**Лабораторная работа №3**

**Теоретическая часть**

1. **разница курсов акций**

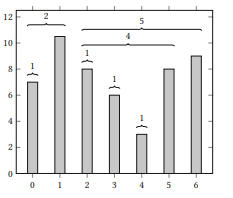
Представьте, что курс акций меняется каждый день. Таким образом, у вас есть ряды чисел, каждое из которых представляет собой финальную стоимость конкретной акции в конкретный день. Дни идут в хронологическом порядке. Дни, когда фондовый рынок закрыт, не считаются.

Разница стоимостей акций в конкретный день — это число следующих друг за другом дней, от выбранного нами и в обратном направлении, до того дня, в который стоимость была меньше или равна стоимости в выбранный нами день. Таким образом, задача разницы курсов акций представляет следующее условие: даны ряды ежедневных курсов акции, и нам требуется найти разницу стоимостей акций для каждого дня в ряду. Для примера обратимся к рисунку 1.1. Первый день считается за 0. На шестой день нашего примера разница составляет пять дней, на пятый день — четыре дня, а на четвертый день — один день.

В реальной жизни в рядах могут находиться тысячи дней, и нам нужно будет просчитать разницу для множества различных рядов, в каждом из которых описаны изменения для разных курсов акций. Мы хотим поручить эту задачу компьютеру.

Многие задачи, решение которых мы поручаем компьютеру, имеют, как правило, не один способ решения, а несколько. Какие-то решения лучше, какие-то хуже. Впрочем, в данном случае «лучше» — весьма условный термин, который по сути ничего не значит, ведь, когда мы говорим «лучше», мы подразумеваем какой-то контекст: лучше в чем-то конкретном. Речь может быть о скорости, памяти или чем-то еще, что связано с такими ресурсами, как время и пространство. Мы еще коснемся данной темы, однако сейчас главное усвоить сказанное выше, потому что решение задачи может оказаться простым и в то же время совсем не оптимальным из-за поставленных условий или критериев.

Допустим, из ряда дней вы взяли день m. Можно найти разницу курсов акций в день m, вернувшись на день назад и оказавшись в дне m – 1. Если стоимость в день m – 1 выше, чем стоимость в день m, тогда вы знаете, что разница курсов акций в день m равна 1. Но если стоимость в день m – 1 ниже или равна стоимости в день m, то разница курсов акций в день m равна 2, а может быть даже больше: в зависимости от стоимости в предыдущий день. Поэтому нам нужно посмотреть цену акций в день m – 2. Если стоимость не выше, чем в день m, то нам надо проверить более ранний день и так далее. Следовательно, можно получить два итога. Итог первый: у нас закончатся дни (то есть мы вернемся в самое начало ряда). В этом случае получается, что на протяжении всех дней до дня m курс акций был ниже или точно таким же, как в день m, и значит, разница курсов равна m. Итог второй: мы наткнемся на день k (где k < m), когда цены на акции были выше, чем в день m, и получим, что разница равна m – k.

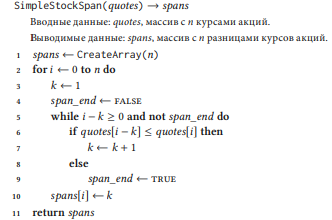
****

**Рисунок 1.1.** Пример разницы стоимостей акций

Если ряд состоит из n дней, то, чтобы найти разницу для каждого дня, мы повторяем описанную выше процедуру n раз. Применив данный метод к ряду, изображенному на рисунке 1.1, можно убедиться, что он работает корректно. Однако такой подход никуда не годится. Проза — прекрасное средство коммуникации и описания реалий нашего мира, но она совершенно не подходит для описания компьютерных процедур, потому что нам надо четко и ясно описать машине поставленную задачу.

Если мы достаточно точны в своих выражениях, то компьютер сможет понять наши команды, после чего мы сможем написать программу. Однако описание команд в компьютерных программах может оказаться совершенно непонятным человеку, потому что машине требуются всевозможные уточнения и детали, нужные ей для работы, но не связанные с решением поставленной задачи напрямую. То, что для компьютера является минимально подробным описанием, для человека может оказаться сверхподробным, и он его едва ли поймет.

**Алгоритм 1.1**. Простой алгоритм для задачи о разнице курсов акций



Поэтому нам стоит выбрать золотую середину и описать команды с помощью структурированного языка, который более точен, чем обычный текст, с пониманием которого у людей не возникнет проблем. Скорее всего, компьютер не поймет заданный структурированный язык как таковой, но если на этом языке написать компьютерную программу, то он без проблем поймет, что мы от него хотим.

**1.1 Алгоритм**

Прежде чем мы приступим к поиску решения нашей задачи с разницей курсов акций, хорошо бы познакомиться с основными терминами. Алгоритм — это последовательность операций, но не простая, а особая. Ее можно описать как конечную последовательность множества этапов, которая должна завершиться через определенный промежуток времени. Каждое действие, каждый этап должны быть четко прописаны так, чтобы человек мог самостоятельно разобрать его на бумаге и понять. Алгоритм выполняет действия, основанные на полученных от нас данных, и выдает результат, показывающий выполненную работу.

Алгоритм 1.1 реализует описанную ранее процедуру. Алгоритм 1.1 демонстрирует, каким образом мы будем описывать алгоритмы. Вместо языка программирования, который заставляет нас заниматься подробностями реализации, не относящимися к логике алгоритма, мы используем псевдокод. Это нечто между настоящим программным кодом и неформальным описанием. Он использует структурированный формат и применяет множество слов, которые наделяет особым значением. Так или иначе, псевдокод нельзя считать настоящим программным кодом. Он предназначен не для того, чтобы его понимали машины, а для того, чтобы его понимали люди. К слову, программы вообще должны быть понятны людям, но на деле все далеко не так просто: существует множество рабочих компьютерных программ, написанных совершенно невнятно.

У каждого алгоритма есть название, они берут вводные данные и выдают некий результат. Мы будем записывать название алгоритма в «ВерблюжьемРегистре» (CamelCase), а вводные данные — в круглых скобках. Затем мы отметим полученные данные с помощью →. В последующих строчках мы будем описывать вводимые и выводимые данные. Название алгоритма, за которым в скобках следуют вводные данные, может применяться для вызова алгоритма. Как только алгоритм написан, мы можем использовать его в качестве черного ящика, который мы заполняем вводными данными: черный ящик потом выдаст выводимые данные, результат заданного алгоритма. Реализованный в языке программирования алгоритм представляет собой именованный фрагмент программного кода — функцию. В компьютерной программе мы вызываем функцию, с помощью которой работает алгоритм.

Некоторые алгоритмы выдают результат неявным образом, так как их действия направлены на часть контекста. Например, мы можем выделить алгоритму место, куда будет вписан итог работы. В этом случае алгоритм не вернет привычных выводимых данных, но изменит контекст. Некоторые языки программирования отличают фрагменты именованного программного кода, возвращающие видимый результат, называя их функциями, от фрагментов именованного программного кода, которые не возвращают выводимые данные в явной форме, но имеют иные побочные действия — их называют процедурами. Различие пришло из математики, где функция всегда выдает какое-то значение. Для нас же алгоритм, реализованный в компьютерной программе, может оказаться как функцией, так и процедурой.

В нашем псевдокоде будут встречаться ключевые слова, выделенные **жирным шрифтом**, которые не требуют разъяснений, если вы хоть немного знакомы с компьютерами и языками программирования. Мы используем символ ← для присвоения и символ = для равенства, а также заимствуем пять символов математических операций (+, –, /, х, ·); у нас два знака умножения, и нам понадобятся оба, в зависимости от эстетических предпочтений. Мы не будем использовать ключевые слова или символы для разграничения блоков псевдокода: мы прибегнем к красной строке.

В данном алгоритме мы применяем массив. Это структура, содержащая данные, которые позволяют нам использовать данные особыми способами. Структура, содержащая данные и позволяющая производить особые операции над ними, называется структурой данных. Таким образом, массив является структурой данных.

Массивы для компьютера — то же самое, что ряды объектов для людей. Они состоят из последовательности элементов, которые хранятся в памяти компьютера. Чтобы получить требуемое для элементов место и создать массив, содержащий n элементов, в первой строчке алгоритма 1.1 мы вызываем алгоритм создания массива, CreateArray. Если вы знакомы с массивами, то вам может показаться странным, что создание массива нуждается в алгоритме. Но все так и есть. Чтобы получить блок памяти для данных, вам нужно найти свободное место в памяти компьютера и отметить его как используемое массивом. Все это делается с помощью вызова CreateArray(n). В результате мы получаем массив с местом для n элементов: сначала там нет никаких элементов, а только место для них. Заполнить свободное пространство нужными данными — как раз задача алгоритма, который вызывает CreateArray(n).

Для массива А с помощью A[i] мы помечаем элемент i и получаем к нему доступ. Позиция элемента в массиве, как у i в A[i], называется индексом. В массиве из n элементов содержатся элементы A[0], A[1],…, A[n – 1]. Сперва это может показаться странным, так как первый элемент — нулевой, а последний (n – 1). Наверняка вы ожидали, что на их месте будут первый и n-й. Однако именно так массивы и работают в большинстве программных языков, к чему вам следует поскорее привыкнуть. Как обычно, когда мы просматриваем массив размером n, мы выполняем перебор от 0 до n – 1. В наших алгоритмах, когда мы говорим, что нечто будет принимать значения от числа x до числа y (полагая, что x меньше y), мы имеем в виду все значения x и выше, но не включая y; посмотрите вторую строчку нашего алгоритма

Мы полагаем, что на доступ к элементу i всегда требуется одно и то же время, чем бы ни являлось i, поэтому доступ к A[0] потребует столько же времени, сколько и к A[n – 1]. В этом заключается характерная особенность массивов: до каждого элемента можно добраться за одно и то же время; массиву не нужно искать какой-то элемент, когда мы получаем к нему доступ по индексу.

Что касается условных знаков, то при описании алгоритмов мы используем для значения переменных, которые фигурируют в них, строчные символы; однако когда переменные будут связаны со структурой данных, мы будем применять символы верхнего регистра, как в массиве A, чтобы выделить их; но это не всегда необходимо. Когда нужно будет дать переменной имя из нескольких слов, мы используем подчеркивание (\_), как в a\_connector, что необходимо, поскольку компьютеры не понимают, что слова, разделенные пробелами, могут составлять единую фразу.

Алгоритм 1.1 использует массивы, в которых находятся числа. Массивы могут содержать любой тип элементов, однако в нашем псевдокоде каждый массив может заключать в себе только один тип. То же самое относится к большинству языков программирования. Например, у вас может быть массив с десятичными числами, массив с дробями, массив элементов, обозначающих людей, и другой массив с элементами, обозначающими адреса. У вас не может быть массива, в котором одновременно и десятичные числа, и элементы, обозначающие людей. Чем могут являться «элементы, обозначающие людей» зависит от особенностей программного языка, использованного в программе. Все языки программирования предлагают способы представления важных вещей.

Особо полезный вид массива — тот, который, содержит символы. Массив символов представляет собой строку символов, то есть последовательность букв, чисел, слов, предложений и чего угодно. Во всех массивах каждый отдельный символ может быть найден по индивидуальному индексу. Если у нас есть строка s = “Hello, World”, то s[0] — это буква “H”, а s[11] — буква “d”.

Подытожим сказанное. Массив — это структура данных, в которой содержится последовательность элементов одного и того же типа. Есть две операции, относящиеся к массивам.

• CreateArray(n) создает массив, который может вместить n-е количество элементов. Массив не запущен, в нем пока нет ни одного элемента, но нужное им место создано и готово к использованию.

• Как мы видели, имея массив A и его i элемента, A[i] получает доступ к выбранному элементу, и получение доступа к любому элементу массива занимает одинаковое время. Попытка получения доступа к A[i] при условии i < 0 приведет к ошибке.

Вернемся к алгоритму 1.1. Следуя сказанному выше, в строчках 2-10 нашего алгоритма содержится цикл, то есть блок кода, который повторяется. Цикл выполняется n раз; единожды для подсчета разницы, если у нас есть цена акций за n дней. Нынешний день, разницу в который мы собираемся вычислить, задан переменной i. Изначально мы находимся в дне 0, самой ранней точке временной линии; каждый раз, проходя через вторую строчку нашего цикла, мы будем перемещаться в день 1, 2, …, n – 1.

Мы используем переменную k для обозначения длины промежутка, отмеряющего разницу; переменная — это название, присвоенное некоему фрагменту данных в нашем псевдокоде. Содержание этих данных, а точнее значение переменной может меняться в процессе выполнения алгоритма, и, следовательно, меняется ее название. Значение k на момент начала расчета разницы всегда равно 1, что и прописано в строчке 3. Также мы используем индикаторную переменную, span\_end. Индикаторные переменные принимают значения истинно (true) и ложно (false) и указывают, выполняется ли что-то или не выполняется. Переменная span\_end примет значение истинно, когда мы достигнем конца промежутка, отмеряющего разницу.

В начале расчета промежутка переменная span\_end будет ложной, как в строчке 4. Длина промежутка просчитана во внутреннем цикле строчек 5-9. Строчка 5 говорит вернуться в прошлое насколько возможно далеко и на расстояние, пока промежуток не закончится. Настолько далеко, насколько позволяет условие i – k ≥ 0, где i – k является индексом дня, в который нам нужно вернуться, чтобы проверить, закончился ли промежуток, обозначающий разницу; к тому же индекс не может равняться нулю, так как он отображает первый день. Проверка конца промежутка отображается в строчке 6. Если промежуток не закончился, то мы увеличиваем его в строчке 7. В противном случае в строчке 9 мы констатируем факт, что промежуток закончился, и, таким образом, цикл остановится, когда процесс вернется к строчке 5. В конце каждой итерации внешнего цикла строчек 2-10 мы помещаем значение k в отведенное в массиве span место (строчка 10). Мы возвращаем промежутки (spans), в которых находятся результаты алгоритма, в строке 11, после выхода из цикла.

Заметьте, что в начале i = 0 и k = 1. Это значит, что условие в строчке 5 совершенно точно не будет выполнено на раннем моменте времени. Так и должно быть, потому что разница может быть равна только 1

Тут самое время вспомнить, что мы только что говорили об алгоритмах и проверке на бумаге. Верный способ понять работу алгоритма — сделать ее самому, своими руками. Если в какой-то момент алгоритм кажется сложным или же вы не уверены, что уловили его суть, тогда возьмите ручку, лист бумаги и распишите его действия в нескольких примерах. Хоть способ и кажется устаревшим, тем не менее он сэкономит вам уйму времени. Если вы что-то не поняли в алгоритме 1.1, разберитесь с ним и возвращайтесь к прочтению книги, лишь когда полностью освоите его.

**1.2. Время выполнения работы и сложность**

В алгоритме 1.1 мы видим решение задачи о разнице курсов акций, однако мы можем найти способ получше. Лучше в данном случае значит — быстрее. Когда мы говорим о скорости в контексте алгоритмов, мы подразумеваем число этапов, которое требуется для его выполнения. Скорость компьютеров тут не важна; хотя они и будут выполнять расчетные этапы все быстрее и быстрее, количество самих этапов останется неизменным, поэтому имеет смысл задуматься над оценкой сложности выполнения алгоритма в этапах, нужных для выполнения работы. Мы называем количество требуемых этапов временем выполнения работы алгоритма, хотя имеем дело с безразмерным числом, не обусловленным единицами времени. Использование единиц времени сделало бы подсчет времени выполнения чем-то, относящимся к специфичной расчетной модели, которая для нас бесполезна.

Подумайте, как много времени займет расчет разницы n курсов акций. Наш алгоритм включает в себя цикл, который начинается в строчке 2 и повторяется n раз: для каждой цены. Тут есть и внутренний цикл, который начинается в строчке 5 и при каждом повторе внешнего цикла будет пытаться найти промежуток, обозначающий разницу стоимостей. Каждую стоимость он сравнит с ценами всех предыдущих дней. При самом неблагоприятном варианте, если указанная цена окажется самой высокой цикл проверит все предыдущие цены. Если стоимость k самая высокая из всех предыдущих, тогда внутренний цикл повторится k раз. Тем не менее в худшем случае, когда все цены идут в порядке возрастания, строчка 7 выполнится следующее число раз:



Если равенство неочевидно, тогда можно добавить числа 1, 2, …, n дважды и увидеть, что все так и есть:

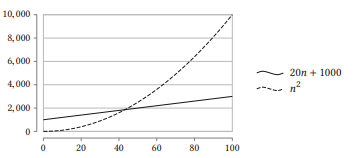


Так как строка 6 — это этап, который будет повторяться наибольшее число раз, n(n + 1)/2 является худшим вариантом для времени выполнения работы алгоритма.

Когда мы говорим о времени выполнения работы алгоритма, нас интересует время выполнения при наличии большого объема вводных данных (в нашем случае число n). Такое время работы называется асимптотическим, потому что оно работает с поведением алгоритма, когда вводные данные постоянно и неограниченно увеличиваются. Для такого случая есть особая запись, нотация. Если для всех значений n бóльших, чем некие изначальные положительные значения, функция f(n) меньше или равна другой функции g(n), имеющий шкалу неких положительных постоянных значений c, которые представляют собой cg(n), то для каждой функции f(n) мы говорим, что O(f(n)) = g(n). Если точнее, мы говорим, что O(f(n)) = g(n), если есть положительные постоянные c и n0 , такие как 0 ≤ f (n) ≤ cg(n) для всех n ≥ n0

Нотация O(f(n)) называется «нотацией большого О». Помните, что нас интересуют большие значения наших вводных данных, потому что именно здесь мы сэкономим больше всего. Взгляните на рисунок 1.2, где изображены две функции: f 1 (n) = 20n + 1000 и f 2 (n) = n2 . Для небольших значений n это f 1 (n), которая принимает наибольшие значения, но ситуация меняется очень стремительно, после чего n2 растет значительно быстрее.

Нотация большого О позволяет нам упрощать функции. Если у нас есть функция f(n) = 3n3 + 5n2 + 2n + 1000, тогда мы упрощаем ее до O(f(n)) = n2 . Почему?



**Рисунок 1.2.** Сравнение O(f(n)

Потому что мы всегда можем подобрать значение c, так что 0 ≤ f(n) ≤ cn3 . Обычно, когда у нас есть функция со множеством членов, наибольший член сильно влияет на рост функции, и мы убираем наименьшие члены при работе с большими О. Таким образом

Описанное нами время выполнения работы алгоритма называется вычислительной сложностью или, коротко, сложностью. Когда мы прибегаем к упрощенным формам функций в изучении времени выполнения работы алгоритмов, оказывается, что время выполнения работы большинства алгоритмов содержит функции, окруженные небольшим количеством упрощенных функций. Это значит, что сложность алгоритмов часто попадает в одну из немногочисленных категорий, или разновидностей.

Прежде всего у нас есть постоянная функция f(n) = c. Это всего лишь значит, что данная функция всегда имеет одно и то же значение, c, вне зависимости от того, чем является n. До тех пор, пока c не примет непомерно высокое значение, у нас есть наилучший вариант, который можно желать для нашего алгоритма. Если говорить о нотации большого О, то мы знаем по определению, что существуют положительные постоянные c и n0 , такие как 0 ≤ f(n) ≤ cg(n) = c · 1. Действительно, c является константным значением функции, а n0 = 1. Следственно, O(c) = O(1). Алгоритм, который ведет себя подобным образом, мы называем алгоритмом постоянного времени. На самом деле этот термин употреблен неверно, потому что тут вовсе не имеется в виду, что алгоритм будет всегда требовать неизменного количества времени, независимого от вводных данных. Имеется в виду, что от вводных данных не зависит верхняя граница времени работы алгоритма. Например, простому алгоритму, который добавляет значение y к значению x, если x > 0, не всегда требуется одинаковое количество времени для выполнения задачи: если x > 0, то он прибавляет нужное значение, а в противном случае ничего не делает. Тем не менее верхняя граница, то есть требуемое для прибавления время, постоянна и попадает в категорию О(1). К сожалению, существует не так уж и много алгоритмов, которые работают за постоянное время. Самые привычные операции, которые исполняются за постоянное время, — это получение доступа к элементу в массиве, который мы считаем постоянным и независимым от индекса выбранного нами элемента; как мы уже видели, в массиве А из n элементов доступ к А[0] занимает столько же времени, сколько и к А[n – 1].

После алгоритмов постоянного времени мы обратимся к алгоритмам, которые выполняются за логарифмическое время. Логарифмическая функция или логарифм — это loga (n), которое обозначает степень, в которую мы должны возвести a, чтобы получить n: если = loga (n) тогда n = ay . Число a — основа логарифма. Из определения логарифма следует, что x = aloga g x , которое показывает, что логарифмирование — это обратное действие от возведения числа в степень. Действительно, log3 27 = 3 и 33 = 27. Если a = 10, то есть логарифм по основанию 10, тогда мы просто пишем y = log(n). При работе с компьютерами мы часто сталкиваемся с логарифмами по основанию 2, называемые двоичными логарифмами, так что мы используем для них особую запись, lg(n) = log2 (n). Они отличны от так называемых натуральных логарифмов с основанием логарифма е, где e ≈ 2,71828. Натуральные логарифмы также имеют особую запись, ln(n) = loge (n).

Сделаем небольшое отступление для тех, кому интересно, откуда взялось число е. Число e часто называется эйлеровым числом, в честь жившего в XVIII веке швейцарского математика Леонарда Эйлера, оно возникает в различных областях. У числа Эйлера есть форма выражения (1 + 1/n)n , где n стремится к бесконечности. Хоть оно и названо в честь Эйлера, число было открыто другим швейцарским математиком, жившим в XVII веке, Якобом Бернулли. Он пытался вывести формулу для расчета постоянно растущих процентных доходов.

Допустим, вы положили d долларов в банк и вклад приносит вам доход R%. Если процентный доход считается раз в год, тогда через год ваш доход составит d + d(R/100). Берем r = R/100, тогда выгода будет d (1 + r). Можете удостовериться, что если R = 50, r = 1/2, то ваша выгода вырастет до 1.5 × d. Если проценты считаются дважды в год, тогда размер процентной ставки каждые шесть месяцев будет r/2. Через шесть месяцев у вас будет d(1+ r/2). Тогда через следующие шесть месяцев, к концу года, у вас будет d(1 + r/2)(1 + r/2) = d(1 + r/2)2 . Если процент рассчитывается по сложной процентной ставке, n раз в год, то вы получите d(1 + r/n) n к концу года. При солидной ставке R = 100% вы получите r = 1; если процент будет бесконечно рассчитываться по сложной процентной ставке, то за ничтожно малое время n возрастет до бесконечности. Тогда если d = 1, ваш доллар к концу года вырастет до (1 + 1/n)n = e. Конец отступления.

Основная особенность логарифмов заключается в том, что логарифмы с разными основаниями отличаются друг от друга постоянным множителем, так как loga (n) = logb (n)/logb (a). Например, lg(n) = log10(n)/log10(2). Таким образом, мы объединяем все логарифмические функции в категорию с общей сложностью, которую мы обычно обозначаем O(log(n)), хотя более специфичное O(lg(n)) тоже встречается довольно часто. Сложность O(lg(n)) появляется, когда алгоритм постоянно разделяет задачу пополам, потому что если вы раз за разом разделяете что-то надвое, то по сути вы прибегаете к логарифмической функции. Значимыми алгоритмами логарифмического времени являются алгоритмы, связанные с поиском: быстрейшие поисковые алгоритмы работают за логарифмическое по основанию 2 время

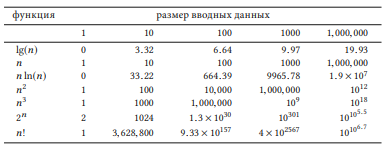
Еще большего времени, чем алгоритмы логарифмического времени, требуют алгоритмы линейного времени, которые выполняются за f(n) = n время, иными словами, за время пропорциональное их вводным данным. Для этих алгоритмов сложность составляет O(n). Для получения ответа им приходится просматривать все вводные данные по порядку. Например, если мы ищем ряд случайных элементов, не заданных ни в каком порядке, тогда нам придется перебрать их все, чтобы найти тот, который нам нужен. Таким образом происходит поиск линейного времени.

Медленней алгоритмов линейного времени — линейно- логарифмические, где f(n) = n log(n), поэтому мы пишем O(n log(n)). Как говорилось выше, у логарифмов могут быть различные основания, хотя на практике намного чаще используются логарифмы с основанием 2. Данные алгоритмы в некотором роде являются смесью алгоритмов линейного времени и алгоритмов логарифмического времени. Они постоянно разделяют задачу надвое и используют алгоритм линейного времени для каждой из частей. Алгоритмы, хорошо справляющиеся с сортировкой, всегда имеют линейно-логарифмическую временную сложность.

Когда функция, описывающая время выполнения алгоритма, полиноминальная f(n) = (a1 nk + a2 nk – 1 + … + an n + b), мы, как видно выше, имеем дело со сложностью O(nk ) и алгоритмом полиноминального времени. Многие алгоритмы выполняются за полиноминальное время; важный их подвид — алгоритмы, которые работают за O(n2 ) время, их называют алгоритмами квадратичного времени. Некоторые методы сортировки квадратичного времени работают непродуктивно, так как используется обычный способ умножения двух чисел с n разрядами у каждого — обратите внимание, что существуют более современные и эффективные способы умножения чисел, и мы используем эти более эффективные способы, ожидая проведения точнейших арифметических расчетов.

Еще медленнее, чем алгоритмы полиноминального времени, алгоритмы экспоненциального времени, где f(n) = cn и c имеет постоянное значение, поэтому O(cn ). Обязательно обратите внимание на разницу между nc и cn. Хотя мы поменяли местами n и показатель степени, это сильно сказывается на результате функции. Как мы говорили, возведение в степень — обратный процесс от логарифмирования и он попросту увеличивает константу до непомерного числа. Будьте внимательны: возведение в степень — это cn ; а степенная зависимость — это особый случай, где c = e, то есть f(n) = ex , где e является числом Эйлера. Возведение в степень случается, когда нам приходится иметь дело с проблемой вводных данных n, где каждые n данных могут принять c разных значений, и нам нужно рассмотреть все возможные случаи. У нас есть c значений для первого ввода данных, и для каждого из случаев у нас есть c значений для второго ввода данных; в общей сложности c × c = c2 . Для каждого из случаев c2 у нас есть c возможных значений для третьего ввода данных, который делает c2 × c = c3 ; и так до последнего ввода данных, который дает cn.

**Таблица 1.1**. Рост функций

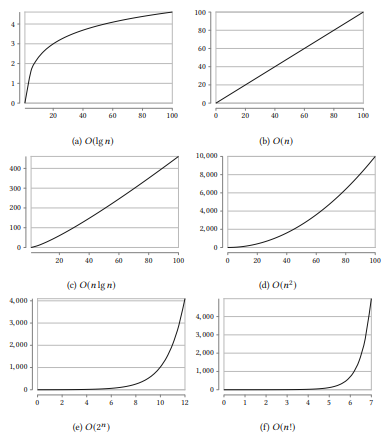


Еще медленнее, чем алгоритмы экспоненциального времени, алгоритмы факториального времени с O(n!), где факториальное число задано как n! = 1 × 2 × … × n и в вырожденном случае 0! = 1. Факториал появляется на сцене, когда для решения задачи нам нужно перебрать все возможные комбинации входных данных. Комбинация — это иное расположение последовательности значений. Например, если мы имеем значения [1, 2, 3], тогда у нас есть следующие комбинации: [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2] и [3, 2, 1]. В первой позиции есть n возможных значений; затем, так как мы использовали одно значение, есть n – 1 возможных значений для второй позиции; это создает n × (n – 1) различных комбинаций для первых двух позиций. Мы в том же духе продолжаем перебор для оставшихся позиций, пока в последней не останется единственно возможное значение. В итоге у нас n × (n – 1) … × 1 = n! Мы встречаем факториальное число при раздаче колоды игральных карт: число возможных тасовок в карточной колоде — 52, и это просто заоблачное число.

Практика показывает, что алгоритмы с полиноминальной временной сложностью хороши в работе, так что зачастую наша задача заключается в том, чтобы отыскать алгоритмы с такой вот производительностью. К сожалению, мы не знаем ни одного алгоритма полиноминального времени, который подошел бы для решения целого круга важнейших задач. Посмотрите на таблицу 1.1; вам должно стать понятно, что если для задачи у нас есть только алгоритм со временем выполнения O(2n ), то такой алгоритм довольно бесполезен и годится разве что для пустячных задач с малыми вводными значениями. Также обратите внимание на рисунок 1.3; в нижнем ряду O(2n ) и O(n!) начинают стремительно увеличиваться для малых значений n.

На рисунке 1.3 линиями изображены графики функций, хотя на деле число n, когда мы изучаем алгоритмы, является натуральным числом, так что нам бы следовало ожидать график рассеяния с точками, а не линиями. Логарифмические, линейные, линейнологарифмические и полиноминальные функции задаются напрямую для действительных чисел, поэтому их совершенно спокойно можно изобразить линиями, используя нормальные функциональные определения. Обычное объяснение возведения в степень касается целых чисел, однако также возможна и степень с рациональным показателем, потому что x(a/b) = (xa) (1/b) = √b xa . Степени с показателями, являющимися действительными числами, определяются как bx = (elnb ) x = ex lnb. Что касается факториалов, то, углубляясь чуть дальше в высшую математику, получается, что они тоже могут быть заданы для всех действительных чисел (отрицательные факториалы считаются равными бесконечности). Таким образом, изображение функций сложности с помощью линий вполне оправданно.

Чтобы вы не подумали, что сложность O(2n ) или O(n!) редко встречаются на практике, рассмотрите известную (скорее даже печально известную) задачу коммивояжера. Суть задачи в том, что коммивояжер должен посетить определенное количество городов, заезжая в каждый город только один раз. Каждый город напрямую связан с каждым другим (возможно, коммивояжер путешествует самолетом). Загвоздка в том, что коммивояжер должен найти самый короткий маршрут. Решение в лоб — это перепробовать все возможные комбинации с очередностью городов. Для n городов — O(n!). Есть алгоритм и получше, он решает задачу через O(n2 2n ) — хоть он и быстрее, однако на практике разница едва ли заметна. Так как же нам решить эту (и прочие похожие) задачи? Оказывается, что, хотя мы и не знаем хорошего алгоритма, который выдаст нам точный ответ, нам могут быть известны хорошие алгоритмы, которые дадут приблизительные результаты.



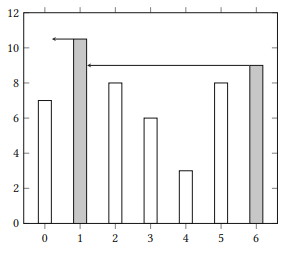
Большое О задает верхнюю границу производительности алгоритма. Противоположная ей нижняя граница, когда мы знаем, что сложность всегда будет не лучше, чем у заданной функции, после нескольких исходных значений. Такое поведение называется «большая Омега», или Ω(f(n)), а точное ее определение: Ω(f (n)) = g(n), если существует положительные константы c и n0 , такие как f (n) ≥ cg(n) ≥ 0 для всех n ≥ n0 . Имея заданные большое О и большую Омегу, мы можем также задать ситуацию, где у нас есть обе, и верхняя, и нижняя границы. Такая ситуация называется «большая Тета», и мы говорим, что Θ(f (n)) = g(n), если и только если O(f (n)) = g(n) и Ω(f (n)) = g(n). Тогда мы знаем, что время выполнения алгоритма ограничено сверху и снизу одной и той же функцией, зависящей от константы. Можете считать это временем выполнения алгоритма, лежащим в области этой функции.

**1.3. Используем стек для разницы курсов акций**

Вернемся к задаче о разнице курсов акций. Мы нашли алгоритм со сложностью O(n(n + 1/2)). Из написанного выше следует, что он равноценен O(n2 ). Можем ли мы сделать лучше? Вернемся к рисунку 1.1. Обратите внимание: когда мы на шестом дне, нам не надо сравнивать все предыдущие дни до первого, так как мы уже «знаем», что стоимость во второй, третий, четвертый и пятый дни ниже или равна стоимости в день шестой. Если мы каким-то образом сохраним это знание, то вместо того, чтобы заниматься лишними сравнениями, нам надо будет сопоставить только курс акций в шестой и первый дни.

Мы коснулись базового момента. Представьте, что вы находитесь в дне k. Если цена акций в день k – 1 ниже или равны цене в день k так, что цены quotes[k – 1] ≤ цен quotes[k] или то же самое для цен quotes[k] ≥ quotes[k – 1], тогда не имеет смысла еще раз сравнивать с k – 1. Почему? Заглянем в будущее, в день k + 1. Если курс в день k+j меньше или равен курсу в день k, quotes[k+ j] < цены[k], тогда нам не надо сравнивать с k – 1, так как промежуток разницы, который начинается с k + j, заканчивается в k. Если цена в k + j больше цены в k, тогда мы понимаем, что она должна быть quotes[k + j] ≥ quotes[k – 1], потому что quotes[k + j] ≥ quotes[k] и quotes[k] ≥ quotes[k – 1]. Так что каждый раз, когда мы возвращаемся в прошлое в поисках конца отрезка, обозначающего разницу, нам нужно отбросить все дни со значениями меньше или равными дню, в который мы высчитываем разницу, и исключить отброшенные дни из последующего анализа.

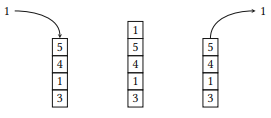
Для большей ясности приведем метафору. Представьте, что сидите на верхушке колоны, которая изображена на рисунке 1.4 и обозначает шестой день. Вы глядите не вниз, а назад, и видите перед собой только колонну, обозначающую день первый. Она — единственное, с чем вам надо сравнивать стоимость акций в шестой день. Таким образом, вам всегда нужно сравнивать нынешний день только с тем, который попадает в поле вашего зрения.

****

**Рисунок 1.4.** Оптимизированная разница курсов акций

Получается, мы впустую тратим время в алгоритме 1.1, когда во внутреннем цикле, идущем от строчки 5, начинаем сравнивать все предыдущие дни. Мы можем избежать лишней работы, используя механизм, с помощью которого у нас всегда будут наготове границы самых больших промежутков.

Помочь в нашей работе может специальная структура для хранения данных, которая называется стеком. Стек — простая структура данных. Такая вещь, на которую мы можем сложить одни за другими все данные, а затем забрать их. Забирать их можно только в обратном порядке, начиная с последних, положенных в стек. Стек устроен так же, как стопка подносов в ресторане: мы можем взять только верхний поднос, а если хотим положить новый, то кладем только сверху. Последний поднос, положенный в стопку, вынимается первым, поэтому стек называют структурой с принципом «последним зашел — первым вышел». На рисунке 1.5 изображены похожие на подносы операции с добавлением и извлечением элементов стека.

****

**Рисунок 1.5. Добавление и извлечение из стека**

Когда мы говорим о структурах данных, нам нужно описать операции, которые мы способны производить над ними. Для массивов мы рассмотрели две операции: одна — создание массива, вторая — добавление элементов. Для стеков, учитывая все вышесказанное, существует пять операций.

. • CreateStack() создает пустой стек.

• Push(S, i) добавляет элемент i в самый верх стека S.

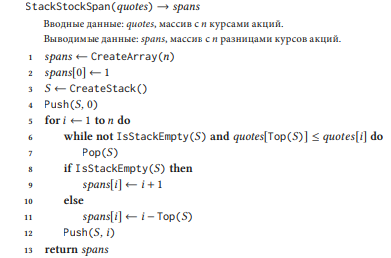
• Pop(S) извлекает верхний элемент стека S. Операция возвращает элемент. Если стек пуст, тогда операция не может быть выполнена (мы получим ошибку). • Top(S) мы получаем значение верхнего элемента стека S, не извлекая при этом сам элемент. Стек не изменяется. Если стек пуст, тогда, опять же, операция не может быть выполнена и мы увидим сообщение об ошибке.

• IsStackEmpty(S) возвращает true, если стек S пуст, или false, если стек S полон.

На самом деле стек не безграничен: мы можем поместить в него определенное количество элементов, пока он не заполнится: в конце концов, у памяти компьютера есть свои пределы. При реализации стека есть дополнительные операции для проверки количества элементов в стеке (его размер) и его заполненности: есть ли в нем еще место или нет. К нашим алгоритмами, использующим псевдокод, это не имеет никакого отношения, поэтому в дальнейшем мы не будем их касаться; точно так же мы поступим с прочими схожими операциями для других структур данных.

С помощью стека мы можем справиться с задачей о разнице курсов акций, применив задумку, которую мы развили в алгоритме 1.2. Как и прежде, в строчке 1 мы создаем массив на n элементов. Разница в первый день по умолчанию равна единице, поэтому, согласно строчке 2, мы запускаем spans[0]. Теперь мы используем стек, чтобы поместить в него все дни, которые нужно сравнить, поэтому в строчке 3 мы создаем новый пустой стек. В начале нам известен лишь очевидный факт, что цена акций в первый день не ниже их цены в первый день, поэтому в строчке 4 мы кладем в стек индекс первого дня, ноль.

**Алгоритм 1.2**. Алгоритм вычисления разницы курсов акций с помощью стека

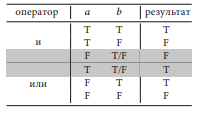


Цикл в строчках 5–12 работает со всеми последующими днями. Внутренний цикл в строчках 6–7 заглядывает в прошлое, чтобы найти последний день, когда курс акций был выше, чем в выбранный нами день. Он проделывает это, вынимая объекты из стека (строчка 7) до тех пор, пока у дня, находящегося вверху стека, не будет меньшая или равная цена по сравнению с выбранным нами днем (строчка 6). Если мы выйдем из внутреннего цикла после того, как стек опустел (строчка 8), и будем находиться в дне i, тогда во все предыдущие дни цена акций будет ниже, поэтому разница будет i + 1. В строчке 9 мы устанавливаем spans[i] для данного значения. Иначе (строчка 10) промежуток тянется дальше от дня i до дня, находящегося наверху стека, так что в строчке 11 мы используем разницу между двумя установками spans[i]. Прежде чем вернуться к началу цикла, мы убираем i с верхушки стека. Таким образом в конце внешнего цикла в стеке останутся дни, в которые цена акций не ниже, чем в выбранный день. Это позволит нам при следующем проходе цикла сравнивать только нужные дни: те, которые которые попадают в поле нашего зрения и которые мы как раз искали.

В строчке 6 нашего алгоритма есть один момент, на который стоит обратить внимание. Речь об ошибке при оценке Top(S), если S пуст. Такого не случится благодаря важному свойству, касающемуся оценки условий, которое называют упрощенным вычислением. Данная характеристика означает, что, когда мы вычисляем выражение, в котором предполагаются логические булевы операторы, вычисление такого выражения прекращается, как только мы получаем итоговый результат: мы не затрудняем себя просмотром остальных частей выражения. Например, выражение если x > 0 и y > 0. Если мы знаем, что x ≤ 0, тогда все выражение ложно, вне зависимости от значения y; нам вовсе не нужно вычислять вторую часть выражения. Точно так же с выражением если x > 0 или y > 0, если мы знаем, что x > 0, то нам не надо вычислять вторую часть выражения с y, так как мы уже знаем, что все выражение истинно, потому что верна первая его часть. Таблица 1.2 показывает общую ситуацию для всех логических выражений, состоящих из двух частей и содержащих операторы и либо или.

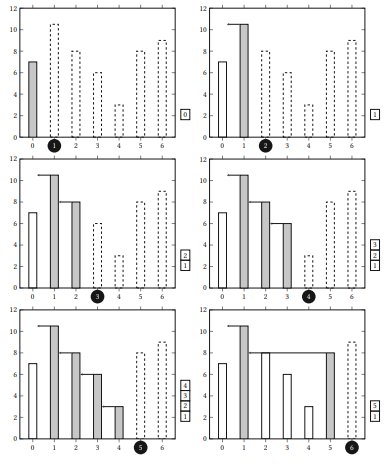
Закрашенные серым ряды показывают, что результат выражения не зависит от второй части и, следовательно, вычисление можно прекратить, как только нам станет известно значение первой части. С помощью упрощенного вычисления, когда IsStackEmpty(S) показывает true, означающее, что не IsStackEmpty(S) является false, мы не будем пытаться вычислить находящееся справа от и, содержащее Top(S), тем самым избегая ошибку.

**Таблица 1.2.** Упрощенное булево вычисление



На рисунке 1.6 вы можете увидеть, как работает наш алгоритм и метафора «взгляда с колонны». На каждом графике мы показываем справа стек в начале каждой итерации цикла; мы также отмечаем закрашенными столбцами дни, находящиеся в стеке, а дни, с которыми мы еще не разобрались, отмечаем пунктирными столбцами. Выбранный нами день, с которым мы работаем, выделен внизу черным кругом.

На первом графике i = 1, и мы должны сравнить значение выбранного дня со значениями других дней, находящихся в стеке: а в стеке у нас только нулевой день. В первый день цена выше, чем в нулевой. Это означает, что с нынешнего момента не нужно сравнивать дни до первого дня — наш взгляд с колонны остановится на нем; потому при следующей итерации цикла, когда i = 2, в стеке содержится число 1. В день второй цена ниже, чем в день первый. Это означает, что любой промежуток, начавшийся с третьего дня, может закончиться на втором дне, если значение третьего дня ниже, чем значение второго; или же он может закончиться на первом дне, если значение третьего дня не меньше, чем во второй день. Но он никак не может закончиться на нулевом дне, даже если цена в нулевой день меньше, чем в первый день.



**Рисунок 1.6**. Взгляд на промежуток, обозначающий разницу курсов акций

Похожая ситуация возникает с i = 3 и i =4. Но когда мы доходим до i = 5, мы понимаем, что больше не нужно сравнивать второй, третий и четвертый дни — они покрыты тенью пятого дня. Если вернуться к метафоре взгляда с колонны, ничто не закрывает наш обзор вплоть до первого дня. Все, что находится между пятым и первым днем, можно убрать из стека, в котором останутся лишь 5 и 1, таким образом, i = 6 нам нужно будет сравнить только с этими двумя днями. Если цена в какой-либо день больше или равна цене в день пятый, она уж точно больше стоимости в четвертый, третий и второй дни; чего мы не можем сказать наверняка — сможет ли она достигнуть значения первого дня. Когда мы доходим до шестого дня, в стеке оказываются числа 6 и 1.

Мы можем продолжить наш анализ и увидеть, что в сравнении с алгоритмом 1.1, в котором мы смогли прийти только к худшему варианту, здесь наша оценка также достигает нижней границы производительности алгоритма: так как нам надо пройти через n-е количество дней, алгоритм не может быть выполнен за менее чем n этапов. Таким образом, вычислительная сложность нашего алгоритма тоже Ω(n) и, стало быть, Θ(n).

Стеки, как и все структуры данных, с которыми мы познакомимся, имеют широкий спектр применения. Поведение «последним вошел — первым вышел» довольно часто встречается в программировании, так что стеки будут попадаться вам везде: от простейших программ, написанных на машинных языках, до объемных задач, решаемых с помощью сверхмощных компьютеров. Именно поэтому вообще и существуют структуры данных. Они являются выжимкой многолетнего опыта решения задач с помощью компьютера. Снова и снова получается так, что алгоритмы используют схожие способы для организации данных, которые они обрабатывают. Люди перевели эти способы в код так, что, когда мы ищем разные подходы к решению задачи, мы прибегаем их возможностям, чтобы разработать алгоритмы.

**Упражнения**

Необходимо выполнить максимальное количество задач (желательно все). Работу выполнять на C# в MS Visual Studio.

1. Стек — несложная для реализации структура данных. Простейшая ее реализация использует массив; напишите реализацию стека на основе массива. Ранее мы говорили о том, что на практике стек имеет куда больше операций, чем пять, упомянутых в данной книге: операции, возвращающие размер стека и проверяющие его наполненность. Убедитесь, что их вы тоже реализовали.

2. Мы рассмотрели два решения для задачи о разнице курсов акций: одна использует стек, а другая обходится без него. Мы говорили, что способ с применением стека быстрее. Проверьте это на практике: реализуйте два алгоритма на выбранном вами программном языке и засеките, сколько времени потребуется на каждое решение задачи. Обратите внимание, что при расчете времени выполнения программы вы должны предоставить достаточно данных, чтобы ей потребовалось разумное количество времени на выполнение работы; учитывая, что в компьютере параллельно происходит множество процессов и время выполнения может попадать под влияние разных факторов, вам нужно не один раз замерить время, чтобы в итоге получить достоверный результат. Так что у вас отличная возможность узнать, как тестируются программы.

3. Стеки используются для реализации арифметических вычислений, написанных на обратной польской нотации (ОПН), также известной как постфиксная запись. В ОПН идут сначала операнды, а потом их операторы, в отличие от инфиксной записи, где операторы располагаются между операндами. Так что вместо 1 + 2, мы пишем 1 2 +. «Польской» нотация называется в честь Яна Лукасевича, который в 1924 году изобрел польскую, или префиксную, запись, на ней у нас получится + 1 2. Преимущество ОПН состоит в том, что здесь не нужны скобки: 1 + (2 × 3) из инфиксной записи превращается в 1 2 3 \* + постфиксной записи. Чтобы вычислить результат, мы читаем запись слева направо. Мы добавляем числа в стек. Когда мы сталкиваемся с оператором, мы извлекаем из стека столько элементов, сколько нужно в качестве операндов, мы производим операцию и помещаем результат в стек. В итоге результат оказывается на самом верху стека (и единственным его элементом). Например при вычислении 1 2 3 \* + 2 – стек, написанный горизонтально в квадратных скобках, становится [ ], [1], [1 2], [1 2 3], [1 6], [7], [7 2], [5]. Напишите калькулятор, который вычисляет арифметические выражения, заданные пользователем в ОПН.

4. Во многих языках программирования мы имеем дело с выражениями, выделенными символами-разграничителями, такими как круглые скобки ( ), квадратные скобки [ ] или фигурные скобки { }. Напишите программу, которая читает последовательность символов-разграничителей, такую как ( ) { [ ] ( ) { } }, и сообщает, когда разграничители симметричны или когда несимметричны, например ( ( ); или когда разграничители разных видов стоят в паре, как ( }. Используйте стек, чтобы запомнить открытые в настоящий момент символы-разграничители.