмов. В самом деле, такой подход может вывести вас на кривую дорожку неверного выбора конкретной реализации алгоритма.

Вопрос заключается не только в том, как найти правильный алгоритм для решения вашей задачи, но и в том, чтобы хорошо в нем разобраться и понять, как он работает, чтобы убедиться, что вы сделали правильный выбор. И, когда вы выбрали алгоритм, как эффективно его реализовать? Каждая глава нашей книги объединяет набор алгоритмов для решения стандартной задачи (например, для сортировки или поиска) или связанных с ней задач (например, поиска пути). В этой главе мы представим формат, используемый нами для описания алгоритмов в этой книге. Мы также подытожим общие алгоритмические подходы, используемые для решения программных задач.

Формат представления алгоритма

Реальная мощь применения шаблона для описания каждого алгоритма проявляется в том, что вы можете быстро сравнивать различные алгоритмы и выявлять общие особенности в, казалось бы, совершенно разных алгоритмах. Каждый алгоритм в книге представлен с использованием фиксированного набора разделов, соответствующих упомянутому шаблону. Мы можем также пропустить некоторый раздел, если он не добавляет ничего ценного к описанию алгоритма, или добавить разделы для освещения некоторой конкретной особенности алгоритма.

Название и краткое описание

Описательное название алгоритма. Мы используем это название для лаконичных ссылок одних алгоритмов на другие. Когда мы говорим об использовании **последовательного поиска**, это название точно передает, о каком именно алгоритме поиска мы говорим. Название каждого алгоритма всегда выделено полужирным шрифтом.

Входные и выходные данные алгоритма

Описывает ожидаемый формат входных данных алгоритма и вычисленных им значений.

Контекст применения алгоритма

Описание задачи, которое показывает, когда алгоритм является полезным и когда его производительность будет наилучшей. Описание свойств задачи/решения, которые необходимо рассмотреть для успешной реализации алгоритма. Это те соображения, которые должны привести вас к решению о выборе данного конкретного алгоритма для вашей задачи.

Реализация алгоритма

Описание алгоритма с использованием реального рабочего кода с документацией. Все исходные тексты решений можно найти в репозитории к книге.

Анализ алгоритма

Краткий анализ алгоритма, включая данные о производительности и информацию, которая должна помочь вам понять поведение алгоритма. Хотя раздел, посвященный анализу, не предназначен для формального доказательства указанной производительности алгоритма, вы должны быть в состоянии понять, почему алгоритм ведет себя именно таким образом. Чтобы пояснить описанное поведение алгоритмов,

мы предоставляем ссылки на работы, в которых представлены соответствующие леммы и доказательства.

Вариации алгоритма

Представляет вариации алгоритма или различные альтернативы.

Формат шаблона псевдокода

Каждый алгоритм в этой книге представлен с примерами кода, которые демонстрируют его реализацию на основных языках программирования, таких как Python, C, C++ и Java. Для читателей, которые не знакомы со всеми этими языками, мы описываем каждый алгоритм с помощью псевдокода с небольшим примером, показывающим его выполнение.

Рассмотрим следующий пример описания производительности, которое именует алгоритм и четко классифицирует его производительность для всех трех случаев (наилучший, средний, наихудший), описанных в главе 2, "Математика алгоритмов".

Последовательный поиск

Наилучший: $O(1)$; средний, наихудший: $O(n)$

```
search (A,t)
    for i=0 to n-1 do          ❶
        if A[i] = t then
            return true
    return false
end
```

❶ Последовательное обращение к каждому элементу от 0 до n-1.

Описание псевдокода преднамеренно краткое. Ключевые слова и имена функций приведены с использованием полужирного шрифта. Все переменные имеют имена, записываемые строчными буквами, в то время как массивы записываются прописными буквами, а обращение к их элементам выполняется с помощью индексной записи `[i]`. Отступы в псевдокоде показывают области видимости инструкций `if` и циклов `while` и `for`.

Перед тем как рассматривать исходные тексты реализаций на конкретных языках программирования, следует ознакомиться с кратким резюме алгоритма. После каждого резюме следует небольшой пример (наподобие показанного на рис. 3.1), призванный облегчить понимание работы алгоритма. Эти рисунки показывают выполнение алгоритмов в динамике, часто с изображением основных шагов алгоритма, представленных на рисунке сверху вниз.

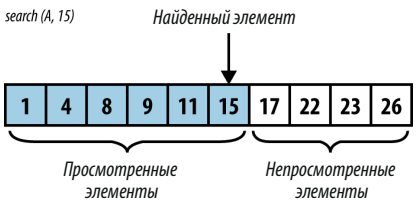

Рис. 3.1. Пример выполнения последовательного поиска

Формат эмпирических оценок

Мы подтверждаем производительность каждого алгоритма путем выполнения ряда хронометражей решения задач, соответствующих индивидуальным алгоритмам. Приложение A содержит более подробную информацию о механизмах, используемых для хронометража. Для правильной оценки производительности набор тестов состоит из множества k индивидуальных испытаний (обычно $k \geq 10$). Лучшее и худшее выполнения отбрасываются, остальные же k–2 испытания проходят статистическую обработку с вычислением среднего значения и стандартного отклонения. Экземпляры задач, приводимые в таблицах, обычно имеют размеры от $n = 2$ до 2^{20}.

Вычисления с плавающей точкой

Поскольку ряд алгоритмов в этой книге включает числовые вычисления, нам нужно описать мощь и ограничения современных компьютеров в этой области. Компьютеры выполняют основные вычисления со значениями, хранящимися в регистрах центрального процессора (CPU). По мере того, как компьютерные архитектуры эволюционировали от 8-разрядных процессоров Intel, популярных в 1970-х годах, до сегодняшних получивших широкое распространение 64-битных архитектур (например, процессоры Intel Itanium и Sun Microsystems Sparc), эти регистры существенно выросли в размерах. Процессор часто поддерживает базовые операции — такие, как сложение, умножение, вычитание и деление — для целочисленных значений, хранящихся в этих регистрах. Устройства для вычислений с плавающей точкой (FPU) могут эффективно выполнять вычисления в соответствии со стандартом IEEE для бинарной арифметики с плавающей точкой (IEEE 754).

Математические вычисления на основе целочисленных значений (таких, как логические значения, 8-битные байты и 16- и 32-разрядные целые числа) традиционно являются наиболее эффективными вычислениями, выполняемыми процессором. Программы часто оптимизированы таким образом, чтобы использовать эту историческую разницу в производительности между вычислениями с целыми числами и с числами с плавающей точкой. Однако в современных процессорах

производительность вычислений с плавающей точкой значительно улучшена. Поэтому важно, чтобы разработчики были ознакомлены со следующими вопросами программирования арифметики с плавающей точкой [28].

Производительность

Общепризнано, что вычисления с целочисленными значениями будут более эффективными, чем аналогичные вычисления с плавающей точкой. В табл. 3.1 перечислены значения времени выполнения 10 000 000 операций, включая результаты в Linux времен первого издания этой книги и результаты Sparc Ultra-2 1996 года. Как видите, производительность отдельных операций может значительно отличаться от платформы к платформе. Эти результаты показывают также огромные улучшения процессоров в течение последних двух десятилетий. Для некоторых результатов указано нулевое время выполнения, так как они быстрее, чем имеющиеся средства хронометража.

Таблица 3.1. Время (в секундах) выполнения 10 000 000 операций

Операция	Sparc Ultra-2	Linux i686	В настоящее время
32-битное целое `CMP`	0,811	0,0337	0,0000
32-битное целое `MUL`	2,372	0,0421	0,0000
32-битное `float MUL`	1,236	0,1032	0,02986
64-битное `double MUL`	1,406	0,1028	0,02987
32-битное `float DIV`	1,657	0,1814	0,02982
64-битное `double DIV`	2,172	0,1813	0,02980
128-битное `double MUL`	36,891	0,2765	0,02434
32-битное целое `DIV`	3,104	0,2468	0,0000
32-битное `double SQRT`	3,184	0,2749	0,0526

Ошибка округления

Все вычисления с использованием значений с плавающей точкой могут привести к ошибкам округления, вызванным бинарным представлением чисел с плавающей точкой. В общем случае число с плавающей точкой является конечным представлением, разработанным для приближения действительного числа, бинарное представление которого может быть бесконечным. В табл. 3.2 показана информация о представлениях чисел с плавающей точкой и представлении конкретного значения `3.88f`.

Таблица 3.2. Представления чисел с плавающей точкой

Тип	Знак, бит	Экспонента, бит	Мантисса, бит
`float`	1	8	23
`double`	1	11	52
Пример бинарного представления `3.88f` как `0x407851ec`			
`01000000 01111000 01010001 11101100` (всего 32 бита)			
`seeeeeee emmmmmmm mmmmmmmm mmmmmmmm`			

Тремя следующими за `3.88f` 32-битными представлениями чисел с плавающей точкой (и их значениями) являются:

- `0x407851ed`: `3.8800004`
- `0x407851ee`: `3.8800006`
- `0x407851ef`: `3.8800008`

Вот некоторые случайным образом выбранные 32-битные значения с плавающей точкой:

- `0x1aec9fae`: `9.786529E-23`
- `0x622be970`: `7.9280355E20`
- `0x18a4775b`: `4.2513525E-24`

В 32-разрядном значении с плавающей точкой один бит используется для знака, 8 битов — для экспоненты и 23 бита — для мантиссы. В представлении Java "степень двойки может определяться путем трактовки битов экспоненты как положительного числа с последующим вычитанием из него значения смещения. Для типа float смещение равно 126" [59]. Если сохраненная экспонента равна 128, ее фактическое значение равно 128 – 126, или 2.

Для достижения наибольшей точности мантисса всегда нормализована таким образом, что крайняя слева цифра всегда равна 1; этот бит *в действительности не должен храниться*, но воспринимается FPU как часть числа. В предыдущем примере мантисса представляет собой

`.[1]11110000101000111101100` =

[1/2] + 1/4 + 1/8 + 1/16 + 1/32 + 1/1 024 + 1/4 096 + 1/65 536 + 1/131 072 + 1/262 144 + 1/524 288+
+ 1/20974152 + 1/4 194 304,

что равно в точности значению `0.970000028610229492187`5.

При хранении с использованием этого представления значения `3.88f` приближенное значение равно $+1 \cdot 0{,}970000028610229492187_5 \cdot 2^2$, т.е. в точности `3.88000011444091796875`. Ошибка этого значения составляет `~0.0000001`. Наиболее распространенный способ описания ошибок чисел с плавающей точкой — использование *относительной ошибки*, которая вычисляется как отношение абсолютной ошибки к точному значению. В данном случае относительная ошибка составляет `0.000000114409179687`5/`3.88`, или `2.9E-8`. Относительные ошибки достаточно часто оказываются меньше одной миллионной.

Сравнение значений с плавающей точкой

Поскольку значения с плавающей точкой являются всего лишь приблизительными, простейшие операции с плавающей точкой становятся подозрительными. Рассмотрим следующую инструкцию:

```
if (x == y) { ... }
```

Действительно ли эти два числа с плавающей запятой должны быть точно равны? Или достаточно, чтобы они были просто приблизительно равны (что обычно описывается знаком "≈")? Не может ли случиться так, что два значения хотя и различны, но достаточно близки, так что они должны рассматриваться как одно и то же значение? Рассмотрим практический пример: три точки, $p_0=(a,b)$, $p_1=(c,d)$ и $p_2=(e,f)$, в декартовой системе координат определяют упорядоченную пару отрезков (p_0,p_1) и (p_1,p_2). Значение выражения $(c-a)(f-b)-(d-b)(e-a)$ позволяет определить, коллинеарны ли эти два отрезка (т.е. находятся ли они на одной линии). Если это значение

- $=0$, то отрезки коллинеарны;
- <0, то отрезки повернуты влево (против часовой стрелки);
- >0, то отрезки повернуты вправо (по часовой стрелке).

Чтобы показать, как при вычислениях Java могут возникать ошибки с плавающей точкой, определим три точки, используя значения от *a* до *f* из табл. 3.3.

Таблица 3.3. Арифметические ошибки с плавающей точкой

	32-битное значение (`float`)	**64-битное значение (`double`)**
$a = 1/3$	`0.33333334`	`0.3333333333333333`
$b = 5/3$	`1.6666666`	`1.6666666666666667`
$c = 33$	`33.0`	`33.0`
$d = 165$	`165.0`	`165.0`
$e = 19$	`19.0`	`19.0`
$f = 95$	`95.0`	`95.0`
$(c-a)(f-b)-(d-b)(e-a)$	`4.8828125E-4`	`-4.547473508864641E-13`

Как вы можете легко определить, три точки, p_0, p_1 и p_2, коллинеарны и лежат на прямой $y=5\cdot x$. При вычислении тестового значения с плавающей точкой для проверки коллинеарности на результат вычисления влияют имеющиеся ошибки, присущие арифметике с плавающей точкой. Использование при вычислениях 32-битных значений с плавающей точкой дает значение 0,00048828125; использование 64-битных значений дает очень небольшое отрицательное число. Этот пример показывает, что как 32-, так и 64-разрядные представления чисел с плавающей точкой не дают истинные значения математических вычислений. И в рассмотренном случае результаты вычислений приводят к разногласиям по поводу того, представляют ли точки угол по часовой стрелке, против часовой стрелки или коллинеарные отрезки. Таков уж мир вычислений с плавающей точкой.

Одним из распространенных решений для этой ситуации является введение небольшого значения δ для определения операции "≈" (приблизительного равенства) между двумя значениями с плавающей точкой. Согласно этому решению, если $|x-y|<\delta$, то мы считаем *x* и *y* равными. Тем не менее даже при такой простой мере

возможна ситуация, когда $x \approx y$ и $y \approx z$, но при этом условие $x \approx z$ не выполняется. Это нарушает математический принцип *транзитивности* и осложняет написание правильного кода. Кроме того, это решение не решает задачу коллинеарности, в которой при принятии решения используется знак полученного значения (нулевое, положительное или отрицательное).

Специальные значения

Хотя все возможные 64-битные значения могут представлять допустимые числа с плавающей точкой, стандарт IEEE определяет несколько значений (табл. 3.4), которые интерпретируются как специальные значения (и часто не могут участвовать в стандартных математических вычислениях, таких как сложение или умножение). Эти значения были разработаны для упрощения восстановления после распространенных ошибок, таких как деление на нуль, квадратный корень из отрицательного числа, переполнение и потеря значимости. Обратите внимание, что несмотря на то, что они могут использоваться в вычислениях, значения положительного и отрицательного нулей также включены в эту таблицу.

Таблица 3.4. Специальные значения IEEE 754

Специальное значение	64-битное представление IEEE 754
Положительная бесконечность	`0x7ff0000000000000L`
Отрицательная бесконечность	`0xfff0000000000000L`
Не число (NaN)	От `0x7ff0000000000001L` до `0x7fffffffffffffffL` и от `0xfff0000000000001L` до `0xffffffffffffffffL`
Отрицательный нуль	`0x8000000000000000`
Положительный нуль	`0x0000000000000000`

Эти специальные значения могут быть результатом вычислений, которые выходят за пределы допустимых границ представлений чисел. Выражение `1/0.0` в Java вычисляется как положительная бесконечность. Если вместо этого инструкция будет записана как `double x = 1/0`, то виртуальная машина Java сгенерирует исключение `ArithmeticException`, поскольку это выражение выполняет целочисленное деление двух чисел.

Пример алгоритма

Чтобы проиллюстрировать наш шаблон алгоритма, опишем алгоритм **сканирования Грэхема** для вычисления выпуклой оболочки множества точек. Это задача, представленная в главе 1, “Мысли алгоритмически”, и показанная на рис. 1.3.

Название и краткое описание

Сканирование Грэхема вычисляет выпуклую оболочку множества точек на декартовой плоскости. Алгоритм находит нижнюю точку *low* во входном множестве *P* и сортирует оставшиеся точки {*P*–*low*} по полярному углу относительно нижней точки в *обратном* порядке. Этот порядок позволяет алгоритму обойти *P* по часовой стрелке от самой низкой точки. Каждый левый поворот для последних трех точек в строящейся оболочке показывает, что последняя точка оболочки была выбрана неправильно и может быть удалена.

Входные и выходные данные алгоритма

Экземпляр задачи построения выпуклой оболочки определяется множеством точек *P*.

Вывод представляет собой последовательность точек (x,y), представляющую обход выпуклой оболочки по часовой стрелке. Какая именно точка выбрана первой, значения не имеет.

Контекст применения алгоритма

Этот алгоритм подходит для декартовых точек. Если точки используют иную систему координат, например такую, в которой бо́льшие значения *y* отражают более низкие точки на плоскости, то алгоритм должен соответственно вычислять нижнюю точку *low*. Сортировка точек по полярному углу требует тригонометрических вычислений.

Сканирование Грэхема

Наилучший, средний, наихудший случаи: $O(n \cdot \log n)$

```
graham(P)
    low = точка с минимальной координатой y в P                    ❶
    Удаляем low из P
    Сортируем P по убыванию полярного угла относительно low        ❷

    hull = {P[n-2], low}                                           ❸
    for i = 0 to n-1 do
      while(isLeftTurn(secondLast(hull),last(hull),P[i])) do
        Удаляем последнюю точку из оболочки                        ❹

        Добавляем P[i] к оболочке
    Удаляем дубликат последней точки                               ❺
    Возвращаем оболочку
```

❶ Точки с одинаковыми значениями y сортируются по координате x
❷ P[0] имеет максимальный полярный угол, а P[n–2] — минимальный
❸ Оболочка строится по часовой стрелке начиная с минимального полярного угла и low.
❹ Каждый поворот влево указывает, что последняя точка оболочки должна быть удалена
❺ Поскольку это должна быть P[n–2]

Реализация алгоритма

Если вы решаете эту задачу вручную, вероятно, у вас не возникает проблемы с отслеживанием соответствующих краев, но может оказаться трудно объяснить точную последовательность выполненных вами шагов. Ключевым шагом в этом алгоритме является сортировка точек по убыванию полярного угла относительно самой нижней точки множества. После упорядочения алгоритм переходит к обходу этих точек, расширяя частично построенную оболочку и изменяя ее структуру, если последние три точки оболочки делают левый поворот, что указывает на невыпуклую форму оболочки (пример 3.1).

Пример 3.1. Реализация сканирования Грэхема

```
public class NativeGrahamScan implements IConvexHull {
    public IPoint[] compute(IPoint[] pts) {
        int n = pts.length;
        if (n < 3) {
            return pts;
        }

        // Находим наинизшую точку и меняем ее с последней
        // точкой массива points[], если она не является таковой
        int lowest = 0;
        double lowestY = pts[0].getY();

        for (int i = 1; i < n; i++) {
            if (pts[i].getY() < lowestY) {
                lowestY = pts[i].getY();
                lowest = i;
            }
        }

        if (lowest != n - 1) {
            IPoint temp = pts[n - 1];
            pts[n - 1] = pts[lowest];
            pts[lowest] = temp;
        }
```

```
// Сортируем points[0..n-2] в порядке убывания полярного
// угла по отношению к наинизшей точке points[n-1].
new HeapSort<IPoint>().sort(pts,0,n-2,
                new ReversePolarSorter(pts[n - 1]));

// *Известно*, что три точки (в указанном порядке)
// находятся на оболочке - (points[n-2]), наинизшая точка
// (points[n-1]) и точка с наибольшим полярным углом
// (points[0]). Начинаем с первых двух
DoubleLinkedList<IPoint>list=new DoubleLinkedList<IPoint>();
list.insert(pts[n - 2]);
list.insert(pts[n - 1]);

// Если все точки коллинеарны, обрабатываем их, чтобы
// избежать проблем позже
double firstAngle=Math.atan2(pts[0].getY()-lowest,
                             pts[0].getX()-pts[n-1].getX());
double lastAngle=Math.atan2(pts[n-2].getY()-lowest,
                            pts[n-2].getX()-pts[n-1].getX());
if (firstAngle == lastAngle) {
    return new IPoint[] { pts[n - 1], pts[0] };
}

// Последовательно посещаем каждую точку, удаляя точки,
// приводящие к ошибке. Поскольку у нас всегда есть как
// минимум один "правый поворот", внутренний цикл всегда
// завершается
for (int i = 0; i < n - 1; i++) {
    while (isLeftTurn(list.last().prev().value(),
                      list.last().value(),
                      pts[i])) {
        list.removeLast();
    }

    // Вставляем следующую точку оболочки в ее
    // корректную позицию
    list.insert(pts[i]);
}

// Последняя точка дублируется, так что мы берем
// n-1 точек начиная с наинизшей
IPoint hull[] = new IPoint[list.size() - 1];
DoubleNode<IPoint> ptr = list.first().next();
int idx = 0;

while (idx < hull.length) {
    hull[idx++] = ptr.value();
    ptr = ptr.next();
}
```

```
        return hull;
    }
    /** Проверка коллинеарности для определения левого поворота. */
    public static boolean isLeftTurn(IPoint p1,IPoint p2,IPoint p3){
        return (p2.getX()-p1.getX())*(p3.getY()-p1.getY())-
               (p2.getY()-p1.getY())*(p3.getX()-p1.getX())>0;
    }
}

/** Клас сортировки в обратном порядке по полярному углу
    относительно наинизшей точки */
class ReversePolarSorter implements Comparator<IPoint> {
    /** Сохраненные координаты x,y базовой точки для сравнения. */
    final double baseX;
    final double baseY;
    /** PolarSorter вычисляет все точки, сравниваемые с базовой. */
    public ReversePolarSorter(IPoint base) {
        this.baseX = base.getX();
        this.baseY = base.getY();
    }
    public int compare(IPoint one, IPoint two) {
        if (one == two) {
            return 0;
        }

        // Оба угла вычисляются с помощью функции atan2.
        // Код работает, так как one.y всегда не меньше base.y
        double oneY = one.getY();
        double twoY = two.getY();
        double oneAngle = Math.atan2(oneY-baseY, one.getX()-baseX);
        double twoAngle = Math.atan2(twoY-baseY, two.getX()-baseX);

        if (oneAngle > twoAngle) {
            return -1;
        } else if (oneAngle < twoAngle) {
            return +1;
        }

        // Если углы одинаковы, для корректной работы алгоритма
        // необходимо упорядочение в убывающем порядке по высоте
        if (oneY > twoY) {
            return -1;
        } else if (oneY < twoY) {
            return +1;
        }

        return 0;
    }
}
```

Если все $n > 2$ точек коллинеарны, то в данном частном случае оболочка состоит из двух крайних точек множества. Вычисленная выпуклая оболочка может содержать несколько последовательных точек, являющихся коллинеарными, поскольку мы не предпринимаем попыток их удаления.

Анализ алгоритма

Сортировка n точек требует $O(n \cdot \log n)$ точек, как известно из главы 4, "Алгоритмы сортировки". Остальная часть алгоритма состоит из цикла `for`, который выполняется n раз; но сколько раз выполняется его внутренний цикл `while`? До тех пор, пока имеется левый поворот, точка удаляется из оболочки; и так до тех пор, пока не останутся только первые три точки. Поскольку в оболочку включаются не более n точек, внутренний цикл `while` может выполняться в общей сложности не более чем n раз. Таким образом, время работы цикла `for` — $O(n)$. В результате общая производительность алгоритма равна $O(n \cdot \log n)$, поскольку стоимость сортировки доминирует в стоимости всех вычислений.

Распространенные подходы

В этом разделе представлены основные алгоритмические подходы, используемые в данной книге. Вы должны понимать эти общие стратегии решения задач, чтобы видеть, каким образом они могут быть применены для решения конкретных задач. В главе 10, "Пространственные древовидные структуры", описываются некоторые дополнительные стратегии, такие как поиск приемлемого приблизительного решения или использование рандомизации с большим количеством испытаний для сходимости к правильному результату, не прибегая к исчерпывающему поиску.

Жадная стратегия

Жадная стратегия решает задачу размера n пошагово. На каждом шаге жадный алгоритм находит наилучшее локальное решение, которое можно получить с учетом имеющейся информации; обычно размер задачи при этом уменьшается на единицу. После того, как будут завершены все n шагов, алгоритм возвращает вычисленное решение.

Например, для сортировки массива A из n чисел жадная **сортировка выбором** находит наибольшее значение из $A[0, n-1]$ и обменивает его с элементом в позиции $A[n-1]$, что гарантирует, что $A[n-1]$ находится в правильном месте. Затем процесс повторяется и выполняется поиск наибольшего значения среди оставшихся элементов $A[0, n-2]$, которое затем аналогичным образом меняется местами с элементом в позиции $A[n-2]$. Этот процесс продолжается до тех пор, пока не будет отсортирован весь массив. Более подробно этот процесс описан в главе 4, "Алгоритмы сортировки".

Жадная стратегия обычно характеризуется медленным уменьшением размера решаемых подзадач по мере обработки алгоритмом входных данных. Если подзадача может быть решена за время $O(\log n)$, то жадная стратегия будет иметь производительность $O(n \cdot \log n)$. Если подзадача решается за время $O(n)$, как в **сортировке выбором**, то общей производительностью алгоритма будет $O(n^2)$.

Разделяй и властвуй

Стратегия "Разделяй и властвуй" решает задачу размера n, разделяя ее на две независимые подзадачи, каждая около половины размера исходной задачи[1]. Довольно часто решение является рекурсивным, прерывающимся базовым случаем, который может быть решен тривиально. Должно также иметься некоторое вычисление *разрешения*, которое позволяет получить решение для задачи из решений меньших подзадач.

Например, чтобы найти наибольший элемент массива из n чисел, рекурсивная функция из примера 3.2 конструирует две подзадачи. Естественно, максимальный элемент исходной задачи является просто наибольшим значением из двух максимальных элементов, найденных в подзадачах. Обратите внимание, как рекурсия прекращается, когда размер подзадачи достигает значения 1; в этом случае возвращается единственный имеющийся элемент `vals[left]`.

Пример 3.2. Применение подхода "разделяй и властвуй" для поиска максимального значения массива

```
/** Применение рекурсии. */
public static int maxElement(int[] vals) {
    if (vals.length == 0) {
        throw new NoSuchElementException("Массив пуст.");
    }
    return maxElement(vals, 0, vals.length);
}

/** Вычисляем максимальный элемент в подзадаче vals[left, right).
 *  Правая конечная точка не является частью диапазона! */
static int maxElement (int[] vals, int left, int right) {
    if (right - left == 1) {
        return vals[left];
    }

    // Решение подзадач
    int mid = (left + right)/2;
    int max1 = maxElement(vals, left, mid);
    int max2 = maxElement(vals, mid, right);
```

[1] Эти условия не являются обязательными. Задача в общем случае может быть разбита и на большее количество подзадач, и подзадачи могут иметь разный размер. — *Примеч. ред.*

```
    // Разрешение: получение результата из решений подзадач
    if (max1 > max2) { return max1; }
    return max2;
}
```

Алгоритм "разделяй и властвуй", структурированный так, как показано в примере 3.2, демонстрирует производительность $O(n)$, если шаг разрешения может быть выполнен за постоянное время $O(1)$ (что и происходит). Если бы шагу разрешения самому требовалось время $O(n)$, то общая производительность алгоритма была бы $O(n \cdot log\ n)$. Обратите внимание, что найти наибольший элемент в массиве можно быстрее — путем сканирования каждого элемента и хранения текущего наибольшего значения. Пусть это будет кратким напоминанием, что подход "разделяй и властвуй" не всегда обеспечивает самое быстрое решение.

Динамическое программирование

Динамическое программирование является вариацией подхода "разделяй и властвуй", которая решает задачу путем ее разделения на несколько более простых подзадач, которые решаются в определенном порядке. Она решает каждую из более мелких задач только один раз и сохраняет результаты для последующего использования, чтобы избежать ненужного пересчета. Затем этот метод решает задачи большего размера и *объединяет* в единое решение результаты решения меньших подзадач. Во многих случаях найденное решение является для поставленной задачи доказуемо оптимальным.

Динамическое программирование часто используется для задач оптимизации, в которых цель заключается в минимизации или максимизации некоторого вычисления. Наилучший способ объяснения динамического программирования — на конкретном примере.

Ученые часто сравнивают последовательности ДНК для определения их сходства. Если вы представите такую последовательность ДНК как строку символов A, C, T и G, то задачу можно переформулировать как вычисление *минимального расстояния редактирования* между двумя строками. Иначе говоря, для заданной базовой строки s_1 и целевой строки s_2 требуется определить наименьшее число операций редактирования, которые преобразуют s_1 в s_2, если вы можете:

- заменить символ s_1 другим символом;
- удалить символ из s_1;
- вставить символ в s_1.

Например, для строки s_1, представляющей последовательность ДНК “GCTAC”, нужны только три операции редактирования данной базовой строки, чтобы преобразовать ее в целевую строку s_2, значение которой — “CTCA”:

- заменить четвертый символ (“A”) символом “C”;
- удалить первый символ (“G”);
- заменить последний символ “C” символом “A”.

Это не единственная последовательность операций, но вам нужно как минимум три операции редактирования для преобразования s_1 в s_2. Для начала цель заключается в том, чтобы вычислить значение оптимального ответа, т.е. количество операций редактирования, а не фактическую последовательность операций.

Динамическое программирование работает путем сохранения результатов более простых подзадач; в данном примере можно использовать двумерную матрицу $m[i][j]$, чтобы записывать результат вычисления минимального расстояния редактирования между первыми i символами s_1 и первыми j символами s_2. Начнем с создания следующей исходной матрицы:

```
0 1 2 3 4
1 . . . .
2 . . . .
3 . . . .
4 . . . .
5 . . . .
```

В этой таблице каждая строка индексируется с помощью i, а каждый столбец индексируется с помощью j. По завершении работы запись $m[0][4]$ (верхний правый угол таблицы) будет содержать результат редактирования расстояния между первыми 0 символами s_1 (т.е. пустой строкой “”) и первыми четырьмя символами s_2 (т.е. всей строкой “CTCA”). Значение $m[0][4]$ равно 4, потому что вы должны вставить четыре символа в пустую строку, чтобы она стала строкой s_2. Аналогично значение $m[3][0]$ равно 3, потому что, имея первые три символа s_1 (т.е. “GCT”), получить из них первые нуль символов s_2 (т.е. пустую строку “”) можно путем удаления всех трех символов.

Хитрость динамического программирования заключается в цикле оптимизации, который показывает, как соединить результаты решения этих подзадач в решение больших задач. Рассмотрим значение $m[1][1]$, которое представляет собой расстояние редактирования между первым символом s_1 (“G”) и первым символом s_2 (“C”). Есть три варианта:

- заменить символ “G” символом “C” с ценой 1;
- удалить символ “G” и вставить “C” с ценой 2;
- вставить символ “C” и удалить “G” с ценой 2.

Понятно, что нам нужна минимальная стоимость, так что $m[1][1] = 1$. Как же можно обобщить это решение? Рассмотрим вычисление, показанное на рис. 3.2.

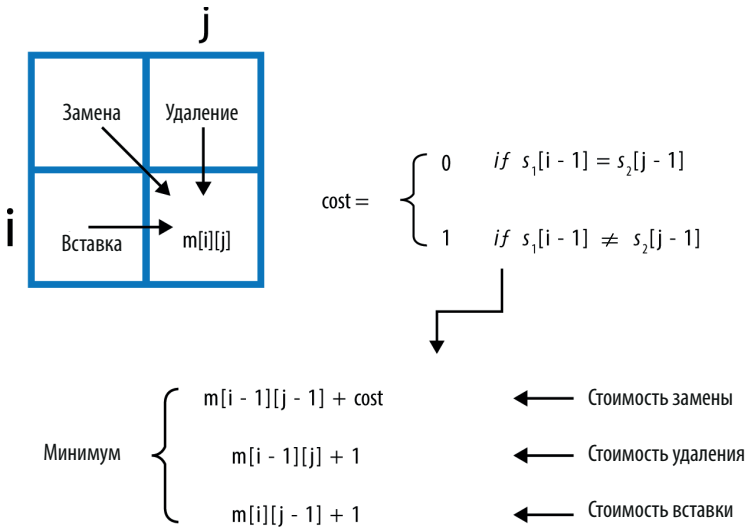

Рис. 3.2. Вычисление m[i][j]

Три варианта вычисления $m[i][j]$ таковы.

Стоимость замены

Вычисляем расстояние редактирования между первыми $i-1$ символами s_1 и первыми $j-1$ символами s_2 и добавляем 1 для замены j-го символа s_2 i-м символом s_1, если они различны.

Стоимость удаления

Вычисляем расстояние редактирования между первыми $i-1$ символами s_1 и первыми j символами s_2 и добавляем 1 для удаления i-го символа s_1.

Стоимость вставки

Вычисляем расстояние редактирования между первыми i символами s_1 и первыми $j-1$ символами s_2 и добавляем 1 для вставки j-го символа s_2.

Визуализируя вычисления, вы увидите, что динамическое программирование должно решать подзадачи в правильном порядке (от верхней строки к нижней строке и слева направо в каждой строке, как показано в примере 3.3). Вычисление проходит по строке от значения индекса $i=1$ до $len(s_1)$. После того как матрица m будет заполнена начальными значениями, вложенный цикл `for` вычисляет минимальное значение для каждой из подзадач в указанном порядке, пока не будут вычислены все значения в m. Этот процесс не является рекурсивным, вместо этого он использует

результаты предыдущих вычислений для задач меньшего размера. Решение полной задачи находится в $m[len(s_1)][len(s_2)]$.

Пример 3.3. Поиск минимального расстояния редактирования с использованием динамического программирования

```
def minEditDistance(s1, s2):
    """ Вычисление минимального расстояния редактирования s1->s2."""
    len1 = len(s1)
    len2 = len(s2)

    # Создание двумерной структуры, такой, что m[i][j] = 0
    # for i in 0 .. len1 и for j in 0 .. len2
    m = [None] * (len1 + 1)
    for i in range(len1+1):
        m[i] = [0] * (len2+1)

    # Настройка начальных значений по горизонтали и вертикали
    for i in range(1, len1+1):
        m[i][0] = i
    for j in range(1, len2+1):
        m[0][j] = j

    # Вычисление наилучших значений
    for i in range(1,len1+1):
        for j in range(1,len2+1):
            cost = 1
            if s1[i-1] == s2[j-1]: cost = 0
            replaceCost = m[i-1][j-1] + cost
            removeCost = m[i-1][j] + 1
            insertCost = m[i][j-1] + 1
            m[i][j] = min(replaceCost,removeCost,insertCost)
    return m[len1][len2]
```

В табл. 3.5 показаны окончательные значения элементов *m*.

Таблица 3.5. Результаты решения всех подзадач

0	1	2	3	4
1	1	2	3	4
2	1	2	2	3
3	2	1	2	3
4	3	2	2	2
5	4	3	2	3

Стоимость подзадачи $m[3][2] = 1$ представляет собой расстояние редактирования между строками “GCT” и “CT”. Как видите, вам нужно только удалить первый символ, что подтверждает корректность найденной стоимости. Однако этот код

показывает только, как вычислить расстояние минимального редактирования; чтобы записать последовательность операций, которые при этом требуется выполнить, нужно записывать в матрицу *prev*[*i*][*j*], какой именно из трех случаев был выбран при вычислении минимального значения *m*[*i*][*j*]. Чтобы восстановить эти операции, надо просто выполнить обратный проход от $m[len(s_1)][len(s_2)]$ с использованием решений, записанных в *prev*[*i*][*j*], и остановиться по достижении *m*[0][0]. Это модифицированное решение приведено в примере 3.4.

Пример 3.4. Поиск минимального расстояния редактирования и операций с использованием динамического программирования

```
REPLACE = 0
REMOVE = 1
INSERT = 2
def minEditDistance(s1, s2):
    """ Вычисление минимального расстояния
        редактирования и операций s1->s2."""
    len1 = len(s1)
    len2 = len(s2)

    # Создание двумерной структуры, такой, что m[i][j] = 0
    # for i in 0 .. len1 and for j in 0 .. len2
    m = [None] * (len1 + 1)
    op = [None] * (len1 + 1)
    for i in range(len1+1):
        m[i] = [0] * (len2+1)
        op[i] = [-1] * (len2+1)

    # Настройка начальных значений по горизонтали и вертикали
    for j in range(1, len2+1):
        m[0][j] = j
    for i in range(1, len1+1):
        m[i][0] = i

    # Вычисление наилучших значений
    for i in range(1,len1+1):
        for j in range(1,len2+1):
            cost = 1

            if s1[i-1] == s2[j-1]: cost = 0
            replaceCost = m[i-1][j-1] + cost
            removeCost = m[i-1][j] + 1
            insertCost = m[i][j-1] + 1
            costs = [replaceCost,removeCost,insertCost]
            m[i][j] = min(costs)
            op[i][j] = costs.index(m[i][j])
```

```
ops = []
i = len1
j = len2
while i != 0 or j != 0:
    if op[i][j] == REMOVE or j == 0:
        ops.append('remove {} char {} of {}'.format(i,s1[i-1],s1))
        i = i-1
    elif op[i][j] == INSERT or i == 0:
        ops.append('insert {} char {} of {}'.format(j,s2[j-1],s2))
        j = j-1
    else:
        if m[i-1][j-1] < m[i][j]:
            fmt='replace {} char of {} ({}) with {}'
            ops.append(fmt.format(i,s1,s1[i-1],s2[j-1]))
        i,j = i-1,j-1

return m[len1][len2], ops
```