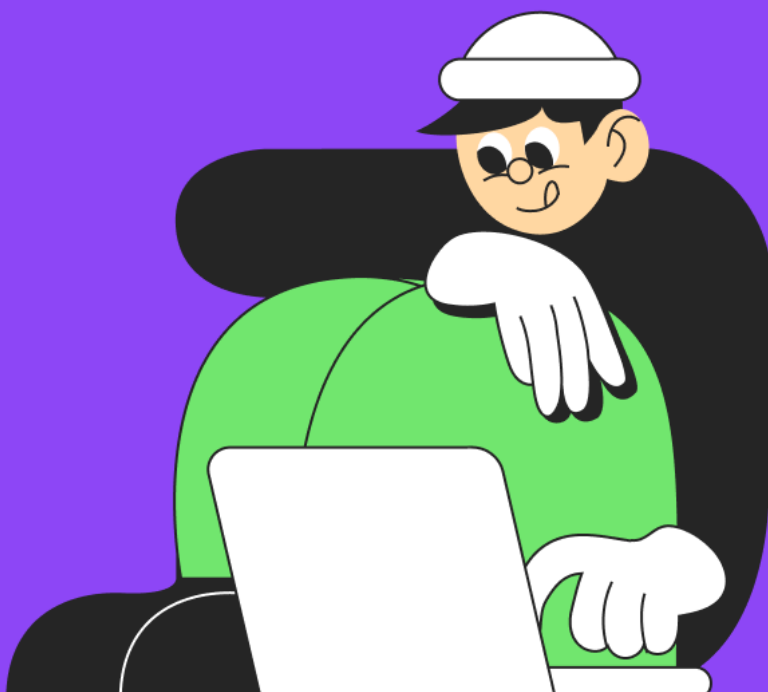
GeekBrains

Рекурсия: продолжение



Оглавление

[00.02.48]
Приветствие

Приветствую вас на очередной лекции по программированию.

Сегодня мы продолжим разбираться с тем, что же такое рекурсия. Разберём решение стандартных и нестандартных задач. Попробуем разобраться, почему рекурсия — это хорошо. И в то же время узнаем, почему же рекурсия — это плохо. Итак, начнём с основ.

Вспомним, что рекурсия — функция, которая вызывает саму себя абсолютно в любом виде. Это может быть, void-метод или void-функция, а также функция, возвращающая какое-то значение, не принципиально. Основная идея заключается в том, что в теле метода, в теле функции есть вызов её самой.

Далее разберём код. Будет очень много кода, причём код, который, скорее всего, покажется вам абсолютно незнакомым и непонятным. Но как я говорил в более ранних лекциях, в первую очередь важно и нужно привыкнуть к коду, чтобы примерно представлять общие механизмы решения.

Если думаете, что любой разработчик может решить любую задачу, вы ошибаетесь. Кто-то когда-то придумал красивое решение, и дальше это решение растеклось по книгам, а потом программисты прошлого передали это знание дальше.

Сейчас, во времена интернета, можете погуглить и найти любое решение, но всегда старайтесь разобраться, что вы нашли. Не надо копировать всё в тупую, используя ctrl+c и ctrl+v, а потом разбираться, для чего оно работает. Так делать, конечно, не надо. В то же время, если нашли какой-то код, разобрали его и поняли, как он работает, перед запуском собственной программы проговаривайте, какой результат ожидаете получить во время запуска.

[00.03.21]
Что важно при описании рекурсии

Итак, главный вопрос, который мне хочется разобрать — это что надо сделать в первую очередь, когда начинаем описывать рекурсию. Возьмите перерывчик, равный одной минуте, и подумайте, что самое важное в написании и описании рекурсивной функции.

Самое важное, друзья мои, при описании рекурсии — прописать условия, когда рекурсия будет заканчивать себя вызывать. Поэтому условие выхода считается одним из ключевых условий. Если у вас будет неконтролируемая рекурсия, это приведёт к следующему:

- к медленной работе всей системы;
- будет казаться, что ваша программа виснет;
- у вас будет много востребованных ресурсов, что нехорошо.

Немного позже посмотрим, как расходуются ресурсы, и так далее.

Теперь предлагаю рассмотреть некоторые примеры решения задач. Сначала посмотрим, как всё решается итеративным подходом, путём классических ветвлений и циклов, а дальше попытаемся решить те же задачи, используя рекурсию.

[00.05.22]
Собираем строку с числами от a до b, a≤b

Итак, первая задача. Для начала обратимся к нашему списку задач и попробуем собрать строку с числами от a до b. В условии оговорено, что число a будет меньше или равно b. В случае итеративного подхода всё предельно просто.

Запускаем цикл, который будет менять счётчик от значения a, меньшим или равным б. В результирующую строку будем собирать конкретное значение счётчика. В то же время, используя рекурсию, надо прописать условие окончания — в нашем случае оно указано в ветке else, когда возвращаем пустую строку, если условие не выполнилось. Соответственно, если a меньше или равно b, собираем строку с текущим значением a и вызываем новую копию метода со значениями аргументов. Первый увеличится на 1.

Дальше в качестве самостоятельной работы предлагаю додумать этот метод так, чтобы получилось на один вызов рекурсии меньше. Даю небольшую подсказку: надо немного подкрутить условия и ветку else прописать несколько иначе.

Как видите, ничего сложного здесь нет. В то же время, если нас попросят вывести число от большего к меньшему, то итеративно подход принципиально отличаться не будет — придётся немного поправить условия. Соответственно, потребуется делать вывод от большего к меньшему и производить не инкремент счётчика, а декремент. В то же время, используя рекурсию, нам достаточно поменять саму собирающую строку. То есть мы будем прибавлять не слева, как это было в предыдущем примере, а справа.

[00.06.58]
Сумма чисел от 1 до n

Следующая задача. Допустим, требуется найти сумму чисел от 1 до n. Самый простой способ — итеративный. Берём цикл от 1 до числа, которое указано, в нашем случае — n, и собираем всё в некую переменную result. Если использовать рекурсию, потребуется прописать условия выхода, а условие выхода здесь — n, равное нулю. В этом случае будем возвращать нейтральный по значению элемент. Либо можно сделать на один вызов меньше и прописать: если n равно 1, возвращается 1, а далее — остальное. Если число отлично от 0, возвращаем текущее значение n и рекурсивный вызов функции с аргументом на 1 меньше.

Хорошо, в этой задаче мы рассмотрели пример использования итеративного и рекурсивного подхода. Теперь вы, как математики, должны знать, как на самом деле решается эта задача. Тем, кто забыл, напоминаю, что у нас есть формула, позволяющая найти сумму чисел от 1 до n. Немного позже мы посмотрим, как быстро работает каждый подход.

[00.08.06]

Факториал числа

Следующая задача — факториал числа. В нашем случае факториалом числа называется произведение чисел от 1 до n. Соответственно, используя итеративный подход, мы описываем функцию, принимающую значение того самого n, факториал которого требуется найти. Дальше заводим некоторую результирующую переменную, по умолчанию будет нейтральный по умножению элемент — 1. Далее идёт цикл от 1 до момента, пока i меньше или равно n. Соответственно, будем домножать результат на текущее значение i, а после этого — возвращать результат.

Если у нас рекурсивный подход, обязательное условие выхода — то, что n, аргумент нашей функции, стал равен 1. В этом случае будем возвращать 1. Кстати, по-хорошему надо описать ещё и 0, потому что 0! тоже считается 1. И остальные условия, когда мы будем возвращать текущее значение n, умноженное на вызов рекурсивной функции с аргументом на 1 меньше. А результат будет одинаковым.

[00.09.03]

Вычисляем a^n

Итак, мы поскладывали числа, пособирали строки, поперемножали числа. Теперь попробуем возвести число a в натуральную степень n.

В общем случае даже эту задачу можно разобщить для целой степени. В чём идея? Число a в степени n по определению натуральной степени — это a*a*a и так далее n раз. Те, кто был на лекциях по математике, информатике, как минимум должен это помнить. Таким образом, если мы будем использовать такой подход, то простой итеративный метод будет выглядеть как перебор счётчика от 1 до n и домножения результата на текущее значение так называемого основания степени. А n — это показатель степени. В случае же рекурсивного подхода сразу опишем условия выхода — начнём читать код от «если n равно 0, то возвращаем единицу».

Не рассматриваем 0^0, как некую неопределённость. И договоримся, что 0^0 для нашего случая будет 1. В общем случае хорошо бы это рассмотреть отдельным вариантом.

Итак, если показатель степени равен 0, будем возвращать 1, иначе — запустим рекурсивный подсчёт, там текущее значение a будет домножаться на вызов рекурсивной функции, где в качестве аргумента передаём основания нашей степени и показатель на 1 меньше.

От себя отмечу, если хотите уменьшить число строк, то язык C# позволяет использовать так называемый тернарный оператор. И та конструкция, которая написана на двух строчках, записывается и одной строкой — return n = 0?1: и так далее. То есть можете это протестировать.

Решили мы задачу? В общем случае даже двумя способами. Вы были на лекциях по математике и информатике и помните различные формулы. Если не помните, подключим несколько флешбэков из лекции по математике.

[00.11.06]

Базовая математика, которая пригодится всем

Мы говорили о формулах сокращённого возведения в степень. Значит, в ситуации, при которой будет чётная степень, мы можем сразу же умножать число на само себя, а показатель степени уменьшать в два раза. Есть две формулы. Соответственно, применив их, мы можем сильно улучшить решение нашей задачи. Опять же, насколько оно станет лучше? Немного позже я покажу механизмы, позволяющие проверить, что такой подход действительно будет лучше. Но об этом чуть позже.

[00.11.37]
Перебор слов

Следующий пример немного сложнее. Пусть у нас будет некоторый алфавит, состоящий из четырёх букв, и нас просят показать все возможные слова, состоящие из T букв. T может равняться 1, 2, 3 и так далее.

Как решить эту задачу простым способом, итеративным? Итак, пусть у нас будет алфавит, который будет храниться в массиве символов. Далее возьмём один цикл и переберём все эти буквы. У нас получатся однобуквенные слова. Если потребуются двухбуквенные слова, мы уже должны будем использовать цикл в цикле. Соответственно, если нам потребуется использовать трёхбуквенные слова, придётся делать цикл в цикле, заключив всё в ещё один цикл. И так далее.

Можете ли вы решить эту задачу в общем виде, если наперёд неизвестно T? То есть сколько букв должно быть в словах? Даю подсказку: можете. Предлагаю сделать небольшой перерывчик на три минутки, чтобы вы подумали, каким образом это можно сделать.

[00.15.43]
Общее решение

Снова сделаю отсылку к лекциям по математике, информатике, где я рассказывал, как разбить некоторое количество участников на некоторое количество команд. Я приводил пример разбиения на две команды, используя двоичную систему счисления. Если букв четыре, попробуйте догадаться, что будет в этом случае: какая система счисления нам потребуется и как эту задачу можно решить итеративно. То есть в определённой системе счисления надо перебрать числа от и до.

Но если вы плохо себя чувствовали на лекциях по математике и информатике, мы можем прибегнуть к помощи рекурсии. Идея:

- у нас будет метод, который примет тот самый алфавит — передадим его просто строкой;
- далее будет массив из букв, которые составят новое слово;
- затем на текущей итерации вызова рекурсии будет собираться длина самого этого слова.

Описываем условия выхода из рекурсии. То есть если длина нашего массива как раз таки совпала с текущей длиной, которую получили на текущем вызове рекурсии, тогда будем просто показывать получившееся слово и на этом заканчивать. В противном случае потребуется запустить цикл по всем элементам нашего алфавита, чтобы собрать очередное слово. Попробуем убедиться в этом и посмотреть, что получится.

Итак, есть код. Дальше для определённости укажем массив из двух символов. То есть мы ожидаем получить все двухбуквенные слова. Запускаем и видим, что я принудительно хотел сделать счётчик этих самых слов, поэтому заводим переменную. То есть сам по себе подсчёт этих символов нам абсолютно не нужен. Но, чтобы быть уверенным, что их действительно 4^2, получаем 16 этих слов. Если у нас, например, будут трёхбуквенные слова, ожидаем получить 4^3. Соответственно, если будут четырёхбуквенные слова, получим 4^4. В общем, мы видим правильные результаты. Но здесь мы не привязаны к количеству циклов, которые надо прописывать вручную. Всё изначально заложено в самой рекурсии.

Лучший ли это способ? Думаю, можно придумать и что-то иное. Опять же, если кого-то смущает, что здесь void-метод, не стоит смущаться. Здесь как раз таки идея заключается в том, чтобы собрать все слова. Я это делаю путём вывода в консоль. В общем случае, когда будете знать немного больше, можно, например, использовать списки, которые будут аккумулировать в себе эти слова и дальше просто возвращать получившийся список. Либо можно это всё собирать в срок таким же образом. Но нужно ли нам это сейчас? То есть идея заключается в том, что метод рабочий: мы можем собрать слова, отвязавшись от конкретного количества указанных или описанных циклов.

[00.18.57]
Ещё рекурсия

Хорошо, мы рассмотрели математический пример использования рекурсии. Но, быть может, рекурсию можно применить где-то в жизни? В частности, я покажу небольшой пример, но нам потребуется так называемый vin api, чтобы работать с директориями. Сначала познакомимся с основными синтаксическими конструкциями и с функционалом, который для этого необходим. Затем перейдём к рекурсии. Итак, посмотрим, в чём заключается идея.

Покажу алгоритм обхода директории. Что это значит? Это значит, что мы хотим посмотреть все папки, лежащие внутри этой папки, и все файлы, находящиеся внутри папки. В свою очередь, каждая папка внутри папки служит хранилищем папок и файлов. Следующая папка будет снова в себе хранить файлы и папки, и так далее. То есть чтобы определить содержимое папки, надо посмотреть, что находится в следующих папках. Таким образом, здесь такая рекурсия просматривается очевидно.

Для начала посмотрим, что именно требуется, чтобы получить доступ к нужной директории. Чтобы работать с директориями, применяется класс DirectoryInfo. Мы должны указать путь к той папке, свойства которой хотим посмотреть. Воспользуемся, например, папочкой HelloCode. Далее скопируем путь и укажем string path. В нашем случае путь — это строка. Далее мы можем выцепить различную информацию, например, какой-нибудь CreationTime.

Для простоты убедимся, что это работает. Мы видим, что папка была создана 11 месяца 29 ноября. То есть какую-то информацию мы можем получить. Аналогичным образом можем

обратиться к di и посмотреть, какие внутри содержится файлы — GetFiles. Если навести курсор на третью строку, то увидим, что результат работы этого метода — массив типа FileInfo. То есть это как раз таки второй функционал, который нам потребуется, или класс, содержащий информацию о конкретном файле. Мы можем прописать код, а дальше, используя обычный цикл, пробежаться по всем элементам, указать конкретный элемент массива и вызвать для него свойство — например, Name.

Запускаем и ожидаем получить информацию о дате создания текущей папки и все файлы, находящиеся в ней. В нашем случае видим наименование файла — csproj, HelloUser, первый файлик и второй файлик. Соответственно, если создадим какой-нибудь ещё файл, для этого просто продублируем программ-файлик и перезапустим наш код, появится три файлика. Действительно, копия появилась. Далее я это убираю, чтобы код не сломался.

Таким образом, у нас есть тип FileInfo и тип DirectoryInfo, позволяющие получить информацию о соответствующих директориях либо о файлах.

[00.22.51]
Как смотреть содержимое папки

Теперь опишем рекурсию, которая как раз таки будет ходить по папкам и смотреть, что же там внутри.

За кулисами я написал код. Метод называется CatalogInfo. В качестве аргумента принимаем путь к текущей папке, а в качестве второго аргумента используем искусственный приём, позволяющий делать отступы, чтобы примерно видеть структуру папки. Далее получаем информацию о той директории, в которую зашли, по указанному пути. Затем получаем массив всех файлов, находящихся в этой папке, и пробегаем по ним, выводя информацию о текущем каталоге. Мы будем рекурсивно заныривать и, соответственно, смотреть все папки, которые получим на этом этапе. После того как закончим вывод папок, получим весь список файлов в текущей директории текущего каталога и покажем их.

Запустим их. В качестве аргумента пути укажем всю папку HelloCode, которую писали на первых лекциях. Для начала немного всё упростим. Пусть это будет первый пример. Скопируем путь. На простом примере посмотрим сначала, что это работает для одной папки. Далее убедимся, что это будет работать и для общего случая, для большой директории. Видим, что что-то пошло не так — неподходящий терминал.

Наблюдаем следующее:

- у нас есть основной каталог, внутри которого находится папка obj;
- внутри папки obj — папка Debug, где лежит папка net6.0;
- далее идёт ref, где находится файл.

Перейдём туда и убедимся в этом. У нас есть основной каталог, далее идёт папочка obj, внутри неё есть папка Debug. Далее идёт net6.0m, где есть папка ref, внутри которой лежит файлик. Искусственные пробелы дают примерную структуру. Если выйдем на верхнюю папку, то есть окажемся на одном уровне с папкой ref, обнаружим файл Example001_HelloConsole. В общем, здесь будет много разных файлов.

Далее, если перейдём, например, в папку bin, также обнаружится папочка Debug, внутри которой лежит net6.0, и так далее. То есть для одной папки работает. Теперь просто в качестве демки вернём большой путь. Запустим — здесь это тоже работает. Такой практический пример использования рекурсии.

Пришлось узнать немного больше:

- что такое CatalogInfo;
- что такое DirectoryInfo;
- для чего всё это нужно;
- как с этим работать.

Здесь есть документация, где можно почитать и узнать что-то новое, чтобы затем уже приниматься писать собственные алгоритмы.

[00.26.07]
Игра в пирамидки

Итак, рекурсия сначала нам помогла с математикой, затем позволила обойти все папки на компьютере. Теперь пришло время поиграть с рекурсией.

Есть замечательная детская игра «Ханойские башни». Суть её заключается в том, что у нас есть некоторая башенка с определённым количеством блинчиков и шпилей. Далее мы должны каким-то образом переместить нужный блин, например, на третий шпиль. Ура, мы победили. На следующем этапе можем взять, например, четыре блина, пять и так далее.

Количество этих блинчиков может быть большим. Чем больше блинов, тем дольше будет идти игра. И что самое важное, так проще запутаться. Натравим на эту игру рекурсию и решим задачу.

Как и в предыдущем примере, весь код я уже написал за кулисами. Сделаем шпиль рабочим и возьмём из него текущий блинчик. Вторым аргументом передадим шпиль, на котором должна оказаться пирамидка.

Далее дадим название нашему промежуточному шпилю, потому что всего их по умолчанию три, и укажем, какое количество блинов есть всего. Затем запустим метод и посмотрим, что конкретно потребуется выполнить для трёх пирамидок.

Итак, мы должны с 1 шпиля переместить блины на 3, а с 1 — на 2. Соответственно, с 1 — на 3, с 1 — на 2. Дальше с 3 — на 2, с 1 — на 3. С 3 — на 2, с 1 — на 3, далее со 2 — на 1, со 2 — на 3. Со 2 — на 1, со 2 — на 3. И далее с 1 — на 3. Как видите, наш алгоритм отработал.

Аналогичным образом мы можем взять, например, четыре шпиля, указать в качестве аргумента 4 и запустить — количество действий не на много увеличится. Если у нас будет, например, пять блинов, то количество действий окажется очень большим. И если решать эту задачу вручную, этот процесс окажется долгим, на каждом этапе будет возникать возможность запутаться, сделать что-то не то. Поэтому придётся делать лишние переставления, но в случае рекурсии этот вариант — беспроигрышный. Три строчки кода дают

нам решения для трёх шпилей и любого количества блинов. Поэтому используйте этот способ, если он вам удобен.

[00.28.58]
Обход разных структур — 1

Хорошо. Где ещё можно воспользоваться рекурсией, чтобы упростить себе жизнь?

Рекурсия часто применяется и в более сложных задачах. Для нашего курса они считаются сложными. Но всё-таки одну простую задачу я предлагаю рассмотреть. Опять же, задача проста в рамках своей теории, но для нас она по-прежнему сложная.

Посмотрим, в чём заключается идея. Допустим, у нас есть некое арифметическое выражение. Вы никогда об этом, скорее всего, не задумывались, но, оказывается, что арифметическое выражение состоит из арифметических выражений. Для логических выражений, кстати, примерно то же самое. В чём идея?

Если рассмотрим, казалось бы, простую запись, станет понятно, что её можно вычислить сразу. Но если мы подставим какие-то a, b, c, d, то есть общие выражения, в этом случае придётся решать задачу в общем виде. Таким образом, если рассмотреть текущую запись, окажется, что значок деления — не что иное, как применение к арифметическому выражению, которое, с одной стороны, выступает как большая скобка, а с другой — как арифметическое выражение, состоящее из одного числа.

Дальше, если посмотреть на выражение, указанное на слайде, окажется, что оно состоит из двух других выражений. Соответственно, каждое из этих выражений будет также состоять из отдельных цифр.

[00.30.16]
Обход разных структур — 2

Далее мы можем на основе разбора выражения построить то, что в дискретной математике в теории графов называется бинарным деревом. Идея в том, что в качестве отдельных операций или чисел, с которыми будем работать, мы представляем узлы дерева. Затем их можно пронумеровать. Но нумеровать мы будем не абы как, а по определённому правилу.

В нашем случае первая операция будет под номером 1. Умножению присвоим 2, а десятке — 3. Дальше по порядку: 4 — для минуса, 5 — для плюса. Но если бы вместо 10 была какая-нибудь операция, то технически под ней тоже появились два каких-то операнда. Для них мы продолжили бы нумерацию, присвоив им 6 и 7. Но у нас их нет. Поэтому со следующей строки уже начинаем нумерацию с 8 и 9, 10 и 11, соответственно.

На следующем этапе можно заметить следующее: если определить так называемую иерархию подчинённости, элемент, располагающийся выше, будет родительским по отношению к элементу, стоящему ниже. И наоборот: нижний элемент будет дочерним по отношению к верхнему. То есть в нашем случае, например, 1 и 3 считаются дочерними узлами по отношению к плюсику. А плюсик считается дочерним элементом по отношению к операции умножения и родительским для 1 и 3. Примерная иерархия подчинённости.

Обратите внимание, если мы возьмём в качестве одного узла i элемент, то его дочерние компоненты, или дочерние узлы, будут с индексами 2i и 2i+1. То есть 2 — это i элемент. Соответственно, 2i — это 4, а 2i+1 — 5. Это будет работать для абсолютно любого количества таких узлов. 5 2i — это 10, 2i+1 — это 11.

Дальше встаёт вопрос, а можно ли такое дерево представить на компьютере, используя примитивные конструкции. Оказывается, можно. И такой конструкцией в нашем случае служит обычный одномерный массив. То есть если под индексом 1 будет храниться операция деления, то дочерними компонентами 2 и 3 будет операция умножения и десятка. Далее для умножения у нас будет 2i, то есть это 4 и 5. Видим минус и плюс.

Далее мы можем описать логику обхода этого дерева, а именно: получить все значения узлов, которые в нём содержатся. Для чего вы будете их получать — уже другая задача:

- возможно, вы хотите посчитать это выражение;
- красиво его распечатать;
- или сделать какую-то условную задачу — например, добавить то, чего не хватает, определить, какой узел будет с условным знаком вопроса, и так далее.

[00.33.19]
Обход разных структур — 3

Теперь рассмотрим код, который будет решать эту задачу. Итак, у нас есть одномерный массив, представляющий собой дерево. emp мы используем, чтобы вся строчка помещалась без горизонтального скролла. Дальше описываем метод, позволяющий делать обход. Соответственно, в качестве аргумента указываем ту позицию, с которой будем начинать этот обход, и делаем проверку. То есть если наша позиция вылетела за количество элементов, хранящихся в нашем дереве, на этом наша рекурсия закончится. Это условие продолжения, когда позиция превзошла количество узлов в нашем дереве.

Далее считаем позицию левого поддерева. То есть если посмотреть, например, операцию деления, то обнаружим у неё левое поддерево и правое. У операции умножения также есть левая и правая части. Левая часть находится на позиции 2i, или 2 pos в моём случае, а правая — на 2i+1.

Затем делаем проверку. Если есть левое поддерево, мы не вылетаем за границы нашего дерева, и элемент, который там хранится, не считается пустым, например, ситуация 6 и 7 здесь обозначена специально, надо рекурсивно запустить обход дерева с текущей позиции.

Далее выводим узел, в нашем случае будет значение конкретной операции или числа, и аналогичным образом делаем для правого поддерева. Если правое поддерево существует, и элемент, в который мы попали, непустой, надо рекурсивно запустить обход.

Запустим этот код и посмотрим, что получится. В нашем случае будет просто обход нужных нам узлов. Итак, видим 4, минус, 2 и так далее.

[00.35.21]

Статья о способах обхода деревьев

Оставлю вам статейку о том, какие бывают способы обхода деревьев. В нашем случае есть прямой, центрированный и обратный обход. Они примерно одинаковые. Как я говорил немного ранее, в рекурсии мы можем просто поменять две строчки местами, и по факту, это будет уже абсолютно другой обход. Потренируйтесь на практике и посмотрите, в каком случае InOrder становится PreOrder, и так далее.

[00.35.50]

Обход разных структур: про деревья, про обходы, книги по алгоритмам из рекомендованного списка, пример

Итак, мы рассмотрели реальные практические задачи. Теперь разберёмся, где ещё могут применяться деревья. Не будем глубоко вникать в эту тему. Но если кому-то интересно посмотреть, как обходить деревья, представлять ими что-то, например, в большом продакшене, оставлю ссылки.

Следующий пункт — использование деревьев в жизни. В частности, любая страничка, которую вы открываете, имеет HTML-представление.

Соответственно, HTML-код, который вы видите, как раз представляет собой дерево. Поэтому при помощи этих же алгоритмов можно ходить по представлению сайта и, например, делать какой-нибудь парсер этого сайта. Например, если хотите «подсмотреть» какие-то товары из какого-нибудь маркета и аккуратно пробросить их в собственный маркет.

Вы знаете о существовании файлов Markdown, но есть более структурированные файлы, использующиеся, например, для передачи информации от клиента к серверу, если речь идёт о клиент-серверных технологиях. Такими файлами считаются файлы JSON и XML. Можете погуглить, для чего они используются, как выглядят и так далее. У нас это часть курса, поэтому останавливаться на них не будем.

Разбор всевозможных выражений. Я показал вам простой пример использования обхода арифметического выражения. В общем случае это могут быть, как я уже говорил, логические выражения и ещё какие-то структуры.

Далее — анализ текста. Когда пишете какой-то код, например, на языке C# или Python, компилятор каким-то образом превращает написанное в то, что каким-то образом работает. И здесь как раз таки используются синтаксические и семантические анализаторы. Всё это работает посредством деревьев, графов и рекурсии.

Следующий пункт — это общая теория графов, обход графов. Я показал очень маленькую часть. То есть идея заключается в том, что есть дискретная математика и теория графов. В общей теории графов есть теория деревьев. И одним из примеров деревьев считаются те самые бинарные деревья, о которых мы говорили. Конечно, это слишком простая задача, но хотя бы так.

[00.38.11]

Время подсчёта

Мы рассмотрели рекурсию, узнали, какая она хорошая, классная и не только. Теперь рассмотрим обратную сторону этой рекурсии и то, какие проблемы могут возникнуть при её использовании.

Первая и самая простая проблема — долгое время подсчёта. Посмотрим, как будет работать подсчёт чисел Фибоначчи рекурсивным способом и классическим итеративным способом.

Здесь я уже привожу пример кода, но с небольшими модификациями. У меня есть две переменные, одна из которых будет использоваться для подсчёта вызова рекурсии, рекурсивного метода, а вторая — считать количество итераций цикла для итеративного подхода. Вызовем демонстрацию для каждого числа.

Будем показывать не от 1 до 40, а от 10 до 40. В чём идея? В цикле для начала я запускаю итеративный подсчёт чисел Фибоначчи. Сделаем более красивую маску — мы сильно вылетим за пределы, и будет горизонтальная прокрутка, зато наглядно. Поставлю минус, чтобы стало чуть красивее. Укажем 15, чтобы числа были друг под другом. Такую же маску организуем для рекурсивного вывода. Итак, я текстом показываю значение n, для какого числа мы считаем, само значение и то, сколько итераций цикла произошло.

После того как посчитали очередное число, обнуляем fIte и на новой итерации этого цикла, то есть для следующего числа Фибоначчи, будем иметь новое представление этого значения. Аналогичным образом делаем всё для рекурсивного подхода, только здесь станет выводиться своя переменная и составляться будет она же. Запустим и посмотрим — числа от 10 до 39 показались практически моментально.

При использовании рекурсии вы можете наблюдать следующее: чтобы посчитать 10 число Фибоначчи, потребовалось 177 вызовов рекурсивного метода. Когда мы считаем число 11, их уже 287. А когда считаем число 38, сразу получается 126 491 971 вызов, и так далее. Как видите, у нас сильно меняется время. Засечём время и посмотрим, как ещё можно установить время выполнения кода. Запомним время до момента начала подсчёта и выведем на экран разницу между временем после введения и подсчёта всех чисел и временем, которое мы запомнили.

Аналогичным образом проделаем расчёт для рекурсивного подхода. Console.ReadLine используется, чтобы после запуска можно было сделать консоль больше.

Итак, посчитав 29 чисел, получилось 45 миллисекунд. Соответственно, чтобы посчитать те же числа, используя рекурсивный подход, серверам приходится страдать. И если вашим сервисом пользуются миллионы людей, а код написан, мягко говоря, не оптимально, то, с одной стороны, пользователь ждёт, а, с другой — приходится ещё оплачивать это впустую потраченное на работу серверов время. Как видите, для той же задачи потребовалось почти 20 секунд. В одной секунде — 1 000 миллисекунд.

Таким образом, это одна из обратных сторон рекурсии, когда всё начинает работать плохо. Хотя в качестве практической задачи предлагаю написать метод вычисления чисел Фибоначчи рекурсивно, но, чтобы он работал в несколько раз быстрее. Попробуйте.

[00.43.29]
Обратная сторона медали — ошибочки

Итак, мы рассмотрели вариант использования рекурсии и её медленную работу. А какие ещё могут возникнуть ошибки?

Часто встречающаяся ошибка при использовании рекурсий — это переполнение стека. Когда запускаете рекурсивную функцию, помните, что память не бесконечная, как и количество заныриваний в эту самую рекурсию. Бывает, что вы пишете код, забываете прописать условия выхода, переполняете стек вызовов, и ваша программа попросту вылетает.

Сымитируем такую ситуацию. Я напишу очень простой метод, чисто академический, который не имеет никакой практической значимости. Но в то же время такой метод явно показывает, как сломать свою программу, и то, что это не бесконечный цикл.

Итак, возьмём метод Rec, который будет вызывать сам себя. Далее наблюдаем вызов этого метода. Соответственно, запускаем код и смотрим, во что же это превратится. Видим какое-то ожидание и после этого наблюдаем stack overflow. То есть рекурсия залетела в себя столько раз, что память закончилась. Этот стек закончился, и наша программа дальше работать не может.

С точки зрения наглядных пособий, ещё можно прописать int=0 и посмотреть, что получится. Итак, видим Dotnet run. Может быть, придётся долго ждать, но долго ждать не пришлось. Получилось примерно то же самое. После определённого значения I вышла ошибка. Здесь, к сожалению, увидеть ошибку не получится, всё происходит слишком быстро. Попробуем ещё раз — просто видим stack overflow.

Иногда показать ошибку не получается, консоль тоже не вмещает бесконечное количество строчек. Но ничего страшного. Основная идея заключается в следующем: когда пишете код или рекурсию, где не получается, или забыли описать условия выхода, то попросту кладёте свою программу, переполняется стейк, и на этом всё заканчивается. Поэтому будьте аккуратны при описании этой рекурсии.

[00.46.17]
Ошибочки: OutMemory, StackOverflow, Slowly

Рассмотрим последнюю демку, где код, который до текущего момента казался хорошим и даже работал, на самом деле оказался не таким хорошим. Разберём пример из предыдущей лекции по рекурсии, где мы делали закрашивание замкнутой области.

Я подготовил некую картинку — просто квадратик. Он будет выглядеть как прямоугольник, но это квадратик. Далее делаем обрамление единичками и печатаем его. Методом закрашивания сделаем закрашивание внутренней области. Итак, изначально он был не закрашенным, а после этого стал закрашенным. Напоминаю, что это квадратик, хотя вы видите прямоугольник.

Теперь немного повеселимся и в качестве картинки возьмём значение не 10x10, а 1000x1000, для современных экранов это небольшая картинка. Дальше по понятным причинам я не буду

пытаться делать распечатывание этой картинки, потому что в консоль это не влезет. Однако запустим систему и посмотрим, к чему это всё приведёт.

Снова вышел stack overflow. Почему это случилось? Дело в том, что код, закрашивающий определённую область, для каждой точки нашей картинки будет вызывать четыре вызова рекурсии. Получается много избыточных элементов — избыточного обхода точек. Мы просматриваем много ненужных точек, которые даже нет смысла просматривать.

Можно ли как-то оптимизировать и улучшить код? Да, можно. Но тогда этот код уже будет составлять не четыре строчки, а гораздо больше, поэтому станет не таким интересным.

[00.46.17]
Заключение

Итак, надеюсь, вы открыли для себя что-то новое:

- узнали о рекурсии и о том, как и где её применять, надо ли это делать;
- что надо контролировать постоянно;
- сколько раз вызывается рекурсия;
- что надо прописывать условия окончания рекурсии.

На этом я с вами прощаюсь. Всего доброго и до встречи на семинарах.