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

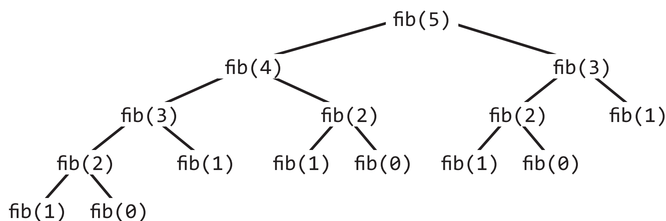
Рекурсия

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.

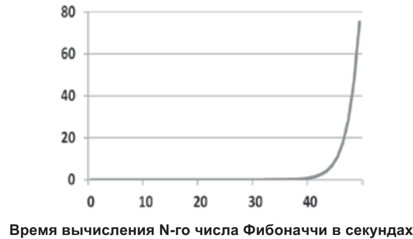
Совет: запомните это обстоятельство для будущих задач. Представление рекурсивных вызовов в виде дерева — отличный способ оценки времени выполнения рекурсивного алгоритма.

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.



Время вычисления N-го числа Фибоначчи в секундах

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.

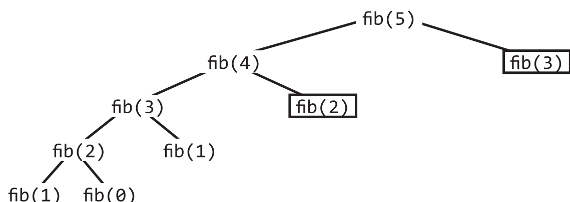
Именно в этом и заключается суть мемоизации.

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  }
```

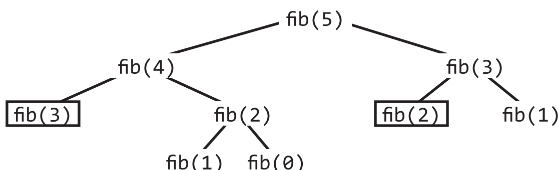
Генерирование 50-го числа Фибоначчи на стандартном компьютере с помощью рекурсивной функции займет около минуты. Метод динамического программирования позволит сгенерировать 10 000-е число Фибоначчи за доли миллисекунды (но не забывайте, что размера типа `int` может оказаться недостаточно).

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 `fib(1)` и `fib(0)` — случаи, которые, как нам уже известно, являются базовыми. Затем эти результаты используются для вычисления `fib(2)`. После этого предыдущие ответы используются для вычисления `fib(3)`, `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  }
```