Часть VI Технические вопросы

Подготовка

Если вы купили эту книгу, то, скорее всего, уже проделали достаточно долгий путь к вершине отличной технической подготовки. Поздравляю, прекрасная работа!

Но существуют как лучшие, так и худшие способы подготовки. Многие кандидаты считают, что достаточно просмотреть задачи и готовые решения. Это в чем-то похоже на попытку научиться считать, читая книги. Но если вы хотите научиться решать задачи, запоминание готовых решений не поможет.

Как организовать подготовку

Для каждой задачи из этой книги:

- *Попытайтесь решить задачу самостоятельно*. Я имею в виду действительно попытаться решить задачу. Вам встретится множество сложных заданий — и это нормально. Когда вы решите задачу, подумайте об эффективности использования пространства (памяти) и времени выполнения. Задайтесь вопросом, можно ли ускорить выполнение программы, оптимизируя использование памяти, и наоборот.
- *Запишите код алгоритма на бумаге*. Вы всю жизнь программировали на компьютере, и это, безусловно, хорошо. Но на собеседовании вам не поможет ни подсветка синтаксиса, ни автозавершение кода, ни компиляция. Сымитируйте условия собеседования, записывая код на бумаге.
- *Протестируйте свой код*. Представьте, что вы — компилятор, проверьте код на наличие ошибок. Вам придется это делать на собеседовании, поэтому лучше подготовиться заранее.
- *Введите написанный код в компьютер «как есть»*, возможно, вы обнаружите множество ошибок. Проанализируйте все ошибки и сделайте все, чтобы на настоящем собеседовании их не допустить.

Очень полезны псевдоинтервью. На `CareerCup.com` вы найдете примеры псевдоинтервью с сотрудниками Microsoft, Google, Amazon — используйте их, когда будете практиковаться с друзьями. Хотя ваши друзья не являются профессиональными интервьюерами, они в состоянии проверить решения задач на программирование и алгоритмизацию.

Что нужно знать

Большинство интервьюеров не будут задавать вопросы о конкретных алгоритмах балансировки двоичного дерева или других сложных алгоритмах. Честно говоря, они сами их уже забыли, ведь такие вещи забываются сразу после окончания учебы.

Вам нужно знать основы.

Структуры данных	Алгоритмы	Концепции
Связные списки	Поиск в ширину	Манипуляции битами
Бинарные деревья	Поиск в глубину	Одиночка (шаблон проектирования)
Графы	Бинарный поиск	Фабричный шаблон проектирования

Структуры данных	Алгоритмы	Концепции
Стеки	Сортировка слиянием	Память (стек vs куча)
Очереди	Быстрая сортировка	Рекурсия
Векторы/Списки массивов	Вставка в дерево, поиск и т. д.	Время порядка «О-большое»[1]
Хэш-таблицы		

Убедитесь, что вы понимаете, как использовать и реализовывать каждую из задач, знаете их область применения, эффективность использования памяти и время выполнения. Вам могут задать вопрос по теме из таблицы или попросить реализовать какую-либо ее модификацию.

Обратите внимание на хэш-таблицы — это наиболее важная тема. Она часто встречается на собеседованиях.

Таблица степеней двойки

Некоторые люди помнят ее как таблицу умножения, если вы не относитесь к их числу, вам нужно подготовиться. Таблица степеней двойки пригодится в задачах масштабируемости — в расчетах необходимого объема памяти для набора данных.

Степень 2	Точное значение (X)	Приближенное значение	X байтов в мегабайте, гигабайте и т. д.
7	128		
8	256		
10	1024	1 тысяча	1 Кбайт
16	65 536		64 Кбайт
20	1 048 536	1 миллион	1 Мбайт
30	1 073 741 824	1 миллиард	1 Гбайт
32	4 294 967 296		4 Гбайт
40	1 099 511 627 776	1 триллион	1 Тбайт

Используя эту таблицу, вы можете легко рассчитать, например, хватит ли имеющегося объема памяти для хэш-таблицы, отображающей каждое 32-битное число в булевое значение.

Если вы проходите телефонное собеседование в веб-ориентированной компании, полезно держать эту таблицу перед глазами.

Нужно ли знать все о программировании на C++, Java или других языках

Хотя я лично никогда не любила вопросы такого рода (например: «Что такое vtable?»), многие интервьюеры действительно их задают. В крупных компаниях — Microsoft, Google, Amazon и других — таким вопросам не уделяется слишком много внимания. Вы должны понимать основные концепции языка, которым вы владеете, но лучше сфокусироваться на структурах данных и алгоритмах.

[1] Про «О-большое» можно прочитать в Википедии http://ru.wikipedia.org/wiki/«О»_большое_и_«о»_малое. — *Примеч. ред.*

В то же время для небольших компаний вопросы, связанные с языками программирования, могут быть очень важны. Поищите компанию, в которой собираетесь проходить собеседование, на CareerCup.com. Если вашей компании там нет, поищите аналогичную компанию. Большинство стартапов проверяют навыки именно «их» языка программирования.

Ответы на технические вопросы

Собеседование не может быть простым. Если вам не удается дать ответ «с ходу» — это нормально! Только десять из более чем ста двадцати человек смогли быстро разобраться в моих любимых задачах.

Если вам достался сложный вопрос, не нужно паниковать. Начните с планирования решения — покажите интервьюеру, что вы не застряли.

Запомните еще одно правило — вы не должны останавливаться, пока интервьюер не скажет, что решение закончено. Что я имею в виду? Если вы уже придумали алгоритм, продолжите рассуждать о возможных ошибках в его работе. Если вы пишете программный код, попытайтесь найти в нем ошибки. Если вы относитесь к числу тех самых 110 кандидатов, у вас, вероятно, будут ошибки.

Пять шагов к решению

Любую техническую задачу на собеседовании можно решить за пять шагов:

1. Задайте интервьюеру вопросы, чтобы снять неоднозначности.
2. Разработайте алгоритм.
3. Запишите псевдокод, но сообщите интервьюеру, что вы намерены написать решение на конкретном языке программирования.
4. В умеренном темпе начните писать программный код.
5. Проверьте написанную программу и внимательно исправьте ошибки.

Остановимся на каждом шаге поподробнее.

Шаг 1. Задайте вопросы

Технические задачи более неоднозначны, чем может показаться на первый взгляд, убедитесь, что вам ясна суть вопроса. Задача может оказаться намного проще, чем казалась в начале. Некоторые интервьюеры (в частности, в Microsoft) специально проверяют, поняли ли вы задачу.

Вот несколько примеров хороших вопросов: какие типы данных используются? Какие объемы данных будут использоваться? Какие исходные предположения нужны для решения? Кто будет пользователем?

Задача: **«Разработайте алгоритм сортировки списка»**

Вопрос: Какой список нужно сортировать? Массив? Связный список?

Ответ: Массив.

Вопрос: Что будет в массиве? Числа? Символы? Строки?

Ответ: Числа.

Вопрос: Числа будут целыми?

Ответ: Да.

Вопрос: Что представляют собой эти числа? Это идентификаторы? Значение чего-либо?

Ответ: Это возраст клиентов.

Вопрос: Сколько будет клиентов?

Ответ: Миллион.

Теперь постановка задачи изменилась: нужно отсортировать массив из миллиона целых чисел в диапазоне от 0 до 130 (максимально возможный возраст). Как ее решить? Достаточно создать массив из 130 элементов и посчитать количество значений для каждого возраста.

Шаг 2. Разработайте алгоритм

Разработайте алгоритм, но при этом помните о пяти подходах алгоритмизации (см. следующий раздел). Пока вы обдумываете алгоритм, не забудьте ответить на вопросы:

- Сколько памяти и времени понадобится для реализации этого алгоритма?
- Что произойдет, если данных будет больше, чем запланировано?
- Какие проблемы могут возникнуть в процессе работы вашего алгоритма? Например, если вы создаете модификацию бинарного дерева поиска, как ваш алгоритм повлияет на время вставки, поиска и удаления?
- Какие компромиссные решения возможны с учетом существующих ограничений? Для каких сценариев компромиссное решение будет наименее оптимальным?
- Нужна ли дополнительная исходная информация (в нашем случае — возраст клиентов или порядок сортировки) для реализации алгоритма? Обычно интервьюер специально предоставляет вам дополнительную информацию.

Метод грубой силы — вполне приемлемый и даже рекомендованный путь. После его разработки вы можете провести оптимизацию. Рано или поздно вы придете к оптимальному решению, но это не означает, что оно будет первым пришедшим в голову.

Шаг 3. Псевдокод

Псевдокод поможет в общих чертах набросать ваши соображения и избежать большей части ошибок. Но не забудьте сообщить интервьюеру, что сейчас вы пишете псевдокод, а затем перейдете к языку программирования. Много кандидатов используют псевдокод, чтобы избежать написания программы, но вы же не хотите быть одним из них?

Шаг 4. Код

Не нужно торопиться, чтобы потом не было мучительно больно. Пишите программный код спокойно и аккуратно. Запомните советы:

- Используйте структуры данных везде, где это возможно; выберите подходящую структуру данных или разработайте собственную. Например, если вас просят определить минимальный возраст группы людей, можно создать специальную структуру данных — `Person`. Этим вы покажете интервьюеру, что способны создать хороший объектно-ориентированный проект.
- Не создавайте очень длинные программы. Когда вы записываете свой код на доске, начинайте с верхнего левого угла, а не с середины, чтобы ответ поместился целиком.

Шаг 5. Проверка

Вам нужно протестировать код. В первую очередь обратите внимание на следующие моменты:

- экстремальные случаи — `0`, отрицательное значение, `null`, максимумы, минимумы;
- ошибки пользователя — что случится, если пользователь передаст `null` или отрицательное значение;
- общие случаи — протестируйте типовое поведение программы.

Если алгоритм сложный или исключительно численный (смещение, булева алгебра и т. д.), тестируйте код по мере написания, а не по завершении работы.

Если вы находите ошибки (а они будут), не торопитесь их исправлять — попытайтесь найти причину их возникновения. Вы же не хотите стать одним из кандидатов, который, обнаруживая, что функция возвращает `true` вместо `false`, просто инвертирует ее значение. Это устранит ошибку в конкретном случае, но неизбежно породит новые.

Работая над ошибками, задумайтесь, почему код перестал работать. Это поможет написать красивый и чистый код намного быстрее.

Пять подходов к алгоритмизации

Не существует универсального суперспособа, решающего любую задачу алгоритмизации, но приведенные далее подходы могут оказаться полезны. Помните: чем больше задач вы решите, тем проще будет выбрать подходящий способ.

Пять способов, приведенных далее, можно использовать по отдельности или вместе. Попробуйте подход «Упростить и обобщить», а затем перейдите к «Сопоставлению с образцом».

Подход 1. Приводим пример

Мы начнем со способа, с которым вы, вероятно, знакомы, даже если никогда не обращали на него внимания. Напишите несколько примеров задачи, и вы увидите, можно ли получить общее правило из них.

Пример: задано время, нужно рассчитать угол между часовой и минутной стрелками.

Давайте начнем, пусть исходное время — `3:27`. Мы можем нарисовать циферблат, установить часовую стрелку на 3 часа, а минутную — на 27 минут.

Введем следующие обозначения: `h` — это часы, `m` — минуты. Предположим, что `h` может принимать значения в диапазоне от `0` до `23` включительно.

Теперь можно сформулировать следующие правила:

- угол между минутной стрелкой и «полуднем»: `360 * m / 60`;
- угол между часовой стрелкой и «полуднем»: `360 * (h % 12) / 12 + 360 * (m / 60) * (1 / 12)`;
- угол между часовой стрелкой и минутной: `(угол часовой стрелки – угол минутной стрелки) % 360`.

Окончательное выражение можно упростить до `30h – 5.5m`.

Подход 2. Сопоставление с образцом

«Сопоставление с образцом» подразумевает, что мы сравниваем задачу с подобной и пытаемся приспособить алгоритм для нашего случая.

Пример. Отсортированный массив был циклически сдвинут так, что элементы оказались в следующем порядке: 3 4 5 6 7 1 2. Как найти минимальный элемент? Вы можете исходить только из предположения, что элементы в массиве не повторяются.

Существуют две задачи, которые можно рассматривать как аналог:

- Поиск минимального элемента в массиве.
- Поиск определенного элемента в сортированном массиве (бинарный поиск).

Поиск минимального элемента в массиве не самый интересный алгоритм — нужно пройтись по всем элементам. Вряд ли он будет полезен.

А вот бинарный поиск можно использовать. Вы знаете, что массив был отсортирован, а затем циклически сдвинут. Поэтому значения увеличиваются, а затем сбрасываются в исходную точку и снова растут. Минимальный элемент оказывается в точке сброса.

Если вы сравните средний и последний элементы (6 и 2), то узнаете, что точка сброса должна находиться между этими значениями (`MID > RIGHT`). Но это может произойти, только если массив сбрасывался между данными значениями.

Если `MID` меньше, чем `RIGHT`, значит, точка сброса находится в левой части массива, либо точки сброса не существует (массив отсортирован). Так или иначе, минимальный элемент находится там.

Мы можем использовать этот подход, деля массив пополам, подобно алгоритму бинарного поиска, и, в конечном счете, найдем минимальный элемент (или точку сброса).

Подход 3. Упростить и обобщить

Этот алгоритм подразумевает многошаговый подход. Во-первых, мы изменяем ограничение (тип данных или количество исходных данных), чтобы упростить задачу. Затем мы решаем упрощенную версию задачи. Как только алгоритм для упрощенной задачи получен, мы обобщаем ее и пытаемся приспособить полученное решение к более сложной версии.

Пример. Требование о выкупе было склеено из вырезанных из газеты отдельных слов. Как проверить, что требование о выкупе (представленное в виде строки) было сделано с использованием конкретной газеты (строки)?

Для упрощения задачи предположим, что из газеты вырезались отдельные буквы, а не слова.

Чтобы решить упрощенную задачу, создадим массив и подсчитаем символы. Каждый элемент массива соответствует одной букве. Сначала мы подсчитываем, сколько раз повторяется каждый знак в требовании о выкупе, а затем проверяем, есть ли в газете все эти символы.

При обобщении алгоритма используется тот же подход. Только вместо создания массива с количеством символов можно создать хэш-таблицу, которая сопоставляет слова с частотой их использования.

Подход 4. Базовый случай и сборка решения

Данный метод идеально подходит для решения целого ряда задач. Мы можем решить задачу для базового случая (`n = 1`). Это обычно означает запись корректного результата. Затем решить задачу для `n = 2`, учитывая, что ответ для `n = 1` уже найден. Затем заняться случаем `n = 3`, учитывая, что ответы для `n = 1` и `n = 2` известны.

В итоге мы можем построить решение, которое позволит найти результат для `N`, если известен правильный результат для `N – 1`. Каждый раз наше решение основывается на предыдущем результате.

Пример. Разработайте алгоритм для вывода всех возможных перестановок символов в строке (считайте, что все символы используются только один раз).

Дана тестовая строка — `abcdefg`.

```
Случай "a" --> {"a"}
Случай "ab" --> {"ab", "ba"}
Случай "abc" --> ?
```

Вот и первый «интересный» случай. Можем ли мы сгенерировать `P("abc")`, если у нас есть ответ `P("ab")`? Итак, у нас появляется дополнительная буква («c») и нам нужно вставить ее во все возможные позиции.

```
P("abc") = вставить "c" во все позиции для всех строк P("ab")
P("abc") = вставить "c" во все позиции для всех строк {"ab","ba"}
P("abc") = соединить ({"cab", "acb", "abc"}, {"cba", "bca", bac"})
P("abc") = {"cab", "acb", "abc", "cba", "bca", bac"}
```

Мы разобрались с шаблоном и можем разработать общий рекурсивный алгоритм. Сгенерируйте все перестановки строки $s_1...s_n$, удалив последний символ (для $s_1...s_{n-1}$). Получив список всех перестановок $s_1...s_{n-1}$, последовательно пройдитесь по каждому элементу-строке списка, добавляя символ s_n в каждую позицию строки.

Данный подход позволяет создавать рекурсивные алгоритмы.

Подход 5. Мозговой штурм структур данных

Данный способ трудно назвать идеальным, но зачастую он срабатывает. Мы просто проходим по списку структур данных и пытаемся использовать каждую из них. Данный подход может оказаться полезным, поскольку в некоторых случаях решение задачи, что называется, «появится на поверхности», как только будет выбрана правильная структура данных (например, дерево).

Пример. Была сгенерирована и сохранена в массив последовательность случайных чисел. Как найти медиану?

Пытаемся устроить мозговой штурм и подобрать адекватную структуру данных:

- Связный список? Вероятно, нет. Связные списки плохо подходят для сортировки.
- Массив? Может быть, но у нас уже есть массив. Как хранить в нем отсортированные элементы? Это довольно сложно. Давайте отложим эту структуру данных и вернемся к ней при необходимости.
- Бинарное дерево? Вполне возможно, бинарные деревья подходят для задач сортировки. Мы можем попробовать усовершенствовать дерево бинарного поиска, а вершина будет медианой. Но будьте осторожны: число элементов может оказаться четным, а медиана окажется между двумя средними элементами. Два средних

элемента не могут оказаться на вершине одновременно. Возможно, этот алгоритм подойдет, мы вернемся к нему позже.

- Куча? Куча — отличный способ для сортировки и отслеживания максимальных и минимальных значений. Это интересный выбор. Если у вас будет две кучи, можно следить за «большими» и «меньшими» частями элементов. «Большая» половина находится в min-куче, так что самый маленький элемент оказывается в вершине, а «меньшая» половина — в max-куче, так что в вершине — наибольший элемент. Такие структуры данных позволяют вам найти потенциальные медианы. Если размер куч изменился, можно повторно провести балансировку, выталкивая элементы из одной кучи в другую.

Обратите внимание, что множество задач легко решаются, если правильно выбрать структуры для используемых данных. Теперь вам нужно понять, какой подход более применим к той или иной задаче.

Как выглядит хороший код

Вы, возможно, неоднократно слышали, что работодатели хотят видеть «хороший» и «чистый» код. Но что это такое и как продемонстрировать его на собеседовании?

О хорошем коде всегда можно сказать, что он:

- **правильный:** код корректно обрабатывает все корректные и некорректные входные данные;
- **эффективный:** код максимально эффективен с точки зрения времени и пространства (памяти). Эффективность включает асимптотическую («О-большое») и практическую (реальную) эффективность. При расчете зависимости времени выполнения программы от ее сложности постоянный коэффициент можно отбросить, но в реальной жизни он может оказать влияние на эффективность;
- **простой:** если вы можете реализовать алгоритм в десяти строках, не пишите сто. Создайте код максимально быстро;
- **читаемый:** другой разработчик должен прочитать ваш код и понять, что и как делается. Для читаемого кода необходимы комментарии, они делают код более понятным. Сдвига строк недостаточно;
- **обслуживаемый:** код должен легко адаптироваться к изменениям, которые возникают на протяжении жизненного цикла продукта, он должен обслуживаться другими программистами так же легко, как и разработчиком.

Следование всем этим правилам требует определенных компромиссов. Например, стоит принести в жертву немного эффективности, но сделать код более удобным в сопровождении и наоборот.

Вы должны думать обо всех этих аспектах, когда пишете программный код на собеседовании.

Структуры данных

Предположим, что вас попросили написать функцию, выполняющую сложение двух простых полиномов, представленных в виде $Ax^a + Bx^b + \ldots$. Интервьюер не хочет, чтобы вы делали парсинг строк, ему нужно, чтобы вы использовали структуру данных, способную хранить полином.

Есть много разных способов решить такую задачу.

Неудачная реализация

Плохая реализация подразумевает хранение полинома в виде массива чисел с двойной точностью, где k-й элемент соответствует элементу x^k. Такая структура может стать причиной ошибок при необходимости представить полином, содержащий отрицательные или дробные степени. Кроме того, для хранения полинома, содержащего только один член x^{1000}, потребуется массив из 1000 элементов.

```
1 int[] sum(double[] poly1, double[] poly2) {
2     ...
3 }
```

Чуть лучшая реализация

Чуть лучшая реализация подразумевает хранение полинома в двух массивах — `coefficients` и `exponents`. Полином в этом случае может храниться в любом порядке, но i-й член полинома должен иметь вид `coefficients[i]` * $x^{\texttt{exponents[i]}}$.

Таким образом, если `coefficients[p] = k` и `exponents[p] = m`, то p-й член будет равен kx^m.

Хотя данная реализация не имеет таких ограничений, как предыдущее решение, она все еще далека от идеала. Мы должны использовать пару массивов для каждого полинома. Если массивы будут разной длины, то в полиноме оказываются «неопределенные» значения. Да и результат будет не очень удобным, так как при вызове функция будет возвращать два массива.

```
1 ??? sum(double[] coeffs1, double[] expon1,
2         double[] coeffs2, double[] expon2) {
3     ...
4 }
```

Хорошая реализация

Хорошая реализация — разработка собственной структуры данных для хранения полинома.

```
1 class PolyTerm {
2     double coefficient;
3     double exponent;
4 }
5
6 PolyTerm[] sum(PolyTerm[] poly1, PolyTerm[] poly) {
7     ...
8 }
```

Кое-кто будет утверждать, что это «сверхоптимизация». Возможно, да, а возможно — нет. Независимо от вашего мнения, это решение продемонстрирует, что вы думаете о коде и не пытаетесь решить задачу самым простым (и самым быстрым) способом.

Обоснованное многократное использование кода

Предположим, что вас попросили написать функцию, проверяющую на равенство двоичное и шестнадцатеричное представление числа, хранимое в виде строки. Изящная реализация этой задачи подразумевает повторное использование кода:

```
1 public boolean compareBinToHex(String binary, String hex) {
2     int n1 = convertToBase(binary, 2);
```

```
3     int n2 = convertToBase(hex, 16);
4     if (n1 < 0 || n2 < 0) {
5         return false;
6     } else {
7         return n1 == n2;

8     }
9 }
10
11 public int digitToValue(char c) {
12     if (c >= '0' && c <= '9') return c - '0';
13     else if (c >= 'A' && c <= 'F') return 10 + c - 'A';
14     else if (c >= 'a' && c <= 'f') return 10 + c - 'a';
15     return -1;
16 }
17
18 public int convertToBase(String number, int base) {
19     if (base < 2 || (base > 10 && base != 16)) return -1;
20     int value = 0;
21     for (int i = number.length() - 1; i >= 0; i--) {
22         int digit = digitToValue(number.charAt(i));
23         if (digit < 0 || digit >= base) {
24             return -1;
25             }
26         int exp = number.length() - 1 - i;
27         value += digit * Math.pow(base, exp);
28     }
29     return value;
30 }
```

Возможно, вы смогли бы написать отдельный код, преобразующий двоичное число в шестнадцатеричное, но это сделает нашу программу более громоздкой и тяжелой в обслуживании. Поэтому мы используем код повторно, вызывая общие методы `convertToBase` и `digitToValue`.

Модульность

Написание модульного кода подразумевает выделение изолированных блоков программы в отдельные методы. Это делает код более удобным, читаемым и тестируемым.

Допустим, что вам нужно написать код, меняющий местами минимальный и максимальный элементы в целочисленном массиве. Эту задачу можно реализовать в одном методе.

```
1 public void swapMinMax(int[] array) {
2     int minIndex = 0;
3     for (int i = 1; i < array.length; i++) {
4         if (array[i] < array[minIndex]) {
5             minIndex = i;
6         }
7     }
8
```

продолжение ⇩