Глава 1
Введение

Представленная вашему вниманию работа составлена по мотивам лекций и семинаров курса «Алгоритмы и структуры данных для поиска», читаемого в Школе анализа данных Яндекса. Полный курс состоит из двух семестров, мы же пока публикуем материалы первого из них с надеждой, что в скором времени мы найдем в себе силы подготовить также и вторую часть.

Как уже было отмечено выше, занятия состоят из лекций и семинаров. На лекциях обсуждаются разнообразные теоретические вопросы, а также изучаются приемы реализации стандартных алгоритмов. Семинары в большей степени ориентированы на решение задач.

Новые комплекты практических заданий периодически выдаются на семинарах, и слушатели должны в течение установленного срока прислать свое решение. Решение состоит из двух частей: теоретическое обоснование выбранного метода (включая оценки сложности) и практическая реализация. В качестве последней принимается код на языке C++.

Реализации, присылаемые студентами Школы, проверяются в автоматическом режиме на наборе заранее подготовленных тестов. Для прохождения теста необходимо выдать правильный ответ, не превышая при этом установленного предела по ресурсам (времени и памяти). Решение засчитывается в случае, если оно успешно проходит все тесты.

С учетом этого процесса и был написан данный конспект. Структурно он состоит из набора *глав*, каждая из которых фокусируется на определенном разделе теории. В конце каждой главы вы найдете примеры *задач*. Для многих задач мы приводим формат ввода-вывода, который ожидается от программ, а также несколько примеров входных и выходных данных. Часть из этих задач, впрочем, более теоретическая, поэтому для них подобных форматов не приводится.

Для всех задач вы найдете описание теоретической части решения в форме, близкой к той, что ожидается от слушателей. Некоторые из этих описаний не вполне формальны, но надеемся, что придирчивый читатель простит нам подобные вольности.

Несмотря на то что данный курс сам по себе является вводным, мы предполагаем, что читатель имеет хотя бы минимальное представление о том, что из себя представляет программирование. В частности, мы предполагаем известными основные понятия какого-либо типичного императивного языка программирования.

Большинство примеров в данной книге будут проиллюстрированы на языке C++. Причиной этого является его широкая распространенность в качестве практического средства, позволяющего писать эффективный и переносимый код.

Помимо знания C++ как такового, мы считаем, что читатель знаком со стандартной библиотекой языка. Это, конечно, не предполагает автоматического понимания устройства структур данных и алгоритмов, присутствующих там. Мы считаем известным разделение на *интерфейс* и *реализацию*. В случае со стандартной библиотекой речь идет о понимании ее интерфейсов, а также их неотъемлемой части — гарантии *сложности* операций. Такие гарантии определяют поведение библиотечного кода в худшем случае.

И все же наша цель — изучение базовых эффективных алгоритмов. Такая цель не предполагает привязку к какой-либо платформе и библиотеке. В частности, желательно разобраться с тем, как именно реализованы библиотечные средства.

1.1. Массивы переменного размера

Читатель, без сомнения, знаком с понятием *массива*. Массив — это средство языка C++, а не библиотеки. Библиотека же предлагает более удобный аналог массива, а именно *вектор* (класс `std::vector`). С точки зрения интерфейса ключевое различие между вектором и массивом состоит в том, что массив имеет фиксированный, выбираемый в момент его создания размер, а вектор — напротив, может изменять количество лежащих в нем элементов динамически во время работы. Как же реализуется такая возможность на практике? Вопрос этот не вполне тривиальный, но достаточно простой, чтобы начать изложение материалов курса с него.

Будем рассматривать простейший случай: нам необходимо предложить структуру данных, хранящую последовательность элементов переменной длины и поддерживающую операцию Insert добавления нового элемента в конец последовательности. Конечно, операция Insert должна быть по возможности быстрой. Кроме того, должна быть возможность обратиться к любому элементу по его индексу за время $O(1)$.

Последнее требование означает, что наиболее близкая к требуемой структура — это обычный массив фиксированного размера, не меньшего текущей длины последовательности. Такое решение также упрощает (по крайней мере в языке C++) взаимодействие с кодом, который ожидает получить на вход обычный массив. Этот массив обычно задается указателем на свой начальный (нулевой) элемент, и в случае вектора такой указатель всегда легко получить. Иными словами, проектируемый нами вектор гарантирует непрерывное расположение своих элементов в памяти.

Поскольку длина массива в общем случае больше текущей длины последовательности, последнюю нужно явно хранить. Возникает естественный вопрос: что же делать в случае, когда происходит вызов Insert, а длина последовательности уже совпадает с длиной массива, так что свободных элементов в конце его нет?

В этом случае придется выполнить операцию *перевыделения* (*reallocation*): создать новый массив большего размера и скопировать в него информацию из старого массива. Эта операция, конечно, не бесплатная: ее временна́я сложность пропорциональна количеству элементов, подлежащих копированию.

Но как выбрать этот новый размер массива? К примеру, можно было бы каждый раз увеличивать размер массива при перевыделении на какое-либо постоянное количество элементов d. Такой метод называется *аддитивным*. Несложно убедиться, что общее время, которое будет затрачено на выполнение последовательности из n операций Insert, составляет[1] $\Omega(n^2)$.

Упражнение 1.1.1. Скрытая константа в этой оценке зависит от d. Как именно?

[1] Мы используем ряд стандартных обозначений асимптотического анализа. Для положительных функций f и g записи $f = O(g)$, $f = \Omega(g)$ обозначают соответственно, что найдется такая константа $C > 0$, что неравенства $f \leqslant C \cdot g$ и $f \geqslant C \cdot g$ выполнены для всех достаточно больших значений аргументов. Если $f = O(g)$ и $g = O(f)$, то пишем $f = \Theta(g)$.

Квадратичную общую сложность последовательности из n вставок, оказывается, возможно уменьшить до линейной. Для этого нужно использовать *мультипликативный* метод: получать новую длину умножением старой на некоторую константу $\alpha > 1$. Для простоты положим $\alpha = 2$, так что размер массива каждый раз при необходимости будет удваиваться (при этом, конечно, начинать нужно с длины 1, а не 0).

Доказать, что последовательность из n операций Insert будет выполняться за время $O(n)$, несложно и непосредственно. Сейчас, однако, будет описан некоторый общий метод получения подобных оценок, который впоследствии нам не раз пригодится.

1.2. Анализ учетных стоимостей

Будем изучать общую ситуацию, при которой имеется некоторая структура данных S. Предположим, что над данной структурой данных выполняется последовательность операций $a_1, \ldots, a_n$. Каждая операция a_i требует определенных временны́х затрат, которые мы обозначим через $c(a_i)$. Числа $c(a_i)$ будем называть *фактическими* стоимостями операций.

Обозначим через s_i состояние структуры S после выполнения i операций ($0 \leqslant i \leqslant n$). В частности, s_0 — это начальное состояние до выполнения операций, а s_n — заключительное состояние, в которое структура переходит после выполнения всех операций из списка.

Наша задача состоит в том, чтобы оценить сумму времен, которые затрачиваются на выполнение всех n операций, т. е.

$$\sum_{i=1}^{n} c(a_i). \tag{1.1}$$

Трудность получения оценки данной суммы связана с тем, что фактические стоимости зачастую слишком сильно различаются. Поэтому оценка вида Cn, где C означает максимально возможную фактическую стоимость, может оказаться слишком грубой.

Предположим, что с каждым из состояний s_i связано некоторое вещественное значение φ_i ($0 \leqslant i \leqslant n$), называемое *потенциалом*. Тогда можно определить значения

$$c'(a_i) := c(a_i) + \varphi_i - \varphi_{i-1} \quad \text{для всех } i = 1, \ldots, n,$$

которые мы будем называть *приведенными* (или *учетными*) стоимостями. Идея состоит в том, что подходящим выбором потенциалов иногда удается добиться того, что максимальное возможное значение приведенной стоимости операции (обозначим его C') оказывается намного меньше максимально возможного фактического значения. Это позволяет оценить сумму (1.1) так:

$$\sum_{i=1}^{n} c(a_i) = \sum_{i=1}^{n} c'(a_i) + \varphi_0 - \varphi_n \leqslant C'n + (\varphi_0 - \varphi_n). \qquad (1.2)$$

Если дополнительно известно, что $\varphi_n \geqslant \varphi_0$, то оценка упрощается до $C'n$.

Применим введенную технику для анализа суммарной сложности последовательности из n вставок в вектор. Фактическая стоимость одной операции Insert зависит от того, происходит ли на данном шаге перевыделение. Если нет, то время работы можно принять за одну условную единицу. Если же перевыделение необходимо, то время работы составляет $m+1$ условных единиц, где m — текущий размер массива (m единиц уходят на копирование существующих значений, а еще одна уходит на сохранение нового значения в перевыделенном участке памяти).

Как видно, фактические времена выполнения операций сильно разнятся. Чтобы выровнять их, введем потенциал следующим образом. Заметим, что если размер массива на каждом перевыделении удваивается, то в любой момент времени, кроме начального (когда вектор пуст), количество занятых элементов s в массиве не меньше половины его длины l. Потенциал φ состояния структуры данных примем равным $2s - l$.

В случае добавления без перевыделения потенциал меняется на $O(1)$, значит, учетная стоимость операции также составляет $O(1)$.

Более интересен случай добавления с перевыделением. Тогда $l = s$, и операция Insert стоит $s+1$ условных единиц. После перевыделения потенциал равен 2, а значит, увеличение потенциала равно $2 - (2s - s) = 2 - s$. Учетная стоимость операции Insert в итоге составляет $(s+1) + (2-s) = 3$ единицы.

В обоих случаях получаем оценку $O(1)$ на учетную стоимость операции Insert. Кроме того, конечный потенциал не меньше начального. Следовательно, по формуле (1.2) общая сложность последовательности операций составляет $O(n)$.

Упражнение 1.2.1. Покажите, что оценка $O(n)$ на общее время выполнения n операций Insert остается справедливой и для любого другого коэффициента перевыделения $\alpha > 1$. Как зависит скрытая в оценке $O(n)$ константа от α?

1.3. Задачи

Задача 1. *Правильной скобочной последовательностью* называется строка, состоящая только из скобок, в которой все скобки можно разбить на пары таким образом, что

- в каждой паре есть левая и правая скобка, причем левая скобка расположена левее правой;
- для любых двух пар скобок либо одна из них полностью находится внутри другой пары, либо промежутки между скобками в парах не пересекаются;
- в паре с круглой скобкой может быть только круглая скобка, с квадратной — квадратная, с фигурной — фигурная.

Примеры:

- если разрешены только круглые скобки:
  - правильные последовательности: `()`, `(())`, `()()`, `()()()`, `(())()`, `((()))`;
  - неправильные последовательности: `)(`, `))`, `((`, `())()(`, `())`, `))((`;
- если разрешены круглые и квадратные скобки:
  - правильные последовательности: `[]`,`()`, `[()]`, `[[([])]()]`;
  - неправильные последовательности: `[)`, `([)]`, `(())()[]][`;
- если разрешены еще и фигурные скобки:
  - правильные последовательности: `[{(())}({})]`, `[]{}()`, `{}`, `()`, `[]`;
  - неправильные последовательности: `[{(})]`, `[(]())]{}`.

Во входном файле задана строка α, состоящая только из скобок (круглых, квадратных и фигурных). Требуется определить, является ли она правильной скобочной последовательностью. Если да, выведите в выходной файл слово `CORRECT`. Если нет, выведите длину максимального префикса α, который либо сам является правильной скобочной последовательностью, либо может быть продолжен до таковой.

Например, для строки `(()))` ответом будет 4, так как строка `(())` является правильной скобочной последовательностью, а строку `(()))` уже нельзя никаким образом продолжить вправо, чтобы получить правильную скобочную последовательность. Для строки `])()(` ответ 0, поскольку строку `]` нельзя продолжить вправо, чтобы получить правильную скобочную последовательность. Для строки `[(()){()()[[]]}]` ответ `CORRECT`.

Указание. Понадобится стек. У класса `std::vector` есть методы `push_back` и `pop_back`, с помощью которых его можно легко имитировать.

Пример входа	Пример выхода
`(())`	`CORRECT`
`([)]`	`2`
`(([{`	`4`

Решение. Рассмотрим первую правую скобку, встречающуюся в строке. Если она находится на первой же позиции, то строка уже некорректна и нельзя ничего приписать справа, чтобы она стала корректной (не найдется пары к самой первой правой скобке).

Если же данная скобка встречается на позиции $r > 0$, то скобка на позиции $r - 1$ является левой, и она является парной к данной правой скобке. Действительно, если бы какая-то левая скобка на позиции $l < r - 1$ была парой к правой скобке на позиции r, то пара к левой скобке на позиции $r - 1$ находилась бы правее r, и тогда эти две пары пересекались бы, что противоречит определению. С другой стороны, если самая первая правая скобка стоит на позиции $r > 0$, то строку еще можно продолжить так, чтобы она стала корректной: нужно дописать столько правых скобок, чтобы закрыть все левые скобки, и остановиться. Поэтому самая первая правая скобка и левая скобка сразу перед ней соответствуют друг другу. Теперь, если выкинуть эту пару скобок из строки, корректность оставшейся строки определяется тем же самым способом, так как пара скобок, стоящая непосредственно рядом, без промежуточных скобок, не мешает выполнению ни одного из наложенных условий. Из этих соображений вытекает следующий алгоритм решения задачи.

Из определения корректной скобочной последовательности следует, что при чтении такой последовательности слева направо всегда есть набор открывающих скобок, ожидающих себе парных за-

крывающих скобок (по первому свойству). При этом если встречается очередная закрывающая скобка, то она должна быть парной именно к последней открывающей скобке, иначе нарушилось бы второе свойство. В тот момент, когда для последней открывающей скобки в наборе находится парная закрывающая, эта открывающая скобка из набора удаляется, ожидающих открывающих скобок становится на одну меньше, и последней становится та, что была предпоследней. В конце, когда мы прочитали все символы строки, стек должен оказаться пустым: к каждой открывающей скобке нашлась парная. Кроме того, в процессе обработки никогда не должно получиться, что мы читаем закрывающую скобку, а стек либо пуст, либо на его вершине находится неправильная открывающая скобка. В этом случае последовательность сразу же получается некорректной, и к ней уже справа ничего нельзя дописать, чтобы последовательность стала корректной.

Такое поведение ожидающих открывающих скобок позволяет хранить их в стеке, используя следующий алгоритм. Изначально стек пустой. Идем по строке слева направо, считывая по одному символу. В зависимости от очередного символа, выполняем действия.

- Если очередной символ — открывающая скобка, то кладем его на стеки, двигаемся дальше.
- Если очередной символ — закрывающая скобка, то посмотрим на состояние стека. Если на стеке нет ни одного элемента или открывающая скобка на вершине стека не соответствует текущей закрывающей скобке, то последовательность неправильная. При этом она могла быть дополнена до правильной последовательности до текущего момента (обоснуйте), а после того, как найдена неправильная закрывающая скобка, она уже не может быть дополнена до правильной. Соответственно, нужно вывести длину прочитанного префикса, не включая последнюю прочитанную скобку, и выйти из программы. Если стек не пуст и в вершине стека лежит соответствующая открывающая скобка, удаляем из стека эту скобку и двигаемся дальше.

В конце, если пройдена вся строка до конца, нужно еще не забыть определить, является ли прочитанная строка правильной скобочной последовательностью или же только началом правильной скобочной последовательности. Если стек в конце просмотра стро-

ки пуст, то прочитанная строка является правильной скобочной последовательностью и нужно выдать CORRECT. Если же стек не пуст, то строку можно дополнить до правильной скобочной последовательности, но она еще такой не является, поэтому нужно выдать длину всей строки.

Для анализа каждого очередного символа строки требуется константное время, поэтому общая сложность алгоритма составляет $O(n)$. Памяти требуется $O(n)$ на хранение самой строки и $O(n)$ на стек.

Задача 2. Дана последовательность $a_1, a_2, \ldots, a_n$ из n различных целых чисел. Для каждого числа a_i найдите наименьшее такое j, что $j > i$ и $a_j > a_i$.

В первой строке входного файла записано число n. Во второй строке — числа $a_1, a_2, \ldots, a_n$; $1 \leqslant n \leqslant 10\,000\,000$, $-2^{31} \leqslant a_i \leqslant 2^{31} - 1$.

В выходной файл для каждого i, $1 \leqslant i \leqslant n$, выведите соответствующее j. Если правее a_i нет чисел, больших его, в качестве j выведите -1.

Пример входа	Пример выхода
5 1 3 2 4 5	2 4 4 5 -1
1 100	-1

Решение. Будем просматривать последовательность слева направо. Заведем стек, в котором будем хранить наибольшее из уже просмотренных чисел, затем следующее по величине из чисел, стоящих правее него, затем следующее по величине из чисел, стоящих правее, и т. д. Пусть просмотрены все числа $a_1, a_2, \ldots, a_i$. Тогда в вершине стека находится число $a_{k_1} = \max\limits_{1 \leqslant j \leqslant i} a_j$, следующее число в стеке $a_{k_2} = \max\limits_{k_1 < j \leqslant i} a_j$ либо следующего числа нет, если $k_1 = i$. Третье число в стеке, если есть второе и $k_2 < i$, равно $a_{k_3} = \max\limits_{k_2 < j \leqslant i} a_j$, а если $k_2 = i$, то в стеке ровно два элемента, и т. д. Число в вершине стека будет минимальным, и далее числа возрастают от головы к хвосту.

Изначально стек пуст. Когда мы рассматриваем очередной элемент последовательности, то сравниваем его с вершиной стека (если стек непуст). До тех пор пока стек непуст и значение в вершине

стека меньше очередного элемента, удаляем вершину из стека, при этом удаляя элемент a_i из стека. Если текущий рассматриваемый элемент a_j, то ответ для i будет равен j — записываем этот результат. Действительно, до тех пор пока мы не рассмотрели j-й элемент последовательности, из определения множества элементов, хранимых в стеке, следует, что для каждого из них еще не нашлось ни одного элемента последовательности правее его и больше. Соответственно, для всех элементов стека, которые меньше очередного, ответ определяется в этот момент. Когда из стека удалены все элементы, меньшие нового, добавляем новый элемент в стек, так как он является максимальным среди всех элементов правее вершины стека. Таким образом, в стеке всегда будет правильное множество элементов, кроме того, в нем всегда будет содержаться самый правый рассмотренный элемент. Алгоритм даст нам правильный ответ, а для чисел, которые останутся в стеке к моменту, когда уже просмотрена вся последовательность, ответ равен -1.

Сложность каждого отдельного шага не оценивается как $O(1)$, так как в течение одного шага может быть удалено вплоть до $n-1$ элементов из стека, если последовательность была $n-1, n-2, n-3, \ldots, 1, n$ и рассматривается n-й элемент, т. е. в худшем случае сложность каждого шага $O(n)$. Однако если оценить сложность всего алгоритма с помощью амортизационного анализа, то получается, что сложность линейна по n. Действительно, каждый элемент добавляем в стек и удаляем из стека не более одного раза. Значит, в сумме сделаем $O(n)$ операций со стеком, а каждая операция со стеком работает за время $O(1)$. Памяти необходимо $O(n)$.

Упражнение 1.3.1. Каково формальное определение потенциала, использованного в вышеуказанном анализе?