
Оглавление

Предисловие	11
Введение	12
Что-то пошло не так.....	12
Мой подход.....	13
Моя страсть.....	13
Часть I. Процесс собеседования.....	14
Почему?.....	15
Как выбираются вопросы	17
Часто задаваемые вопросы.....	18
Часть II. За кулисами	19
Microsoft.....	20
Amazon.....	21
Google.....	22
Apple	23
Facebook	24
Palantir	25
Часть III. Нестандартные случаи	27
Профessionал.....	27
Тестеры и SDET	27
Менеджеры программ и менеджеры продукта.....	28
Ведущие разработчики и менеджеры	30
Стартапы.....	31
Для интервьюеров	32
Часть IV. Перед собеседованием.....	37
Получаем «правильный» опыт	37
Идеальное резюме	38
Часть V. Подготовка к поведенческим вопросам	41
Поведенческие вопросы	41
Ответы на поведенческие вопросы.....	43

Часть VI. «О» большое	47
Аналогия	47
Временная сложность	47
Пространственная сложность	49
Константы	50
Исключение второстепенных факторов	51
Составные алгоритмы: сложение и умножение	52
Амортизированное время	52
Сложность Log N	53
Сложность рекурсивных алгоритмов	54
Часть VII. Технические вопросы	56
Как организовать подготовку	56
Что нужно знать	56
Процесс решения задачи	58
Метод оптимизации 1: поиск BUD	63
Метод оптимизации 2: интуитивный подход	66
Метод оптимизации 3: упрощение и обобщение	66
Метод оптимизации 4: базовый случай и расширение	67
Метод оптимизации 5: мозговой штурм структур данных	67
Неправильные ответы	68
Если вы уже знаете ответ	69
«Идеальный» язык для собеседований	69
Как выглядит хороший код	70
Не сдавайтесь!	71
Часть VIII. После собеседования	72
Реакция на предложение и на отказ	72
Предложение работы	73
Переговоры	75
На работе	76
Часть IX. Вопросы собеседования	78
1. Массивы и строки	79
Хеш-таблицы	79
ArrayList и динамические массивы	80
StringBuilder	81
Вопросы собеседования	82

2. Связные списки.....	84
Создание связного списка	84
Удаление узла из односвязного списка.....	85
Метод бегунка.....	85
Рекурсия и связные списки	86
Вопросы собеседования	86
3. Стеки и очереди	88
Реализация стека	88
Реализация очереди.....	89
Вопросы собеседования	90
4. Деревья и графы	92
Разновидности деревьев.....	92
Бинарные деревья и бинарные деревья поиска	93
Обход бинарного дерева.....	95
Бинарные кучи (min-кучи и max-кучи).....	96
Нагруженные (префиксные) деревья.....	97
Графы	98
Список смежности	98
Поиск в графе.....	100
Вопросы интервью	102
5. Операции с битами	105
Расчеты на бумаге.....	105
Биты: трюки и факты	106
Поразрядное дополнение и отрицательные числа.....	106
Арифметический и логический сдвиг	106
Основные операции: получение и установка бита.....	107
Вопросы собеседования	109
6. Головоломки.....	111
Простые числа	111
Вероятность	113
Начинайте говорить.....	115
Правила и шаблоны	115
Балансировка худшего случая	116
Алгоритмический подход	117
Вопросы собеседования	117

8 Оглавление

7. Объектно-ориентированное проектирование	120
Как подходить к решению заданий	120
Паттерны проектирования	121
Вопросы собеседования	122
8. Рекурсия и динамическое программирование	125
С чего начать	125
Решения рекурсивные и итеративные	126
Динамическое программирование и мемоизация	126
Вопросы собеседования	130
9. Масштабируемость и проектирование систем	133
Работа с вопросами	133
Проектирование: шаг за шагом	134
Масштабируемые алгоритмы: шаг за шагом	136
Ключевые концепции	137
Дополнительные факторы	140
Идеальных систем не бывает	140
Пример: найдите все документы, содержащие список слов	141
Вопросы собеседования	143
10. Сортировка и поиск	145
Распространенные алгоритмы сортировки	145
Алгоритмы поиска	148
Вопросы собеседования	149
11. Тестирование	152
Чего ожидает интервьюер	152
Тестирование реального объекта	153
Тестирование программного обеспечения	154
Тестирование функций	156
Поиск и устранение неисправностей	157
Вопросы собеседования	158
12. С и С++	159
Классы и наследование	159
Конструкторы и деструкторы	160
Виртуальные функции	160
Виртуальный деструктор	161
Значения по умолчанию	162
Перегрузка операторов	163

Указатели и ссылки.....	163
Шаблоны	164
Вопросы собеседования	165
13. Java	167
Подход к изучению	167
Перегрузка vs переопределение	167
Java Collection Framework	168
Вопросы собеседования	169
14. Базы данных	171
Синтаксис SQL и его варианты	171
Денормализованные и нормализованные базы данных.....	171
Команды SQL.....	172
Проектирование небольшой базы данных.....	174
Проектирование больших баз данных	175
Вопросы собеседования	175
15. Потоки и блокировки	177
Потоки в Java	177
Синхронизация и блокировки	179
Взаимные блокировки и их предотвращение.....	182
Вопросы собеседования	183
16. Задачи умеренной сложности.....	185
17. Сложные задачи.....	190
Часть X. Решения	196
1. Массивы и строки	197
2. Связные списки.....	214
3. Стеки и очереди	234
4. Деревья и графы	248
5. Операции с битами	286
6. Головоломки.....	299
7. Объектно-ориентированное проектирование	317
8. Рекурсия и динамическое программирование	355
9. Масштабируемость и проектирование систем	387

10 Оглавление

10. Сортировка и поиск.....	415
11. Тестирование.....	437
12. С и C++	443
13. Java	456
14. Базы данных	465
15. Потоки и блокировки	472
16. Задачи умеренной сложности.....	488
17. Сложные задачи.....	560

Часть XI. Дополнительные материалы **665**

Полезные формулы.....	666
Топологическая сортировка.....	668
Алгоритм Дейкстры.....	669
Разрешение коллизий в хеш-таблице.....	672
Поиск подстроки по алгоритму Рабина – Карпа.....	673
AVL-деревья.....	674
Красно-черные деревья	676
MapReduce	680
Дополнительные материалы.....	681

Часть XII. Библиотека кода **683**

HashMapList<T, E>	683
TreeNode (бинарное дерево поиска)	684
LinkedListNode (связный список)	685
Trie и TrieNode	686

Часть XIII. Подсказки(скажите с сайта издательства www.piter.com) **689**

1. Структуры данных.....	690
2. Концепции и алгоритмы.....	699
3. Вопросы, основанные на знаниях.....	713
4. Дополнительные задачи	716

<http://goo.gl/ssQdRk>

8

Рекурсия и динамическое программирование

Существует огромное множество разнообразных рекурсивных задач, но большинство из них строится по похожим схемам. Подсказка: если задача рекурсивна, то она может быть разбита на подзадачи.

Совет: мой опыт обучения показывает, что интуиция «Да это похоже на рекурсивную задачу» срабатывает примерно с 50%-ной точностью. Используйте этот инстинкт, это очень полезные 50%. Но не бойтесь взглянуть на задачу под другим углом, даже если на первых порах она показалась вам рекурсивной. Существует 50%-ная вероятность того, что первое впечатление было ошибочным.

Когда вы получаете задание, начинающееся со слов «Разработайте алгоритм для вычисления N -го...», или «Напишите код для вывода первых n ...», или «Реализуйте метод для вычисления всех...» — скорее всего, речь пойдет о рекурсии.

С чего начать

Рекурсивные решения по определению основываются на решении подзадач. Очень часто вам приходится вычислять $f(n)$, добавляя, вычитая или еще как-либо изменяя решение для $f(n-1)$. В других случаях задача может решаться для первой половины набора данных, а затем для второй половины с последующим слиянием результатов.

Существует много способов деления задачи на подзадачи. Три самых распространенных метода разработки алгоритма — восходящий, нисходящий и половинчатый.

Восходящая рекурсия

Восходящая рекурсия обычно более понятна на интуитивном уровне. Вы знаете, как решить задачу для самого простого случая — например, для списка всего с одним элементом. Затем задача решается для двух элементов, затем для трех и т. д. Подумайте о том, как построить решение для конкретного случая, основываясь на решении для предыдущего случая (или нескольких предыдущих случаев).

Нисходящая рекурсия

Нисходящая рекурсия выглядит более сложной, так как она менее конкретна. Тем не менее в отдельных случаях такой подход к решению задачи оказывается оптимальным.

В этом случае необходимо решить, как разделить задачу для случая N на подзадачи. Будьте осторожны с перекрывающимися случаями.

Половинчатая рекурсия

Помимо восходящей и нисходящей рекурсии также часто эффективно работает метод разбиения набора данных на две половины.

Например, алгоритм бинарной сортировки основан на «половинчатом» методе. При поиске элемента в отсортированном массиве вы сначала определяете, какая половина массива содержит искомое значение. Затем происходит рекурсивный переход, и поиск продолжается в выбранной половине.

Алгоритм сортировки слиянием также относится к «половинчатым» методам. Каждая половина массива сортируется по отдельности, после чего отсортированные половины объединяются.

Решения рекурсивные и итеративные

Рекурсивные алгоритмы могут быть весьма неэффективны по затратам памяти. Каждый рекурсивный вызов добавляет новый уровень в стек; это означает, что если алгоритм проводит рекурсию до уровня n , он использует как минимум $O(n)$ памяти.

По этой причине часто бывает лучше реализовать рекурсивный алгоритм в итеративном виде. Все рекурсивные алгоритмы могут быть реализованы в итеративном виде, хотя иногда это приводит к существенному усложнению кода. Прежде чем браться за рекурсивный код, спросите себя, насколько сложно будет реализовать его в итеративном виде, и обсудите достоинства и недостатки каждого варианта с интервьюером.

Динамическое программирование и мемоизация

Среди разработчиков распространено мнение о сложности задач динамического программирования, однако бояться их не стоит. Собственно, когда вы усвоите суть метода, такие задачи становятся очень простыми.

Методология динамического программирования в основном сводится к определению рекурсивного алгоритма и нахождению перекрывающихся подзадач. Полученные результаты кэшируются для будущих рекурсивных вызовов.

При другом подходе следует проанализировать закономерность рекурсивных вызовов и использовать итеративную реализацию. При этом по-прежнему возможно «кэширование» предшествующей работы.

Замечание по поводу терминологии: некоторые специалисты называют нисходящее динамическое программирование «мемоизацией» (memoization), а термин «динамическое программирование» используют только для восходящих методов. Мы не будем различать эти случаи, и используем термин «динамическое программирование».

Один из простейших примеров динамического программирования — вычисление n -го числа Фибоначчи. Хороший подход к задачам такого рода часто заключается в том, чтобы реализовать задачу в обычном рекурсивном виде, а затем добавить в решение кэширование.

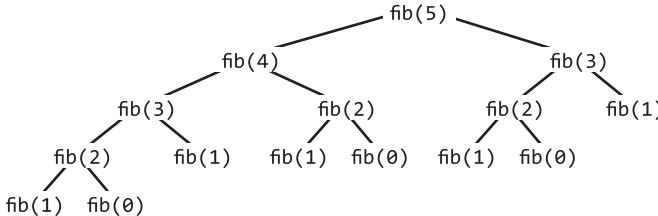
Числа Фибоначчи

Рекурсия

Начнем с рекурсивной реализации. Просто, не так ли?

```
1 int fibonacci(int i) {
2     if (i == 0) return 0;
3     if (i == 1) return 1;
4     return fibonacci(i - 1) + fibonacci(i - 2);
5 }
```

Каково время выполнения этой функции? Немного подумайте, прежде чем отвечать. Если вы ответили $O(n)$ или $O(n^2)$ (как отвечает большинство людей), подумайте снова. Проанализируйте последовательность выполнения кода. Попробуйте нарисовать ветви выполнения в виде дерева (то есть нарисовать дерево рекурсии) — это полезно в этой и многих других рекурсивных задачах.



Заметьте, что на всех листьях дерева находятся вызовы `fib(1)` и `fib(0)`. Они могут рассматриваться как базовые случаи.

Общее количество узлов в дереве определяет время выполнения, так как за пределами своих рекурсивных вызовов каждый вызов выполняет работу $O(1)$. Следовательно, время выполнения определяется количеством вызовов.

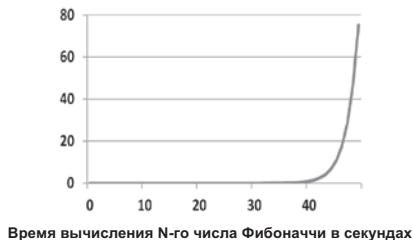
Совет: запомните это обстоятельство для будущих задач. Представление рекурсивных вызовов в виде дерева — отличный способ оценки времени выполнения рекурсивного алгоритма.

Сколько узлов в этом дереве? Пока мы добираемся до базовых случаев (листьев), каждый узел содержит два дочерних узла. Иначе говоря, в каждом узле порождаются две ветви.

Корневой узел имеет два дочерних узла. У каждого из них два своих дочерних узла (таким образом, на уровне «внуков» расположены четыре узла). У каждого из них два своих дочерних узла, и т. д. Если повторить это рассуждение n раз, количество узлов будет приблизительно равно $O(2^n)$.

На самом деле оно немного лучше $O(2^n)$. Взглянув на поддерево, вы можете заметить, что правое поддерево любого узла всегда меньше левого поддерева (кроме листовых узлов и узлов, находящихся непосредственно над ними). Если бы они имели одинаковые размеры, то время выполнения было бы равно $O(2^n)$. Но поскольку размеры правого и левого поддеревьев различны, истинное время выполнения близко к $O(1.6^n)$. Впрочем, запись $O(2^n)$ формально верна, так как она описывает верхнюю границу времени выполнения (см. « O , Θ и Ω » на с. 48). В любом случае, достигается экспоненциальное время выполнения.

В самом деле, если реализовать этот алгоритм на компьютере, вы заметите, что количество секунд возрастает экспоненциально.



Нужно поискать возможность оптимизации.

Нисходящее динамическое программирование (или мемоизация)

Проанализируем время рекурсии. Есть ли в дереве идентичные узлы?

Идентичных узлов много. Например, вызов `fib(3)` встречается дважды, а вызов `fib(2)` встречается трижды. Зачем каждый раз вычислять результаты с нуля?

В действительности при вызове `fib(n)` объем работы не должен заметно превышать $O(n)$ вызовов, поскольку существуют всего $O(n)$ возможных значений, которые могут передаваться `fib`. Каждый раз, когда мы вычисляем `fib(i)`, результат следует просто кэшировать и использовать его в будущем.

Именно в этом и заключается суть мемоизации.

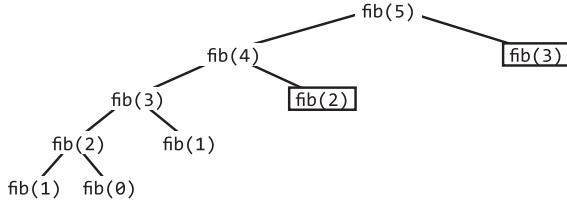
Всего одно небольшое изменение — и время, необходимое на выполнение этой функции, уменьшится до $O(n)$. Нужно всего лишь кэшировать результаты функции `fibonacci(i)` между вызовами:

```

1 int fibonacci(int n) {
2     return fibonacci(n, new int[n + 1]);
3 }
4
5 int fibonacci(int i, int[] memo) {
6     if (i == 0 || i == 1) return i;
7
8     if (memo[i] == 0) {
9         memo[i] = fibonacci(i - 1, memo) + fibonacci(i - 2, memo);
10    }
11    return memo[i];
12 }
```

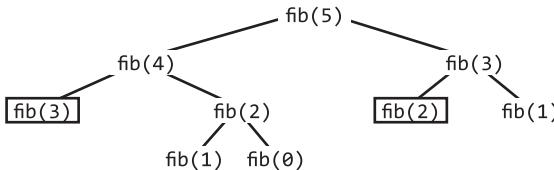
Генерирование 50-го числа Фибоначчи на стандартном компьютере с помощью рекурсивной функции займет около минуты. Метод динамического программирования позволит сгенерировать 10 000-е число Фибоначчи за доли миллисекунды (но не забывайте, что размера типа `int` может оказаться недостаточно).

Если нарисовать дерево рекурсии, оно выглядит примерно так (прямоугольники изображают кэшированные вызовы, которые немедленно возвращают управление):



Сколько узлов теперь содержит дерево? Можно заметить, что дерево теперь уходит в глубину приблизительно n . Каждый из этих узлов также имеет один дочерний узел, так что в результате дерево содержит приблизительно $2n$ дочерних узлов. Таким образом, время выполнения составляет $O(n)$.

Часто бывает полезно представить дерево рекурсии в следующем виде:



На самом деле рекурсия происходит *не так*. Однако с расширением узлов, находящихся на более высоком уровне, вместо нижних узлов создается дерево, которое разрастается в первую очередь в ширину (а не в глубину). Иногда такое представление упрощает вычисление количества узлов в дереве. Здесь мы всего лишь изменяем то, какие узлы раскрываются, а какие возвращают кэшируемые значения. Попробуйте применить этот прием, если у вас возникнут трудности при вычислении времени выполнения задачи динамического программирования.

Восходящее динамическое программирование

Мы также можем взять этот метод и реализовать его средствами восходящего динамического программирования. Считайте, что происходит то же самое, что и при рекурсивном подходе с мемоизацией, но в обратном порядке.

Сначала мы вычисляем $\text{fib}(1)$ и $\text{fib}(0)$ — случаи, которые, как нам уже известно, являются базовыми. Затем эти результаты используются для вычисления $\text{fib}(2)$. После этого предыдущие ответы используются для вычисления $\text{fib}(3)$, $\text{fib}(4)$ и т. д.

```

1 int fibonacci(int n) {
2     if (n == 0) return 0;
3     else if (n == 1) return 1;
4
5     int[] memo = new int[n];
6     memo[0] = 0;
7     memo[1] = 1;
8     for (int i = 2; i < n; i++) {
9         memo[i] = memo[i - 1] + memo[i - 2];
10    }
11    return memo[n - 1] + memo[n - 2];
12 }
  
```