



Урок 4

Эмпирическая оценка алгоритмов на Python

Измерения времени работы с использованием timeit.
Профайлер.

[Введение](#)

[Эмпирическая оценка алгоритмов](#)

[Измерения времени работы с использованием timeit](#)

[Профайлер](#)

[Оценка сложности алгоритма](#)

[Примеры](#)

[Асимптотический анализ](#)

[Примеры для практического закрепления](#)

[Пример: Задача 1. Оптимизация алгоритма на примере вычисления чисел Фибоначчи](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Введение

На этом курсе мы изучали алгоритмы обработки и анализа данных. Пришло время задуматься об их эффективности и сложности. Сегодня разберем какие методы оценки алгоритма существуют.

Эмпирическая оценка алгоритмов

В разработке приложений есть важный процесс, который особенно необходим при создании крупных проектов – это оптимизация алгоритмов. Проектирование алгоритма можно рассматривать как способ достижения его низкой асимптотической сложности (посредством его эффективности), а оптимизацию – как уменьшение констант, скрытых в этой асимптотике.

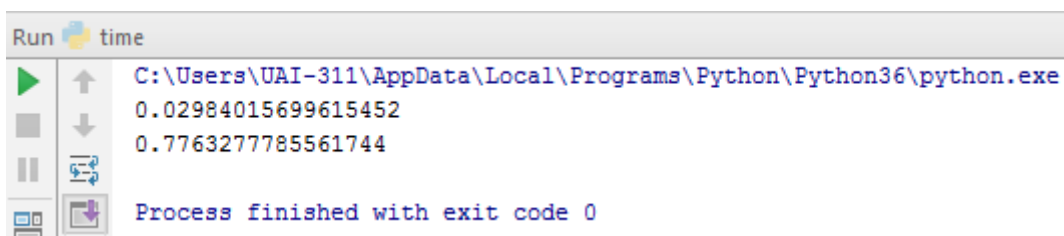
Измерения времени работы с использованием timeit

Одним из первых критериев эффективной работы алгоритма является время его выполнения. Практически во всех языках уже существует автоматизированная возможность измерения этого параметра.

В Python это модуль **timeit**. Для получения гарантированно надежных данных потребуется выполнить много работы, но для практических целей **timeit** подходит.

```
import timeit
print(timeit.timeit("x = 2 + 2"))
print(timeit.timeit("x = sum(range(10))"))
```

В результате увидим время выполнения каждой операции:



Функция `timeit` будет запускать код несколько раз, чтобы увеличить точность. Если прошлые запуски влияют на последующие, то вы окажетесь в затруднительном положении. Например, если вы измеряете скорость выполнения `mylist.sort()`, список будет отсортирован только в первый раз. Во время остальных тысяч запусков он уже будет отсортированным – это даст нереально маленький результат.

Больше информации об этом модуле можно найти в [документации стандартной библиотеки Python](#).

Профайлер

Чтобы не делать ошибочных догадок о том, какая часть кода требует оптимизации, стоит воспользоваться профайлером.

Профилировка – это измерение производительности всей программы и ее фрагментов. Цель – найти «горячие точки»: участки кода, на выполнение которых расходуется больше всего времени.

Профайлер – основной инструмент оптимизатора программ. Бывает так, что программа работает медленно из-за единственной машинной инструкции.

Например, инструкции деления, которая многократно выполняется в глубоко вложенном цикле.

Программист, приложивший титанические усилия для улучшения остального кода, окажется очень удивлен, что в результате производительность приложения возросла едва ли на 10%-15%.

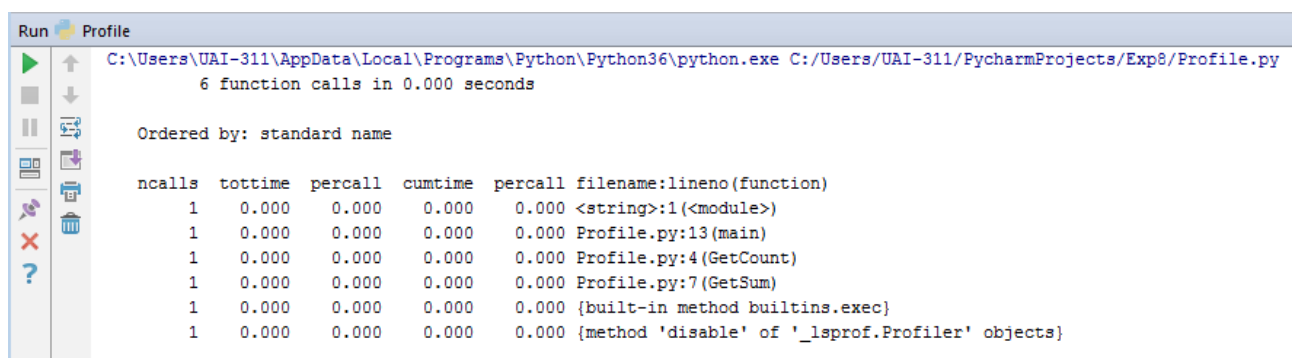
Большинство профайлеров поддерживают следующий набор базовых операций:

- Определение общего времени исполнения каждой точки программы;
- Определение удельного времени исполнения каждой точки программы;
- Определение причины и/или источника конфликтов;
- Определение количества вызовов точки программы;
- Определение степени покрытия программы.

В стандартной поставке Python есть несколько профайлеров, но рекомендуется использовать **cProfile**. Им так же легко пользоваться, как timeit, но он дает больше подробной информации о том, на что тратится время при выполнении программы. Если основная функция вашей программы называется main, вы можете использовать профайлер следующим образом:

```
def get_count(items):  
    return len(items)  
  
def get_sum(items):  
    sum_ = 0  
    for i in items:  
        sum_ += i  
    return sum_  
  
def main():  
    a = [3,5,6,7]  
    s = get_count(a)  
    t = get_sum(a)  
  
cProfile.run('main()')
```

В результате увидим информацию о главном процессе:



The screenshot shows the 'Run' and 'Profile' tool window in PyCharm. The command executed is 'C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe C:/Users/UAI-311/PycharmProjects/Exp8/Profile.py'. It shows 6 function calls in 0.000 seconds, ordered by standard name. The table below lists the function calls with their respective metrics.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.000	0.000	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	Profile.py:13(main)
1	0.000	0.000	0.000	0.000	Profile.py:4(GetCount)
1	0.000	0.000	0.000	0.000	Profile.py:7(GetSum)
1	0.000	0.000	0.000	0.000	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Такой код выведет отчет о времени работы различных функций программы. Если в вашей системе нет модуля cProfile, используйте **profile** вместо него. Больше информации о них можно найти в документации.

Если вам не так интересны детали реализации, а просто нужно оценить поведение алгоритма на конкретном наборе данных, воспользуйтесь модулем **trace** из стандартной библиотеки. С его помощью можно посчитать, сколько раз будет выполнено то или иное выражение или вызов в программе.

- **Будьте внимательны в выводах, основанных на сравнении времени выполнения!**

Во-первых, любая разница может определяться случайностью. Риск меньше, если вы используете специальные инструменты вроде timeit: они измеряют время вычисления выражения несколько раз (или даже повторяют весь замер), выбирая лучший результат. Случайные погрешности неизбежны. Если разница между двумя реализациями не превышает некоторой допустимой величины, то нельзя сказать, что эти реализации различаются. Но и утверждать, что они одинаковы, мы тоже не можем.

Проблема усложняется при сравнении более двух реализаций. Количество пар для сравнения увеличивается пропорционально квадрату количества сравниваемых версий. Так растет и вероятность того, что как минимум две из версий будут казаться слегка различными. Это проблема множественного сравнения.

Для нее есть статистические решения, но самый простой способ – повторить эксперимент с двумя подозрительными версиями. Возможно, потребуется сделать это несколько раз и ответить: по-прежнему ли они выглядят похожими?

Во-вторых, есть несколько моментов, на которые нужно обращать внимание при сравнении средних величин. Как минимум, надо сравнивать средние значения реального времени работы. Чтобы получить показательные числа при измерении производительности, обычно время работы каждой программы нормируется делением на время выполнения стандартного, простого алгоритма.

Это может сработать, но в ряде случаев сделает бессмысленными результаты замеров. Несколько полезных указаний на эту тему можно найти в статье [«How not to lie with statistics: The correct way to summarize benchmark results»](#) (авторы – Philip Fleming, John Wallace). Также можно почитать [«Don't compare averages»](#) (авторы – Holger Bast, Ingmar Weber) или более новую статью [«The harmonic or geometric mean: does it really matter?»](#) (авторы – Daniel Citron, Adham Hurani, Alaa Gnadrey).

И в-третьих, выводы могут не подлежать обобщению. Подобные измерения на другом наборе входных данных и «железе» могут дать иные результаты. Если кто-то будет пользоваться результатами ваших измерений, необходимо последовательно задокументировать, каким образом они были получены.

- **Будьте осторожны, делая выводы об асимптотике из экспериментов!**

Если необходимо сделать окончательные выводы об асимптотике алгоритма, то необходимо проанализировать ее, как описано ранее в этой главе. Эксперименты могут давать намеки, но они проводятся на конечных наборах данных. А асимптотика происходит при сколь угодно больших размерах данных.

Если вы не работаете в академической сфере, то цель асимптотического анализа – сделать вывод о поведении алгоритма, реализованного конкретным способом и запущенного на определенном наборе данных. Значит, измерения должны быть соответствующими.

Например, вы предполагаете, что алгоритм работает с квадратичной сложностью, но не можете окончательно доказать это. Можете ли вы использовать эксперименты для доказательства предположения?

Эксперименты (и оптимизация алгоритмов) имеют дело в основном с постоянными коэффициентами, но выход есть. Основной проблемой является то, что вашу гипотезу нельзя проверить экспериментально. Если вы утверждаете, что алгоритм имеет сложность $O(n^2)$, то данные не могут это ни подтвердить, ни опровергнуть.

Но если сделать гипотезу более конкретной, то она станет проверяемой. Основываясь на некоторых данных, можно положить, что время работы программы никогда не будет превышать $0.24n^2 + 0.1n + 0.03$ секунд в вашем окружении. Это опровергаемая гипотеза. Если сделано множество измерений, но так и не найдены контрпримеры – значит гипотеза может быть верна. А это подтверждает гипотезу о квадратичной сложности алгоритма.

Оценка сложности алгоритма

Сложность алгоритмов обычно оценивают по времени выполнения или используемой памяти. Точное время выполнения мало кого интересует: оно зависит от процессора, типа данных, языка программирования и других параметров. Важна лишь асимптотическая сложность – при стремлении размера входных данных к бесконечности.

Допустим, алгоритму нужно выполнить $4 \cdot n^3 + 7 \cdot n$ условных операций, чтобы обработать n элементов входных данных. При увеличении n на итоговое время работы будет значительно больше влиять возведение n в куб, чем умножение его на 4 или же прибавление $7n$. Таким образом, временная сложность этого алгоритма равна $O(n^3)$, то есть зависит от размера входных данных кубически.

Использование заглавной буквы O (или так O -нотация) пришло из математики, где ее применяют для сравнения асимптотического поведения функций. Формально $O(f(n))$ означает, что время работы алгоритма (или объем занимаемой памяти) растет в зависимости от объема входных данных не быстрее, чем некоторая константа, умноженная на $f(n)$.

Примеры

1. $O(n)$ – линейная сложность

Линейной сложностью обладают, например, алгоритмы поиска в уже отсортированном массиве данных: когда необходимо произвести проход по n элементам массива, чтобы найти необходимый.

2. $O(\log n)$ – логарифмическая сложность

Логарифмической сложностью обладает, например, бинарный поиск. Для нахождения конкретного элемента уже упорядоченный массив делится пополам. Если полученный средний элемент больше элемента, который мы ищем, то можно отбросить вторую половину массива. В противоположном случае можно исключить левую сторону, так как средний элемент меньше. Деление пополам продолжается до тех пор, пока искомый элемент не будет найден. В итоге проверим $\log n$ элементов.

3. $O(n^2)$ – квадратичная сложность

Такую сложность имеет алгоритм сортировки вставками. В классической реализации он состоит из двух вложенных циклов. Внешний – для прохода по всему массиву, а внутренний – для поиска места элемента в уже упорядоченном массиве. Таким образом, количество операций будет зависеть от размера массива как $n \cdot n$, то есть n^2 .

Время работы алгоритма может не зависеть от размера входных данных. Тогда сложность обозначают как $O(1)$. Например, для определения значения третьего элемента массива не нужно ни запоминать элементы, ни проходить по ним. Надо просто дождаться в потоке входных данных третий элемент, и это будет результатом. На его вычисление для любого количества данных нужно одно и то же время.

Аналогично проводят оценку и по памяти, когда это важно. Но алгоритмы могут использовать значительно больше памяти при увеличении размера входных данных, зато работать быстрее. И может быть и наоборот. Это помогает выбирать оптимальные пути решения задач, исходя из текущих условий и требований.

Асимптотический анализ

Когда мы говорим об измерении сложности алгоритмов, мы подразумеваем анализ времени, которое потребуется для обработки очень большого набора данных. Такой анализ называют асимптотическим. Сколько времени потребуется на обработку массива из десяти элементов? Тысячи? Десяти миллионов? Если алгоритм обрабатывает тысячу элементов за пять миллисекунд, что случится, если мы передадим в него миллион? Будет ли он выполняться пять минут или пять лет? Это стоит выяснить раньше заказчика.

Порядок роста описывает, как сложность алгоритма растет с увеличением размера входных данных. Чаще всего, он представлен в виде O -нотации: $O(f(x))$, где $f(x)$ – формула, выражающая сложность алгоритма. В формуле может присутствовать переменная n , представляющая размер входных данных. Ниже приводится список наиболее часто встречающихся порядков роста, но он не исчерпывающий.

- **Константный $O(1)$**

Порядок роста $O(1)$ означает, что вычислительная сложность алгоритма не зависит от размера входных данных. Помним, что единица в формуле не значит, что алгоритм выполняется за одну операцию или занимает очень мало времени. Он может потребовать и микросекунду, и год. Важно то, что это время не зависит от входных данных.

```
def get_count(items):  
    return len(items)
```

- **Линеарифметический – $O(n \cdot \log n)$**

Линеарифметический, или линейно-логарифмический, алгоритм имеет порядок роста $O(n \cdot \log n)$. Некоторые алгоритмы типа «разделяй и властвуй» попадают в эту категорию. Далее мы увидим два таких примера – сортировка слиянием и быстрая сортировка.

- **Наилучший, приемлемый (средний) и наихудший вариант алгоритма**

Что имеется в виду под порядком роста сложности алгоритма $O(n)$? Это усредненный случай? Или наихудший? А может быть, наилучший?

Обычно подразумевается наихудший случай, за исключением тех ситуаций, когда он сильно отличается от среднего. Есть алгоритмы, которые в среднем имеют порядок роста $O(1)$, но периодически могут становиться $O(n)$. В этом случае можно сказать, что алгоритм работает в среднем за константное время, и объяснять случаи, когда сложность возрастает.

Порядок роста сложности алгоритма $O(n)$ означает, что алгоритм потребует не более n шагов.

Что мы измеряем?

При измерении сложности алгоритмов и структур данных мы обычно говорим о двух вещах:

1. О количестве операций, требуемых для завершения работы;
2. Об объеме ресурсов (в частности, памяти), который необходим алгоритму.

Алгоритм, который выполняется в десять раз быстрее, но использует в десять раз больше места, может вполне подходить для серверной машины с большим объемом памяти. Но на встроенных системах, где количество памяти ограничено, его использовать нельзя.

Наиболее часто определение количества операций, необходимого для завершения работы, рассматривается на примере алгоритмов сортировки.

Операции, количество которых обычно измеряется:

- сравнения («больше», «меньше», «равно»);
- присваивания;
- выделение памяти;

То, какие операции мы учитываем, обычно ясно из контекста.

Примеры для практического закрепления

Пример: Задача 1. Оптимизация алгоритма на примере вычисления чисел Фибоначчи

Классический пример организации рекурсивного перебора – это вычисления ряда Фибоначчи.

На языке Python этот алгоритм выглядит так:

```
def f(n):  
    if n < 2:  
        return n  
    return f(n - 1) + f(n - 2)
```

Пока не будем заострять внимание на корректности входных данных.

Этот алгоритм имеет очень высокую временную сложность:

1. Каждый вызов функции порождает следующие два;
2. Последующие вызовы функции влекут за собой свои два вызова;
3. Так будет происходить, пока входной параметр функции не достигнет значения единицы.

Получим дерево вызова функции, наибольшая длина которого будет равна N , а число вызовов функции – 2^N .

Запишем получившееся дерево вызова функции:

1. $2^0 = 1$ вызов: $f(n)$

2. $2^1 = 2$ вызова: $f(n-1)$, $f(n-2)$
3. $2^2 = 4$ вызова: $f(n-1-1)$, $f(n-1-2)$, $f(n-2-1)$, $f(n-2-2)$
4. $2^3 = 8$ вызовов: $f(n-3)$, $f(n-4)$, $f(n-4)$, $f(n-5)$, $f(n-4)$, $f(n-5)$, $f(n-5)$, $f(n-6)$
5. $2^4 = 16$ вызовов: $f(n-4)$, $f(n-5)$, $f(n-5)$, $f(n-6)$, $f(n-5)$, $f(n-6)$, $f(n-6)$, $f(n-7)$, $f(n-5)$, $f(n-6)$, $f(n-6)$, $f(n-7)$, $f(n-6)$, $f(n-7)$, $f(n-7)$, $f(n-8)$
6. $2^5 = 32$ вызова: $f(n-5)$, $f(n-6)$, $f(n-6)$, $f(n-7)$, $f(n-6)$, $f(n-7)$, $f(n-7)$, $f(n-8)$, $f(n-6)$, $f(n-7)$, $f(n-7)$, $f(n-8)$, $f(n-7)$, $f(n-8)$, $f(n-9)$, $f(n-6)$, $f(n-7)$, $f(n-7)$, $f(n-8)$, $f(n-7)$, $f(n-8)$, $f(n-8)$, $f(n-9)$, $f(n-7)$, $f(n-8)$, $f(n-8)$, $f(n-9)$, $f(n-8)$, $f(n-9)$, $f(n-9)$, $f(n-10)$
7. ...
8. 2^k вызовов
9. ...
10. $f(n-m) == f(1)$, $f(n-m) == f(1)$, ... , $f(n-m) == f(1)$

Оценка вычислительной сложности алгоритма в 2^n поверхностная, но достаточная, чтобы заключить, что алгоритм будет затрачивать существенное время даже на относительно малых значениях n . Поэтому для практического использования его требуется оптимизировать.

Результат измерения производительности:

Time taken for $f(32)$: 1.057438 seconds

Функция многократно вызывается для одного и того же значения параметра n . Поэтому на первом этапе применим мемоизацию. Для этого напишем декоратор:

```
def memorize(func):
    def g(n, memory={}):
        r = memory.get(n)
        if r is None:
            r = func(n)
            memory[n] = r
        return r
    return g
```

И применим его к функции:

```
@memorize
def f(n):
    if n < 2:
        return n
    return f(n - 1) + f(n - 2)
```

Теперь повторный вызов функции с одинаковым значением не будет приводить к рекурсии. Вместо этого результат будет возвращаться из словаря memoгу, в который записываются результаты предыдущих вычислений.

Хронология вызовов будет такой:

$f(n), f(n-1), f(n-2), f(n-3), f(n-4), \dots, f(1), f(0), f(1), \dots, f(n-5), f(n-4), f(n-3), f(n-2).$

Добавляя мемоизирующий декоратор, удалось снизить сложность алгоритма с экспоненциальной (2^n) до линейной ($2n$).

Мемоизирующий декоратор и функцию $f()$ можно еще немного улучшить:

```
def memorize(func):
    @wraps(func)
    def g(n, memory={0: 0, 1: 1}):
        r = memory.get(n)
        if r is None:
            r = func(n)
            memory[n] = r
        return r
    return g

@memorize
def decorated_f(n):
    return decorated_f(n - 1) + decorated_f(n - 2)
```

Результат измерения производительности:

Time taken for decorated_f(32): 0.000043 seconds

Но в алгоритме есть еще как минимум два недостатка.

Во-первых, для мемоизации результатов выполнения функции $f()$ используется словарь и накладные расходы на хеширование ключа —,при доступе к значениям. Этого можно избежать, используя список, в котором доступ к значениям будет производиться по индексу без хеширования.

Во-вторых, на вызов декорирующей функции тратится дополнительное время. Но в большей степени наличие декорирующей функции влияет на ограничение максимального числа Фибоначчи, которое может быть вычислено с помощью алгоритма. Это число равно примерно 480 для ограничения рекурсии Python, по умолчанию равного 1000. Попробуем устранить эти недостатки.

Оптимизированный алгоритм:

```
def f(n, memory=[0, 1]):
    if n < len(memory):
        return memory[n]
    else:
        r = f(n - 1) + f(n - 2)
        memory.append(r)
    return r
```

При заполнении списка используется тот факт, что для вычисления следующего числа Фибоначчи требуется сначала вычислить предыдущее.

Максимальное число Фибоначчи, которое может быть вычислено этим алгоритмом, равно примерно 960. Лимит можно увеличить, вызвав функцию `sys.setrecursionlimit(limit)`, а также соответственно увеличив стек интерпретатора. Но алгоритм останется ограничен конечной глубиной стека.

Но при последовательном вычислении чисел Фибоначчи от меньшего к большему стек перестает быть ограничением максимального числа. Это происходит за счет мемоизации, которая предотвращает глубокую рекурсию, возвращая ранее вычисленные значения из памяти.

Результат измерения производительности:

Time taken for builtin_list_memoization_f(32): 0.000035 seconds

Рекурсивные алгоритмы часто проигрывают в производительности своим нерекурсивным аналогам: в первую очередь, из-за более высокой вычислительной сложности; а во вторую – из-за накладных расходов на рекурсивные вызовы функции. В последней версии алгоритма узким местом являются накладные расходы на вызов функции.

После переписывания алгоритма в нерекурсивную форму он будет выглядеть так:

```
def f(n):  
    if n < 2:  
        return n  
    pp = 0  
    p = 1  
    for i in range(n - 1):  
        pp, p = p, pp + p  
    return p
```

Алгоритм не содержит встроенных ограничений на максимальное число, которое может быть вычислено.

Теоретически данный алгоритм имеет линейную вычислительную сложность. Но это верно только до тех пор, пока операцию сложения можно считать атомарной. На больших числах, которыми процессор не может оперировать непосредственно (например, загрузить в регистры и выполнить инструкцию add), суммирование не является атомарной операцией и начинает зависеть от количества разрядов в числе. А оно зависит от n – в результате алгоритм имеет более высокую, чем линейная, вычислительную сложность на больших n .

В нелинейной вычислительной сложности можно убедиться и по результатам измерений производительности. Для значений n до 10000 наблюдается примерно линейный рост затрат времени в зависимости от n . Далее наблюдается очевидно нелинейный характер роста временных затрат.

Результат измерения производительности:

Time taken for loop_f(32): 0.000004 seconds

Time taken for loop_f(70): 0.000005 seconds

Time taken for loop_f(100): 0.000010 seconds

Time taken for loop_f(1000): 0.000110 seconds

Time taken for loop_f(10000): 0.003389 seconds

Time taken for loop_f(100000): 0.210829 seconds

Time taken for loop_f(1000000): 19.969845 seconds

Зависимость количества знаков числа Фибоначчи от его номера:

```
len(str(fibonacci.loop_f(1)))
len(str(fibonacci.loop_f(10)))
len(str(fibonacci.loop_f(100)))
len(str(fibonacci.loop_f(1000)))
len(str(fibonacci.loop_f(10000)))
len(str(fibonacci.loop_f(100000)))
len(str(fibonacci.loop_f(1000000)))
```

Дополнительно нерекурсивную версию можно снабдить мемоизирующим декоратором (на случай многократных вызовов):

```
@memorize
def f(n):
    if n < 2:
        return n
    pp = 0
    p = 1
    for i in range(n - 1):
        pp, p = p, pp + p
    return p
```

Еще лучше – запоминать рассчитанные значения в списке: так доступ к ним будет быстрее, чем при использовании словаря.

Существует функция прямого расчета значения n-ного числа Фибоначчи:

$F(n) = (((1 + \sqrt{5}) / 2)^n - ((1 - \sqrt{5}) / 2)^n) / \sqrt{5}$

Алгоритм и его оптимизированные версии:

```
SQRT_5 = math.sqrt(5)
GOLDEN_RATIO = (1 + SQRT_5) / 2
OTHER_PRECALC = (1 - SQRT_5) / 2

ONE_PLUS_SQRT_5 = (1 + SQRT_5)
ONE_MINUS_SQRT_5 = (1 - SQRT_5)

def closed_form_f(n):
    return round((GOLDEN_RATIO**n - OTHER_PRECALC**n) / SQRT_5)

def closed_form_f_improved(n):
    return round((ONE_PLUS_SQRT_5**n - ONE_MINUS_SQRT_5**n) / (2**n * SQRT_5))

def closed_form_f_more_improved(n):
    return round(GOLDEN_RATIO**n / SQRT_5)
```

У этих алгоритмов имеются существенные ограничения, которые делают их практически незаменимыми. Во-первых, накопленная ошибка вычислений не позволяет точно рассчитать значение числа Фибоначчи более $n = 70$. Во-вторых, ограничения на возможные значения дробных чисел, используемых в арифметике Python, не позволяют выполнить даже приближенный расчет числа Фибоначчи более $n = 1474$.

Rounding error at closed_form_f(71)

Numeric overflow error at closed_form_f(1475)

Rounding error at closed_form_f_improved(71)

Numeric overflow error at closed_form_f_improved(605)

Rounding error at closed_form_f_more_improved(71)

Numeric overflow error at closed_form_f_more_improved(1475)

Вычислительная сложность алгоритмов – $\log n$, так как формула содержит возведение в степень, вычислительная сложность которой – $\log n$.

Результат измерения производительности:

Time taken for closed_form_f(32): 0.000039 seconds

Time taken for closed_form_f(70): 0.000008 seconds

Time taken for closed_form_f_improved(32): 0.000009 seconds

Time taken for closed_form_f_improved(70): 0.000016 seconds

Time taken for closed_form_f_more_improved(32): 0.000007 seconds

Time taken for closed_form_f_more_improved(70): 0.000004 seconds

Кроме вышеперечисленного, существует теоретически более производительный алгоритм расчета числа Фибоначчи: через матрицы и рекуррентные соотношения.

Подведем итоги урока:

1. После написания программы стоит посмотреть на время ее выполнения. Если вы видите, что его нужно уменьшить, но не знаете, где слабое звено, можно воспользоваться профайлерами. Они определяют, на какую операцию или функцию затрачивается наибольшее время;
2. Если вычислительная сложность алгоритма высока, стоит задуматься над его оптимизацией: уменьшить количество вызовов функций, перестроить алгоритм из рекурсивного перебора в итерационный. Возможно, следует уйти от многократных переборов данных с помощью условий проверки. Думайте, играйте с исходными данными и уже разработанными решениями.

Практическое задание

1. Проанализировать скорость и сложность одного любого алгоритма, разработанных в рамках практического задания первых трех уроков.

Примечание: попробуйте написать несколько реализаций алгоритма и сравнить их.

2. Написать два алгоритма нахождения i -го по счёту простого числа.
 - Без использования Решета Эратосфена;
 - Использовать алгоритм решето Эратосфена

Примечание ко всему практическому заданию: Проанализировать скорость и сложность алгоритмов. Результаты анализа сохранить в виде комментариев в файле с кодом.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. <https://www.python.org>
2. Марк Лутц. Изучаем Python, 4-е издание.