В задании 16 требуется вычислить значение функции, заданной рекуррентным выражением. В этом выражении формула для вычисления следующего элемента последовательности зависит от некоторых условий, которым должен удовлетворять аргумент функции.

Демоверсия-2024:

Алгоритм вычисления значения функции F(n), где n – натуральное число, задан следующими соотношениями:
F(n) = n при n > 2024;
F(n) = n × F(n + 1), если n ≤ 2024.
Чему равно значение выражения F(2022) / F(2024)?

Демоверсия-2020:

Ниже на пяти языках программирования записан рекурсивный алгоритм F.

Бейсик

```
SUB F(N)
  PRINT(N)
  IF N >= 3 THEN
    F(N / 2)
    F(N - 1)
  END IF
END SUB
```

Python

```
def F(n):
  print(n, end='')
  if n >= 3:
    F(n // 2)
    F(n - 1)
```

Алгоритмический язык

```
алг F(цел n)
нач
  вывод n
  если n >= 3 то
    F(div(n, 2))
    F(n - 1)
  все
кон
```

Паскаль

```
procedure F(n: integer);
begin
  write(n);
  if n >= 3 then
  begin
    F(n div 2)
    F(n - 1)
  end
end;
```

C++

```
void F(int n){
  std::cout << n;
  if(n >= 3){
    F(n / 2);
    F(n - 1);
  }
}
```

Запишите подряд без пробелов и разделителей все числа, которые будут выведены на экран при выполнении вызова F(5). Числа должны быть записаны в том же порядке, в котором они выводятся на экран.

Можно заметить, что задания достаточно похожи, хоть запись выражения и представлена в разных формах. То есть от сдающего требуют проанализировать рекуррентное выражение (рекурсивную функцию) и найти его значение при определенных параметрах.

Также можно сказать, что задание в текущем его виде предполагает большую математическую строгость. Так как, например, вычисление факториала для 2022 достаточно сложная задача даже для компьютера (операции с длинными числами выполняются тем дольше, чем больше их длина). Поэтому можно сделать вывод, что ФИПИ хочет, чтобы подобные задания решались аналитически.

Что ж, желание ФИПИ оставим ФИПИ. И разберемся с наиболее популярными методами решения таких заданий.

Примеры решения заданий

Пример ручного решения задачи из демоверсии-2024

Заметим, что с каждым шагом рекурсии значение параметра увеличивается. Поэтому начнем преобразовывать выражение с числителя представленной дроби.

F(2022) / F(2024) = 2022·F(2023) / F(2024) = 2022·2023·F(2024) / F(2024)

На втором шаге в числителе и знаменателе заметим вызов F(2024), которые можно сократить по правилам математики.

Следовательно, F(2022) / F(2024) = 2022·F(2023) / F(2024) = 2022·2023·F(2024) / F(2024) = 2022·2023

Ответ: 4090506

Пример программного решения из демоверсии-2024

<table>
<tr><th>Python</th><th>PascalABC.net</th></tr>
<tr><td>

```python
def F(n):
  if n >= 2024:
    return n
  else:
    return n*F(n+1)

print(F(2022) / F(2024))
```

</td><td>

```pascal
###
function F(n: BigInteger): BigInteger;
begin
  if n >= 2024 then
    F := n
  else
    F := n*F(n+1)
end;

print(F(2022) / F(2024))
```

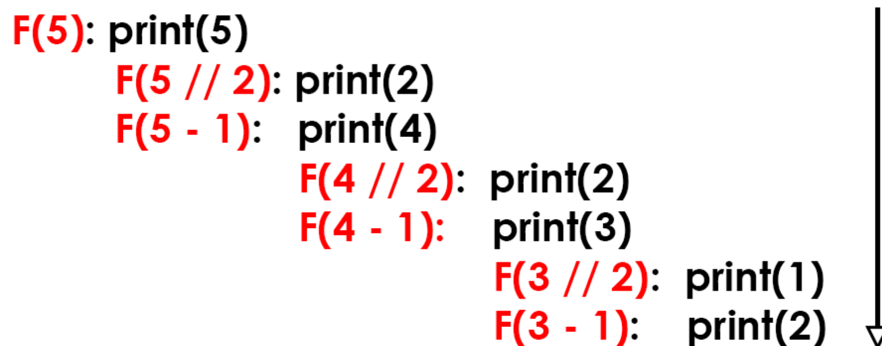
</td></tr>
</table>

Ответ: 4090506

Пример ручного решения из демоверсии-2020

При вызове F(5) алгоритм сначала выведет значение **5**, после чего выполнит вызов F(5 // 2) = F(2). Для F(2) выведет **2** и передаст управление вверх (продолжит выполнять F(5)). Выполнится вызов F(4) – будет выведено **4** и вызвана функция F(4 // 2) = F(2), которая выведет **2** и передаст управление обратно F(4). Вызывается F(3), в ходе выполнения которой печатается **3** и вызывается F(3//2) = F(1). Печатается **1** и вызывается F(3-1) = F(2) – печатается **2** и управление передается на уровень выше, после чего завершается выполнение F(3), далее F(4) и F(5).

Также данное рассуждение можно представить в виде изображения



Ответ: 5242312

Пример программного решения из демоверсии-2020

Просто перепишем код из задания и запустим его.

```
def F(n):
  print(n, end='')
  if n >= 3:
    F(n // 2)
    F(n - 1)
print(F(5))
>> 5242312
```

Как можно заметить, ничего сложного. Однако, могут быть постановки, где придется подумать больше. Таковые были и на экзамене. Поэтому рассмотрим методы детальнее и попробуем максимально обезопасить себя от ловушек на экзамене.

Также становится понятно, почему ФИПИ ушел от постановки задачи в виде программы – при такой постановке задача сдающего сводится к переписыванию кода и его запуску. Однако, существует ряд задач, которые при подобной постановке предполагают либо существенное изменение кода, либо вообще аналитическое решение (из-за невозможности в приемлемое время ответить на вопрос задачи).

Аналитический подход

В первую очередь необходимо ввести несколько определений.

Выход из рекурсии – определение значения рекурсивной функции (рекуррентного выражения) без повторного обращения к этой самой функции. В задании из демоверсии-2024 это было выражение F(n) = n.

Глубина рекурсии – максимальное количество вычисляемых в определенный момент времени значений функции или максимальное количество вложенных вызовов рекурсивной функции. Так, например, при вызове F(2022) из условия демо-2024 нужно было вычислить F(2023) для этого вычислить F(2024), значение которой уже определялось без следующего шага рекурсии. Таким образом глубина рекурсии была 3 (пока не определилось значение F(2024), нельзя было определить значение F(2023), а без него нельзя было определить значение F(2022)).

Шаг рекурсии (рекурсивный вызов) – вложенный вызов функцией самой себя. Так запись F(n) = n · F(n+1) осуществляет шаг рекурсии. Принято считать, что рекурсия «шагает» вглубь. Далее мы ответим на вопрос почему.

Важно отметить, что если в описании рекурсивной функции существует несколько обращений к ней самой, то совершается несколько шагов. Однако все они совершаются с одной глубины рекурсии. Так как возврат значения после вызова первой вложенной функции возвращает нас на глубину, с которой осуществлялся вложенный рекурсивный вызов (в некотором роде происходит шаг назад).

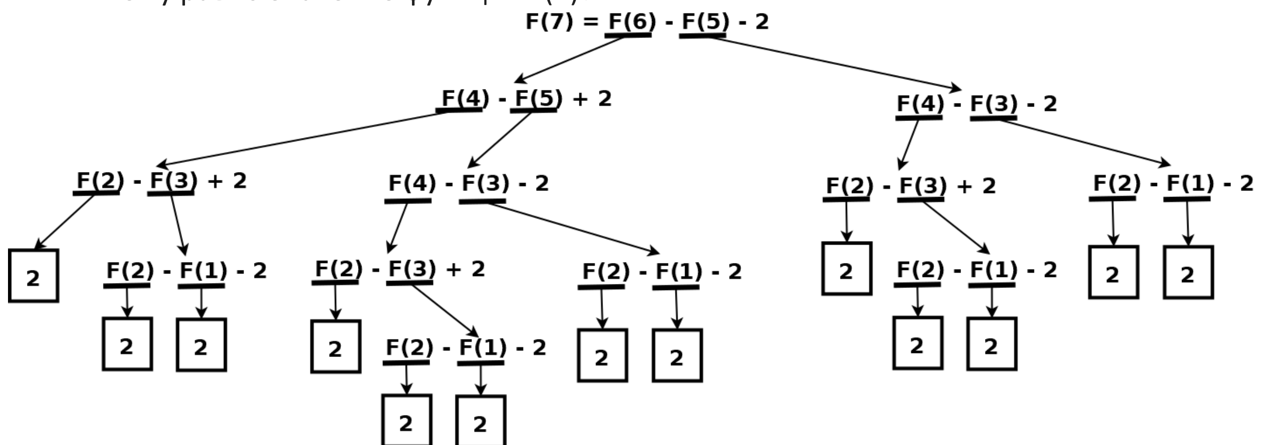
Например, при вызове F(5) из задачи в демоверсии-2020 сначала осуществлялся шаг рекурсии для F(5//2), после завершения F(5 // 2), контроль выполнения возвращался на уровень для F(5) и потом делался еще один шаг рекурсии для F(5-1). То есть шага было два, но оба из них делались с глубины 1.

Методы ручного решения

Вычисление рекуррентных выражений традиционно изображают в виде дерева. Для более наглядной иллюстрации рассмотрим построение дерева на условии задачи из варианта с апробации-2023, изменив значение параметра (чтобы не делать иллюстрацию огромной).

Апробация 10.03.23 с измененным вопросом

Алгоритм вычисления значения функции $F(n)$, где n — натуральное число, задан следующими соотношениями:
$F(n)=2$ при $n<3$;
$F(n)=F(n-2)-F(n-1)+2$, если $n>2$ и при этом n чётно;
$F(n)=F(n-1)-F(n-2)-2$, если $n>2$ и при этом n нечётно;
Чему равно значение функции $F(7)$?



В дереве каждый новый уровень – вложенный вызов функции (шаг рекурсии). Максимальное количество шагов, совершаемых в ходе подсчета равно 6 (количество уровней в дереве). Числа 2 в рамке – шаги с выходом из рекурсии (условие $n < 3$).

При работе с подобными деревьями принято глубиной называть количество «уровней», отсюда и берет начало традиция называть количество выполнняемых вложенных вызовов рекурсии глубиной. Иногда встречается термин «высота» дерева, однако «высота рекурсии» лично мне не встречалась никогда.

Для вычисления результата работы рекурсивной функции (значения рекуррентного выражения) нет необходимости строить полное дерево вычислений. Можно пойти и другим путем: начать восстанавливать значения с условия выхода из рекурсии.

Так для решения вышеупомянутого задания будем находить в порядке возрастания n значения F(n).

F(1) = 2
F(2) = 2
F(3) = F(2) – F(1) – 2 = 2 – 2 – 2 = -2
F(4) = F(2) – F(3) + 2 = 2 – (-2) + 2 = 6
F(5) = F(4) – F(3) – 2 = 6 – (-2) – 2 = 6
F(6) = F(4) – F(5) + 2 = 6 – 6 + 2 = 2
F(7) = F(6) – F(5) – 2 = 2 – 6 – 2 = -6

Гораздо быстрее и удобнее. И еще этот метод мы будем использовать, когда дойдем до разбора программных методов и кеширования (там это наблюдение применяется для оптимизации).

Взаимная/косвенная рекурсия

Встречаются случаи, когда рассматриваемая функция F не вызывает свою копию явно. Однако при вызове другой функции (или функций), вызванной при выполнении F, происходит повторный вызов F.

Пример из демоверсии-2016.

Ниже на пяти языках программирования записаны две рекурсивные функции (процедуры): F и G

Бейсик

```
DECLARE SUB F(N)
DECLARE SUB G(N)
SUB F(N)
  IF N > 0 THEN G(N-1)
END SUB

SUB G(N)
  PRINT "*"
  IF N > 1 THEN F(N-3)
END SUB
```

Python

```
def F(n):
  if n > 0:
    G(n - 1)

def G(n):
  print("*")
  if n > 1:
    F(n - 3)
```

Алгоритмический язык

```
алг F(цел n)
нач
  если n > 0 то
    G(n - 1)
  все
кон

алг G(цел n)
нач
  вывод "*"
  если n > 1 то
    F(n - 3)
  все
кон
```

Паскаль

```
procedure F(n: integer); forward;
procedure G(n: integer); forward;

procedure F(n: integer);
begin
  if n > 0 then
    G(n - 1)
end;

procedure G(n: integer);
begin
  writeln('*');
  if n > 1  then
    F(n - 3)
end;
```

C++

```
void F(int n);
void G(int n);

void F(int n){
  if(n > 0)
    G(n - 1);
}
void G(int n){
  if(n > 1)
    F(n - 3);
}
```

Сколько символов «звёздочка» будет напечатано на экране при выполнении вызова F(11)?

Можно заметить, что как функция F, так и функция G не вызывают непосредственно сами себя. Однако при вызове F(n) для n > 2 вызывается G(n-1), которая в свою очередь вызывает F((n-1)-3).

Таким образом вызов F(n) может привести к повторному вызову функции F с другим значением аргумента. Мы наблюдаем косвенную рекурсию.

Решим данную задачу от условия выхода. Так же заметим, что для вычисления F(n) необходимо предварительно вычислить G(n-1). Для удобства и компактности записи представим рассуждение в виде таблицы, где в первой строке обозначим операции, которые выполняются при запуске функции, жирным шрифтом во второй выделим итоговое количество звездочек.

		-1	0	1	2	3	4	5	6	7	8	9	10	11
G					1* F(-1)	1* F(0)	1* F(1)	1* F(2)	1* F(3)	1* F(4)	1* F(5)	1* F(6)	1* F(7)	1* F(8)
	P	**1***	**1***	**1***	**1***	**1***	**2***	**2***	**2***	**2***	**3***	**3***	**3***	**4***
F				G(0)	G(1)	G(2)	G(3)	G(4)	G(5)	G(6)	G(7)	G(8)	G(9)	G(10)
	P	**0***	**0***	**1***	**1***	**1***	**1***	**2***	**2***	**2***	**2***	**3***	**3***	**3***

Ответ: 3

Обработка исключительных случаев

Бывают случаи, когда рекуррентное отношение в ходе вычисления не достигает условия выхода. Конечно, использование таких отношений в реальной жизни вызывает вопросы. Однако в рамках учебных задач рассмотрение класса задач с неопределимыми значениями вполне допустимо. На ЕГЭ таких задач пока не было.

Для решения таких задач нужно уметь проанализировать представленное рекуррентное отношение и определить, для каких значений оно вообще вычислимо.

Рассмотрим следующую задачу:

> Алгоритм вычисления функций $F(n)$, где n – натуральное число, задан следующими соотношениями:
> $F(n) = n + 1$ при $n<3$
> $F(n) = n + 2\cdot F(n+2)$ когда $n\geq 3$ и четно
> $F(n) = F(n-2) + (n - 2)$ когда $n\geq 3$ и нечетно.
> Найдите минимальное значение n, для которого значение $F(n)$ будет трехзначным?

Нетрудно заметить, что при вычислении рекуррентного отношения для четного числа значение аргумента будет бесконечно увеличиваться, так как при добавлении к любому четному числу 2 будем иметь результатом еще одно четное число.

$F(4) = 4 + 2\cdot F(6) = 4 + 2\cdot(6 + 2\cdot F(8)) = 16 + 4\cdot F(8) = 16 + 4\cdot(8 + 2\cdot F(10)) = 48 + 8\cdot F(10) = \ldots$

Так мы можем понять, что отношение вычислимо только для нечетных значений от 3 и для чисел 1, 2 (на n накладывается условие натуральности).

Будем находить значения F(n) до тех пор, пока не получим трехзначное число. В первой строке запишем перебираемые значения n, во второй – соответствующие значениям n значения F(n).

n	1	3	5	7	9	11	13	15	17	19	21
F(n)	2	3	6	11	18	27	38	51	66	83	102

Ответ: 102

Преобразование рекурсивных алгоритмов

Также одно и тоже отношение может описывать алгоритм для различных групп чисел по-разному. Например, четные числа будут обрабатываться одной веткой, а нечетные другой. При этом обе ветки будут вычислимы, в отличии от предыдущего примера.

Рассмотрим это на примере следующей задачи:

> Алгоритм вычисления функции F(n) задан следующими соотношениями:
> $F(n)=n\%3$, для $n<5$,
> $F(n)=2\cdot F(n-2)+n\cdot 2$, при четном значении $n>4$,
> $F(n)=F(n-2)+n\%4$, при нечетном значении $n>4$.
> Для какого наибольшего натурального значения n результат F(n) не превышает 1500?
> *Примечание:* % - оператор нахождения остатка от деления.

Заметим, что снова имеем дело с обработкой четных и нечетных чисел. Однако теперь обе ветки приводят к результату (значение аргумента сводится к условию выхода).

При решении такого задания неопытный решающий может найти наименьшее значение n, для которого значение F(n) будет больше 1500 и в качестве ответа взять предыдущее значение. Однако такое решение будет неверным, так как для четных и нечетных чисел значение функции растет неодинаково. Рассмотрим это на примере первых 10 четных и 10 нечетных чисел.

n	4	6	8	10	12	14	16	18	20	22
F(n)	1	14	44	108	240	508	1048	2132	4304	8652

n	3	5	7	9	11	13	15	17	19	21
F(n)	0	1	4	5	8	9	12	13	16	17

Заметим, что значения отношения для нечетных значений увеличиваются значительно медленнее. Следовательно, значение 17 при неаккуратном решении было бы ошибочным ответом. Также становится очевидно, что продолжать перебирать нечетные числа до значения 1500 - процесс достаточно время затратный.

Поэтому сделаем наблюдение об изменении значения для нечетных n. n % 4 для нечетных значений может принимать значение либо 1, либо 3. Причем эти значения будут чередоваться с шагом 2.

Поэтому можем сказать, что $F(n) = F(n-4) + 4$ для $n \geq 7$. Тогда мы можем заменить рекуррентное отношение

$F(n) = F(n-2) + n \% 4$, для нечетных $n > 4$

на два отношения

$F(n) = 4 \cdot (n - 3)/4 = n - 3$, для $n \geq 7$ и $n \% 4 = 3$

$F(n) = 1 + 4 \cdot (n - 5)/4 = 1 + n - 5 = n - 4$, для $n \geq 7$ и $n \% 4 = 1$

Теперь остается только найти значение n, для которого отношения будут больше или равны 1500.

$n - 3 > 1500$

$n > 1503$

1503 удовлетворяет условию n % 4 = 3. То есть все нечетные числа, большие 1503, с остатком 3 при делении на 4 будут давать значение F(n) > 1500.

n – 4 > 1500

n > 1504

Наложим условие на остаток n % 4 = 1. Первое число, которое удовлетворяет этому критерию, 1505.

Следовательно наибольшее число, для которого значение рекуррентного отношения не превосходит 1500 равно 1503.

Ответ: 1503

Алгоритмы анализа разрядов числа

Также в заданиях, пока что только авторских, могут встретиться условия на анализ состава числа в определенной системе счисления. Ярким маркером таких заданий является целочисленное деление и нахождение остатка отделения от одного и того же числа. Раньше подобные алгоритмы встречались в задачах других линий, например, циклические алгоритмы в 22 задании 2021 года.

Пример задачи:

> ***Статград 27.10.2021***
> Алгоритм вычисления значения функции $F(n)$, где n – целое неотрицательное число, задан следующими соотношениями:
> $F(0)=0$
> $F(n)=F(n/2)$ если $n>0$ и при этом n чётно
> $F(n)=1+F(n-1)$ если n нечётно.
> Сколько существует таких чисел n, что $1 \le n \le 500$ и $F(n)=8$?

Заметим, что при нечётных значениях к значению F(n-1) добавляется 1. После чего алгоритм переходит в ветку для числа на 1 меньше и уменьшает его вдвое.

Например, для F(5) = 1 + F(4) = 1 + F(2) = 1 + F(1) = 1 + 1 + F(0) = 1 + 1 = 2

Теперь рассмотрим преобразование двоичного числа $F(101_2) = 1 + F(100_2) = 1 + F(10_2) = 1 + F(1_2) = 1 + 1 + F(0_2) = 1 + 1 = 2$. Можно заключить, что алгоритм ищет количество единиц в двоичной записи числа. И нам необходимо найти количество натуральных целых чисел ≤ 500, в записи которых 8 единиц. Это числа 11111111_2 и девятизначные двоичные числа с одним нулем меньшие 111110100_2 – 101111111_2, 110111111_2, 111011111_2, 111101111_2.

Ответ: 5

Еще пример задачи на анализ разрядов в числе

Сайт К.Ю.Полякова, 6075
Обозначим частное от деления натурального числа a на натуральное число b как a // b, а остаток как a % b. Алгоритм вычисления функции F(n), где n – неотрицательное число, задан следующими соотношениями:
F(n) = 0, если n = 0
F(n) = F(n//10) + (n % 10).
Найдите количество таких чисел в диапазоне от 865 432 015, 1 585 342 628, для которых F(n) > F(n+1).

Если на каждом шаге находится сумма от целочисленного деления на 10 и прибавляется остаток от деления на 10, то мы имеем дело с рекуррентным отношением, вычисляющим сумму разрядов.

Необходимо найти такие числа, при увеличении которых на 1 получается число с меньшей суммой цифр. Такое возможно только при возникновении единицы переноса. Что в свою очередь возможно только для чисел, которые оканчиваются на 9.

Минимальное число в обозначенном диапазоне, оканчивающееся на 9 равно 865432019, максимальное – 1585342619. Вычтем из большего меньшее, разделим на 10 (подходит каждое 10 число диапазона) и добавим 1, так как необходимо, чтобы в обозначенный диапазон входили оба конца отрезка.

Ответ: 71991061

Программный метод

Самый очевидный способ, который работал даже на компьютерном ЕГЭ-2022 без доработок, но в ЕГЭ-2023 уже давал частые сбои, - просто преобразовать рекуррентное соотношение из задания в программный код.

Пример решения

Основная волна 2022

Алгоритм вычисления значения функции F(n), где n – натуральное число, задан следующими соотношениями:

$F(n)=1$ при $n<3$;

$F(n)=F(n-1)+n-1$ если $n>2$ и при этом n чётно;

$F(n)=F(n-2)+2\times n-2$ если $n>2$ и при этом n нечётно.

Чему равно значение функции F(34)?

Python	PascalABC.net

`def F(n):`
`  if n < 3:`
`    return 1`
`  elif n % 2 == 0:`
`    return F(n-1) + n - 1`
`  else:`
`    return F(n-2) + 2*n - 2`
 	
`print(F(34))`
	`###`
	`function F(n: BigInteger): BigInteger;`
	`begin`
	`  if n < 3 then`
	`    F := 1`
	`  else if n mod 2 = 0 then`
	`    F := F(n-1) + n - 1`
	`  else`
	`    F := F(n-2) + 2*n - 2;`
	`end;`
	
	`print(F(34))`

Ответ: 578

То есть единственное, что надо уметь, это представить условия в задании с помощью конструкций на выбранном языке программирования.

Пресловутый стек и “stack overflow”

Чтобы понять о чем пойдет речь далее, необходимо разобраться, как работают подпрограммы со вложенными вызовами и что за зверь – «стек». Разберемся на общем уровне, чтобы описание так или иначе могло бы быть применено к популярным языкам программирования и компиляторам/интерпретаторам.

Начнем с простого примера, такого хрестоматийного, что сводит скулы, - вычисление корней квадратного уравнения.

Предположим, что у нас арифметические операции - сложение, вычитание, деление и умножение - являются базовыми. А операции возведения в квадрат, нахождение корня, нахождения дискриминанта и сама функция нахождения корней квадратного уравнения не определены. Работать будем традиционно в рациональных числах. Сам код запишем на псевдоязыке, чтобы обобщить описание.

Для удобства не будем приводить алгоритм нахождения квадратного корня, обозначив место, где он должен быть записан.

```
Функция Квадрат(x) := x*x

Функция Корень(x) := АлгоритмВычисленияКорня(x)

Функция Дискриминант(a, b, c) := Квадрат(b) – 4*a*c

Функция КорниКвадратногоУравнения(a, b, c) :=
 Если Дискриминант(a, b, c) = 0
 То (-b / (2*a)
 Иначе Если Дискриминант(a, b, c) > 0
  То (-b - Корень(Дискриминант(a, b, c)) / (2*a), -b + Корень(Дискриминант(a, b, c)) / (2*a))
  Иначе «Нет корней».
```

До вызова любой функции стек пустой, размер стека не определяется автоматически по описанию функции и расширение стека происходит по мере необходимости. Поэтому разберем заполнение стека на конкретном примере. Стек будем изображать в виде стопки (буквальный перевод), в которую каждый новый элемент будет добавлен на верх.

Код	Что в стеке
КорниКвадратногоУравнения(1, 2, 1)	Переменные: a = 1, b = 2, c = 1 [Место вызова «КорниКвадратногоУравнения» в основной программе]
Если Дискриминант(a, b, c) = 0	Переменные: a = 1, b = 2, c = 1 [Переменные: a = 1, b = 2, c = 1 Место передачи управления: Функция «КорниКвадратногоУравнения», 1 строка после Если] [Место вызова «КорниКвадратногоУравнения» в основной программе]

<table>
<tr><td>Квадрат(x)</td><td>Переменные: x = 2<table><tr><td>Переменные: a = 1, b = 2, c = 1
Место передачи управления: Функция «Дискриминант», начало строки 1</td></tr><tr><td>Переменные: a = 1, b = 2, c = 1
Место передачи управления: Функция «КорниКвадратногоУравнения», 1 строка после Если</td></tr><tr><td>Место вызова «КорниКвадратногоУравнения» в основной программе</td></tr></table></td></tr>
<tr><td>x*x</td><td>2*2 = 4<table><tr><td>Переменные: a = 1, b = 2, c = 1
Место передачи управления: Функция «Дискриминант», начало строки 1</td></tr><tr><td>Переменные: a = 1, b = 2, c = 1
Место передачи управления: Функция «КорниКвадратногоУравнения», 1 строка после Если</td></tr><tr><td>Место вызова «КорниКвадратногоУравнения» в основной программе</td></tr></table></td></tr>
<tr><td colspan="2">Функция квадрат возвращает значение 4, из стека удаляется последний элемент (область для функции Квадрат) и управление передается в место, откуда была вызвана функция Квадрат(2) – начало строки 1 в функции Дискриминант.</td></tr>
<tr><td>Квадрат(x) – 4*a*c</td><td>Переменные: a = 1, b = 2, c = 1, Квадрат(2) = 4
Возврат значения 0 функцией «Дискриминант»
4 – 4*1*1 = 0<table><tr><td>Переменные: a = 1, b = 2, c = 1
Место передачи управления: Функция «КорниКвадратногоУравнения», 1 строка после Если</td></tr><tr><td>Место вызова «КорниКвадратногоУравнения» в основной программе</td></tr></table></td></tr>
<tr><td colspan="2">Функция Дискриминант возвращает значение 0, из стека удаляется область для функции Дискриминант и передается управление функции КорниКвадратногоУравнения в место после вызова функции Дискриминант.</td></tr>
<tr><td>Если Дискриминант(a, b, c) = 0</td><td>Переменные: a = 1, b = 2, c = 1, Дискриминант(1, 2, 1)=0
Проверка условия «Если Дискриминант(a, b, c) = 0» в функции «КорниКвадратногоУравнения»<table><tr><td>Место вызова «КорниКвадратногоУравнения» в основной программе</td></tr></table></td></tr>
<tr><td colspan="2">Условие выполняется, поэтому выполняется ветка вычислений в блоке «То »</td></tr>
<tr><td>То (-b / (2*a)</td><td>Переменные: a = 1, b = 2, c = 1, Дискриминант(1, 2, 1)=0
Возврат значения -1 функцией «КорниКвадратногоУравнения»

-2 / (2*1) = -1<table><tr><td>Место вызова «КорниКвадратногоУравнения» в основной программе</td></tr></table></td></tr>
</table>

Функция КорниКвадратногоУравнения возвращает значение -1 и передает управление в тот блок кода, откуда была вызвана. Стек очищается полностью.

Почему важно понимать как накапливается стек? Дело в том, что обычно в компиляторах и интерпретаторах существует ограничение на его размер. Или, говоря иначе, количество вложенных вызовов функций. Это своего рода защита от переполнения памяти и/или ухода в бесконечную рекурсию.

Основная волна-2023

Алгоритм вычисления значения функции $F(n)$, где n — натуральное число, задан следующими соотношениями:
$F(n)=3$ при $n<3$;
$F(n)=2\cdot n+5+F(n-2)$, если $n\geq 3$.
Чему равно значение выражения $F(3027)-F(3023)$?

Если написать решение такой задачи в лоб на Python, то получим ошибку переполнения стека. Дело в том, что условие выхода от 3027 находится на расстоянии 3026 (1513 шагов рекурсии, если учесть шаг -2). В то время как по умолчанию в Python максимальный максимальное количество вложенных вызовов функции - 1000.

Да, в *python*, например, можно увеличить размер стека с помощью метода sys.setrecursionlimit. Однако подобный способ не поможет, если в ходе выполнения функции она же сама вызывается внутри кода несколько раз.

Пример решения на python

```
from sys import setrecursionlimit
setrecursionlimit(2000)
def F(n):
  if n < 3:
    return 3
  return 2*n + 5 + F(n-2)

print(F(3027) - F(3023))
# 12114
```

Мемоизация или когда увеличение стека не работает

Увеличение стека хорошо срабатывает, когда мы имеем дело с «линейной» рекурсией. Говоря иначе, когда функция вызывает не более одной копии самой себя за один шаг. Еще такую рекурсию называют хвостовой.

> ***Апробация 10.03.23***
>
> Алгоритм вычисления значения функции $F(n)$, где n — натуральное число, задан следующими соотношениями:
>
> $F(n)=2$ при $n<3$;
> $F(n)=F(n-2)-F(n-1)+2$, если $n>2$ и при этом n чётно;
> $F(n)=F(n-1)-F(n-2)-2$, если $n>2$ и при этом n нечётно;
>
> Чему равно значение функции $F(29)$?

Просто переписав задание на язык программирования мы достаточно быстро получим ответ (на моем компьютере порядка 3-5 секунд). Попробуем увеличить значение параметра до 40 и ждать результата придется уже порядка минуты (опять же на моем компьютере).

Конечно, такую задачу в рамках ЕГЭ мы не встретим, судя по документам 16 задача считается аналитической, но можем встретить подобную задачу с выражением.

> ***Апробация 10.03.23 с измененным вопросом (1)***
>
> Алгоритм вычисления значения функции $F(n)$, где n — натуральное число, задан следующими соотношениями:
>
> $F(n)=2$ при $n<3$;
> $F(n)=F(n-2)-F(n-1)+2$, если $n>2$ и при этом n чётно;
> $F(n)=F(n-1)-F(n-2)-2$, если $n>2$ и при этом n нечётно;
>
> Чему равно значение функции $F(200)$ – F(193) + F(195)?

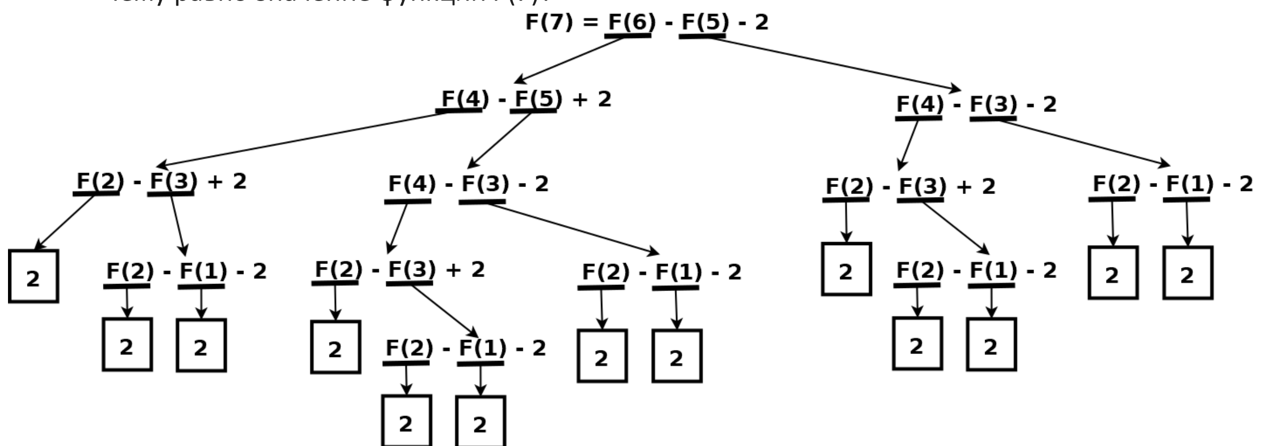
Если запустить решение без оптимизаций, даже с использованием setrecursionlimit ждать придется долго. Так как проблема не в длине стека, а в общем количестве вызовов. Если попробовать построить дерево вызовов (аналогичное дереву, которое мы строили при разборе ручных техник решения), то мы увидим порядка 2^{200} вызовов функций (точное число находится в промежутке $[2^{100}; 2^{200}]$). Что сравнимо с 10^{60} и даже при условии выполнения одного вызова на такт процессора с частотой 4 Ггц составит порядка 10^{50} секунд или примерно 10^{42} лет.

На помощь при решении подобных проблем приходят методы мемоизации (от memory – память и optimization - оптимизация). Суть методов сводится к запоминанию всех уже вычисленных ранее значений функции. Так, например, единожды вызвав F(20) при дальнейших вызовах функции F с таким же аргументами значение будет сразу получено из памяти и функция не будет вычисляться заново. Для использования данного метода можно либо подключить уже готовые библиотеки, либо реализовать свою схему хранения значений функции.

Чтобы проиллюстрировать оптимизацию по времени вычисления возьмем уже разобранную ранее задачу с аналитическим решением через построение дерева.

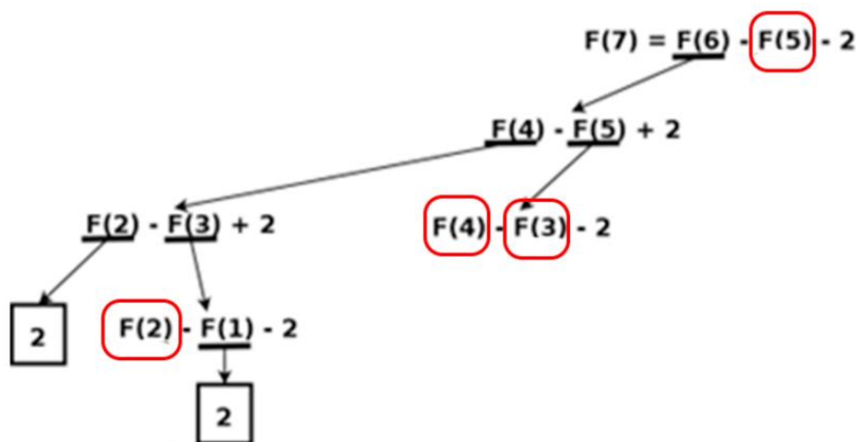
Апробация 10.03.23 с измененным вопросом

Алгоритм вычисления значения функции $F(n)$, где n — натуральное число, задан следующими соотношениями:
$F(n)=2$ при $n<3$;
$F(n)=F(n-2)-F(n-1)+2$, если $n>2$ и при этом n чётно;
$F(n)=F(n-1)-F(n-2)-2$, если $n>2$ и при этом n нечётно;
Чему равно значение функции $F(7)$?



До применения мемоизации выполняется 1 вызов F(7), 1 вызов F(6), 2 вызова F(5), 3 вызова F(4), 5 вызовов F(3), 8 вызовов F(2) и 5 вызовов F(1). Их вызовы можно посчитать на рисунке выше.

Если мы мемоизируем значения функции для параметров, то текущее дерево вызовов можно будет представить следующим образом, где красной рамкой выделены ранее найденные значения функции, которые были сохранены в памяти и поэтому вычислять их заново не нужно.



Нетрудно заметить, что вычисление значения F(7) теперь требует значительно меньшего количество вычислений значений функции.

Ручной способ мемоизации

Самый простой способ ручной мемоизации – завести структуру в глобальной области видимости, которая будет хранить в себе результаты работы функции для уже переданных параметров. В данной структуре будут храниться уже вычисленные значения.

Пример мемоизации на Python для задания с апробации 2023 с измененным вопросом (1)

```
mem = {}
def F(n):
  if n in mem:
    return mem[n]
  if n < 3:
    mem[n] = 2
  elif n % 2 == 0:
    mem[n] =  F(n-2) - F(n-1) + 2
  else:
    mem[n] = F(n-1) - F(n-2) - 2
  return mem[n]

print(F(200) - F(193) + F(195))
```

Теперь значение находится почти мгновенно. При этом изменения очень простые. Однако можно и сильно проще.

Мемоизация с помощью встроенных средств

Использование встроенных средств мемоизации и кеширования может значительно упростить данный процесс.

Например, в python есть библиотека functools, которая содержит функцию lru_cache (с python 3.8 есть еще функция cache, аналогичная lru_cache(None)).

lru_cache – декоратор (функция-обертка, которая вызывается при вызове оригинальной функции), запоминая уже вычисленные значения. С его помощью мемоизацию в предыдущей рассмотренной задаче можно сделать следующим образом. Значение None, переданное в качестве аргумента, обозначает, что нужно сохранять все вычисленные значения.

```
from functools import lru_cache
@lru_cache(None)
def F(n):
  if n < 3: return 2
  if n % 2 == 0:
    return F(n-2) - F(n-1) + 2
  else:
    return F(n-1) - F(n-2) - 2

print(F(200) - F(193) + F(195))
```

Применение декоратора lru_cache значительно упрощает мемоизацию. Однако, надо понимать, что теперь при вызове функции F сначала вызывается функция lru_cache, то есть стек заполняется в два раза быстрее – на каждый вызов F для новых параметров в стек сначала добавляется состояние функции lru_cache и потом состояние функции F.

Описание выше несколько упрощено, дабы не усложнять раздел техническими деталями.

Не вдаваясь глубоко в детали работы декоратора, для общего понимания можно описать работу lru_cache и F следующим образом:

- создается словарь для хранения наборов параметров,

- при обращении к F вызывается lru_cache (модель упрощена),

- lru_cache проверяет вызывалась ли функция F с таким набором параметров,

-- если вызывалась, то сразу возвращает значение,

-- если нет, вызывает функцию, запоминает значение и возвращает результат.

Упрощенная модель работы с lru_cache

```
cache = {}
def lru_cache(n):
  if n not in cache:
    cache[n] = F(n)
  return cache[n]


def F(n):
  if n < 3: return 2
  if n % 2 == 0:
    return lru_cache(n-2) - lru_cache(n-1) + 2
  else:
    return lru_cache(n-1) - lru_cache(n-2) - 2

print(lru_cache(200) - lru_cache(193) + lru_cache(195))
```

На самом деле все сложнее и красивее, но в рамках данного материала мы не будем разбирать, как работают декораторы в *python*. Для решения заданий это знание не поможет.

Мемоизация и глубокая рекурсия

На данном этапе может показаться, что lru_cache + setrecursionlimit в Python – панацея для любой задачи на рекуррентные отношения. На самом деле не совсем так. Дело в том, что максимальный размер стека ограничен и зависит от характеристик компьютера. Поэтому когда размер стека дойдет до верхней границы по памяти, вычислительный процесс аварийно завершится. Случится тот самый “stack overflow”.

Значит, нужно иметь в арсенале методы, которые позволяют работать с функциями безопасно. Как минимум избегая аварийных завершений процесса.

Для решения проблем с глубиной рекурсии можно воспользоваться методом, который мы разбирали в материале по аналитическим методам решения – накапливать значения от условия выхода.

> ***Основная волна-2023***
> Алгоритм вычисления значения функции $F(n)$, где n — натуральное число, задан следующими соотношениями:
> $F(n)=3$ при $n<3$;
> $F(n)=2 \cdot n+5+F(n-2)$, если $n \geq 3$.
> Чему равно значение выражения $F(3027)-F(3023)$?

```
from functools import lru_cache

@lru_cache(None)
def F(n):
  if n < 3:
    return 3
  return 2*n + 5 + F(n-2)

# переберем все значения от условия выхода
for i in range(1, 3028):
  F(i) # lru_cache запомнит значения для всех i

# достанем из памяти значения для 3027 и 3023
print(F(3027) - F(3023))
```

Такой подход позволит не углубляться при вычислении дальше, чем на 1 шаг. Так как все значения, на которые опирается вычисление значения функции, уже будут находиться в памяти.

Динамическое программирование

Вот так упрощая себе жизнь мы подошли к динамическому программированию. Возможно, у некоторых читателей на этом месте начнется паника и желание закрыть материал. Но ведь мы подошли к этому моменту постепенно, и теперь материал должен залететь на ура.

Коротко к чему мы пришли:

- чтобы рекурсивная функция не работала долго, надо сохранять уже вычисленные значения,
- чтобы рекурсивная функция не была «глубокой», надо вычислять её значения от условия выхода из рекурсии.

Так получается, что теперь мы либо знаем возвращаемое значение (например, для условия выхода), либо берем уже найденное значение. Почему бы теперь не избавиться от функции?

Облегченная схема:

- заводим структуру для хранения значений,
- заполняем её с запасом от условия выхода,
- находим все последующие значения в порядке удаления от условия выхода.

Основная волна-2023

Алгоритм вычисления значения функции $F(n)$, где n — натуральное число, задан следующими соотношениями:

$F(n)=3$ при $n<3$;

$F(n)=2\cdot n+5+F(n-2)$, если $n\geq3$.

Чему равно значение выражения $F(3027)-F(3023)$?

Решение на *python* (для мемоизации использован словарь, аналогичное решение можно реализовать с помощью списка).

```
F = {}
for n in range(3028):
  if n < 3:
    F[n] = 3
  else:
    F[n] = 2*n + 5 + F[n-2]
print(F[3027] - F[3023])
```

И все, код стал еще меньше и еще читаемее.

Проблемы погрешности

Не все так радужно в программном королевстве. И, если для больших целых значений есть специальные типы, то при работе с вещественными значениями возникают проблемы погрешности.

Все дело в том, что все числа представляются в двоичном виде и, если мы делим число на значение, которое имеет отличные от 2 простые делители или имеет в разложении большую степень 2, то итоговое значение сохраняется приближенно. Или если производится много операций деления, то малое значение после очередного деления может превратиться в 0, хотя математически нулю не равно.

А еще 0.1 + 0.2 ≠ 0.3 😊

Поэтому, если в рекуррентных отношениях используется операция деления, нужно быть весьма аккуратными при переносе такого задания в код.

На ЕГЭ пока что таких заданий не встречалось, однако такие задания не являются задачами повышенной сложности, так как, напомню, задания линии №16 считаются аналитическими.

> Алгоритм вычисления значения функции *F*(*n*), где *n* — натуральное число, задан следующими соотношениями:
> *F*(*n*) = 3 при *n*<3;
> *F*(*n*)=F(n-1)/n, если *n*≥3.
> Чему равно значение выражения *F*(499)/*F*(503)?

При попытке запуска такой задачи после перевода её в код получим ошибку «Деление на 0». Так произойдет потому, что при очередном делении результатом будет настолько малое значение, что при округлении будет сохранен только 0 (в python версии 3.10 это значение 179).

В таком случае можно воспользоваться приемом, обоснование которого исходит из понимания, что значения должны сократиться и ответ должен получиться целым. Поэтому можно переписать условие выхода на n < 500 и возвращать, например, в качестве ответа 1. Если получается нецелое значение, то вместо 1 можно подставить самое близкое к результату целое число. Если при повторном запуске получается подставленное число без дробной части, то это и есть правильный ответ.

Для приведенного задания в Python получаем

```
def F(n):
    if n < 500:
        return 1
    return F(n-1) / n
print(F(499)/F(503))
```

```
63252752999.999985
```

```
def F(n):
    if n < 500:
        return 63252753000
    return F(n-1) / n
print(F(499)/F(503))
```

```
63252753000.0
```

Объяснить подход можно просто – если мы в качестве условия выхода зададим число, которое делится на все накапливаемые делители без остатка, то полученное значение не будет накапливать погрешность, связанную с округлением.

Конечно, в популярных языках программирования есть и другие способы работать с погрешностью. Например в python есть библиотека fractions и метод Fraction в ней. Который хранит число как дробь: числитель и знаменатель хранятся отдельно и значение вычисляется только при операции вывода на экран.

```
from fractions import Fraction
def F(n):
    if n < 3:
        return Fraction(3)
    return F(n-1) / n


print(F(499)/F(503))
```

Программное решений аналитических заданий с кодом

Задание 600 с сайта kompege.ru

Определите, сколько символов * выведет эта процедура при вызове *F*(28):

Python	C++

<pre>def F(n):
   print('*')
   if n>=1:
     print('*')
     F(n-1)
     F(n-2)</pre>
	<pre>void F (int n) {
	   cout << '*';
	   if (n>=1) {
	      cout << '*';
	      F(n-1);
	      F(n-2);
	   }
	}</pre>

Pascal

```
procedure F(n:integer);
begin
   write('*');
   if n>=1 then begin
      write('*');
      F(n-1);
      F(n-2);
   end;
end;
```

1 способ (наиболее распространенный)

Завести глобальную переменную и вместо печати * добавлять к этой переменной 1. При программировании на python важно указать, что внутри функции будет использована переменная из глобальной области видимости (`global count`).

Python

```
count = 0
def F(n):
   global count
   count += 1 # print('*')
   if n>=1:
     count += 1 # print('*')
     F(n-1)
     F(n-2)

F(28)
print(count)
```

Преимущества метода: простота реализации, минимальные изменения оригинального кода.

Недостатки: при большей глубине рекурсии или большем количестве вложенных вызовов придется думать над оптимизацией.

Например, если нужно будет найти количество * при вызове F(50), то нас ждет неприятный сюрприз – придется очень долго ждать.

И тут неопытный читатель может сказать: «Есть же lru_cache». И тут же и попадет в ловушку новичка. Штука в том, что при таком изменении кода НЕОБХОДИМО, чтобы функция именно выполнялась. Потому что если будут возвращаться значения (в случае с функцией из задания None), то операции count += 1 не будут выполняться!

Python (некорректный код)

```
from functools import lru_cache
count = 0
@lru_cache(None)
def F(n):
   global count
   count += 1 # print('*')
   if n>=1:
     count += 1 # print('*')
     F(n-1)
     F(n-2)

F(28)
print(count)
```

Такой код вместо 2496118 выведет 82, что, как несложно понять, явно не то, что мы ожидаем от нашей функции.

2 способ (надежный и оптимизируемый через мемоизацию)

Перепишем функцию так, чтобы она возвращала количество выводимых на экран символов *.

Python

```
def F(n):
   count = 0
   count += 1 # print('*')
   if n>=1:
     count += 1 # print('*')
     count += F(n-1)
     count += F(n-2)
   return count

F(28)
print(count)
```

Так как теперь функция подсчитывает количество выводимых символов, мы можем кешировать результаты. Потому что не важно будет функция запускать заново или значение будет найдено в памяти.

Python

```
from functools import lru_cache

@lru_cache(None)
def F(n):
   count = 0
   count += 1 # print('*')
   if n>=1:
     count += 1 # print('*')
     count += F(n-1)
     count += F(n-2)
   return count

print(F(28))
```

Обработка выражений с невычислимыми значениями

Для иллюстрации рассмотрим мою авторскую задачу, которую рассматривали в рамках обзора аналитических решений..

> Алгоритм вычисления значения функции F(n), где n – натуральное число, задан следующими соотношениями:
> F(n) = n + 1 при n < 3
> F(n) = n + 2·F(n+2), если n ≥ 3 и чётно,
> F(n) = F(n–2) + n - 2, если n ≥ 3 и нечётно.
> Сколько существует чисел n, для которых значение F(n) определено и будет трехзначным?

Python не знает ничего про четные числа и их свойства. Поэтому ля решения этой задачи кодом есть два пути: указать python, что нужно игнорировать ошибки на превышение глубины рекурсии, или перебирать только те значения, которые не будут приводить к этой самой ошибке.

Игнорирование ошибок на превышение глубины рекурсии

Для этого нам понадобится конструкция try-except. Важно заключать в нее только ту строку, которая потенциально может привести к ошибке.

```
def F(n):
  if n < 3:
    return n +1
  if n % 2 == 0:
    return n + 2*F(n+2)
  else:
    return F(n-2) + n – 2

c = 0
for i in range(1000):
  try:
    if 100 <= F(i) <= 999:
      c += 1
  except:
    ...
print(c)
```

Представленная программа будет работать следующим образом:

- если значение вычислимо (глубина рекурсии не больше 1000), то при трехзначном значении к с будет добавлена 1,
- если значение не вычислимо (глубина рекурсии больше 1000), то ошибка будет перехвачена и в блоке except ничего не будет выполнено.

У метода, конечно же, есть ограничения и слабые места. Одно из таких слабых мест – надо понимать может ли рекурсия иметь глубину, выходящую за допустимые технические пределы (в python за 1000).

Например, если изменить вопрос в задаче следующим образом, то такой «апгрейд» программы не приведет к ответу.

> Алгоритм вычисления значения функции F(n), где n – натуральное число, задан следующими соотношениями:
> F(n) = n + 1 при n < 3
> F(n) = n + 2·F(n+2), если n ≥ 3 и чётно,
> F(n) = F(n–2) + n - 2, если n ≥ 3 и нечётно.
> Найдите значение выражения F(4013) – F(4007)?

При попытке запуска получим превышение глубины рекурсии. Бороться с этим можно увеличением допустимой глубины рекурсии, либо мемоизацией. В первом случае нужно понимать, что увеличивать глубину нельзя бесконечно (память выделяемая под стек конечна), во втором случае нужно правильно прописывать проход по значениям, чтобы не столкнуться с ситуацией, когда превышается глубина рекурсии.

Перебор только значений, для которых функция вычислима

Этот способ предполагает первичный анализ кода, когда мы замечаем невычислимость функции для четных значений (для примера выше). Для избежания описанных проблем будем перебирать только нечетные значения.

```
def F(n):
 if n < 3:
   return n +1
 if n % 2 == 0:
   return n + 2*F(n+2)
 else:
   return F(n-2) + n – 2

c = 0
for i in range(1, 1000, 2):
   if 100 <= F(i) <= 999:
     c += 1
print(c)
```