# Эмпирическая оценка алгоритмов

**Введение**

Итак, теперь мы знаем, как оценить эффективность алгоритма без получения цифровых значений времени их выполнения. Для этого мы применяем О-нотацию. Но существуют случаи, когда необходимо узнать и цифровые значения.

1. Алгоритм перенесен на код языка программирования, но определить сложность в О-нотации представляется затруднительной задачей.
2. Требуется собрать статистику времени работы алгоритмов для формирования отчетов.

**Модуль timeit**

Предлагает разработчику простые средства замера быстродействия фрагментов Python-кода. Подробное описание классов модуля доступно в документации. Рассмотрим некоторые из основных классов.

class timeit.**Timer**(*[stmt='pass', setup='pass']*)

Здесь stmt – выражение, которое мы замеряем, setup – выражение, выполняемое перед основным замеряемым выражением.

Timer.**timeit**(*[number=1000000]*)

Команда выполнит setup-выражение один раз и вернет время, необходимое для выполнения основного stmt-выражения number-раз. По умолчанию number=1000000.

**Листинг 1. task\_1.py**

|  |
| --- |
| *"""Генерация списков"""* **from** timeit **import** Timer   **def** test\_concat():  my\_lst = []  **for** i **in** range(1000):  my\_lst = my\_lst + [i]   **def** test\_cycle():  my\_lst = []  **for** i **in** range(1000):  my\_lst.append(i)   **def** test\_gener():  my\_lst = [i **for** i **in** range(1000)]   **def** test\_range():  my\_lst = list(range(1000))   t1 = Timer(**"test\_concat()"**, **"from \_\_main\_\_ import test\_concat"**) print(**"concat "**, t1.timeit(number=1000), **"milliseconds"**)  t2 = Timer(**"test\_cycle()"**, **"from \_\_main\_\_ import test\_cycle"**) print(**"append "**, t2.timeit(number=1000), **"milliseconds"**)  t3 = Timer(**"test\_gener()"**, **"from \_\_main\_\_ import test\_gener"**) print(**"comprehension "**, t3.timeit(number=1000), **"milliseconds"**)  t4 = Timer(**"test\_range()"**, **"from \_\_main\_\_ import test\_range"**) print(**"list range "**, t4.timeit(number=1000), **"milliseconds"**)  **""" concat 1.1779784 milliseconds append 0.0715625000000002 milliseconds comprehension 0.033750200000000063 milliseconds list range 0.011227300000000273 milliseconds  Вы можете объяснить получение таких результатов? """** |

Timer.**repeat**(*[repeat=3[,number=1000000]]*)

Выполнение вызова timeit() c указанным repeat (количеством повторений замеров). При этом аргумент number передается в timeit.

Начиная с Python 2.6 разработчику доступны более удобные функции:

timeit.**timeit**(*stmt[, setup[, timer[, number=1000000]]]]*)

Создает экземпляр класса Timer, передает в конструктор входные параметры, вызывает метод timeit(), возвращает результат в секундах типа float.

timeit.**repeat**(*stmt[, setup[, timer[, repeat=3[, number=1000000]]]]*)

Создает экземпляр класса Timer, передает в конструктор входные параметры, вызывает метод repeat(), возвращает список результатов.

На практике использование этих функций можно встретить чаще, чем предыдущий вариант.

Кстати, замерять мы можем не только код в виде функций, но и просто блоки кода.

**Листинг 2. task\_2.py**

|  |
| --- |
| *"""Замеряем обычные блоки кода"""* **from** timeit **import** timeit  print(timeit(**"x = 2 + 2"**)) print(timeit(**"x = sum(range(10))"**))  print(timeit(**""" for i in range(3):  y = i + 2  a = 4  if a == y:  1/2 """**))  **""" 0.010643799999999981 0.3782346000000001 0.40320979999999995 """** |

И выявлять таким образом проблемные участки кода.

Рассмотрим пример посложнее.

**Листинг 3. task\_3.py**

|  |
| --- |
| *"""Замеряем обычные блоки кода Получаем массив квадратов диапазона чисел """* **from** timeit **import** repeat, default\_timer  *# до запуска замеров выполняем генерацию массива* setup = **"elems=range(2000)"** *# выражения, которые необходимо выполнить для замеров* statements = [  **'[el\*el for el in elems]'**,  **'''res=[] for el in elems:  res.append(el\*el)'''**,  **'map(lambda el: el\*el, elems)'**]  *# перебираем выражения* **for** st **in** statements:  *# для каждого выражение вычисляем время и переносим его на 10000 запусков этого кода  # делаем по три замера* print(repeat(st, setup, default\_timer, 3, 10000))   **""" [1.3041752, 1.3166882000000002, 1.356252] [2.4010173999999997, 2.4017775000000006, 2.421308] [0.00282190000000071, 0.0021614999999997053, 0.0019029000000010399] """** |

Продолжим практиковаться с замерами.

**Листинг 4. task\_4.py**

|  |
| --- |
| *"""Еще замеры с timeit"""* **import** timeit  **""" Эта команда выполнит выражение setup один раз,  а затем возвратит время в секундах типа float,  которое требуется что бы выполнить основное выражение number раз. """   def** concat\_test():  my\_lst = []  **for** i **in** range(1000):  my\_lst += [i]   **def** append\_test():  my\_lst = []  **for** i **in** range(1000):  my\_lst.append(i)   *# через строку кода* STR\_CODE = **''' l = []  for i in range(1000):   l += [i] '''** *# еще через строку кода* STR\_CODE\_2 = **''' j = sum(range(1, 1000)) '''** print(timeit.timeit(**"concat\_test()"**, setup=**"from \_\_main\_\_ import concat\_test"**, number=1000)) print(timeit.timeit(**"append\_test()"**, setup=**"from \_\_main\_\_ import append\_test"**, number=1000)) print(timeit.timeit(STR\_CODE, number=1000)) print(timeit.timeit(STR\_CODE\_2, number=1000))  **""" Сможете объяснить результаты? """** |

**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:

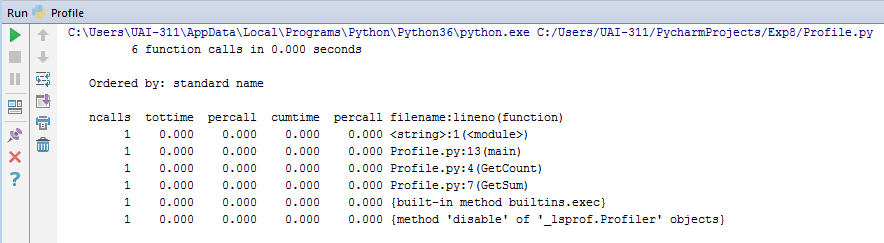
* Определение общего времени исполнения каждой точки программы;
* Определение удельного времени исполнения каждой точки программы;
* Определение причины и/или источника конфликтов;
* Определение количества вызовов точки программы;
* Определение степени покрытия программы.

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. Профилирование кода с **cProfile** это достаточно просто. Все что вам нужно сделать, это [импортировать модуль](https://python-scripts.com/import-modules-python) и вызвать его функцию **run**.

**Листинг 5. task\_5.py**

|  |
| --- |
| *"""Профилировка"""* **import** cProfile   **def** get\_count(items):  **return** items.\_\_len\_\_()   **def** get\_sum(items):  sum = 0  **for** i **in** items:  sum += i  **return** sum   **def** main():  my\_lst = [3, 5, 6, 7]  res\_count = get\_count(my\_lst)  res\_sum = get\_sum(my\_lst)   cProfile.run(**'main()'**) |

В результате увидим информацию о главном процессе:



Первая строка показывает, что в ней 6 вызова функций. На запуск всех функций ушло 0 секунд. Слабых мест нет. Следующая строка говорит нам, в каком порядке результаты выдачи. Здесь есть несколько столбцов.

* **ncalls** – это количество совершенных вызовов;
* **tottime** – это все время, потраченное в данной функции;
* **percall** – ссылается на коэффициент tottime, деленный на ncalls;
* **cumtime** – совокупное время, потраченное как в данной функции, так и наследуемых функциях. Это работает также и с рекурсивными функциями!
* Второй столбец **percall** – это коэффициент cumtime деленный на примитивные вызовы;
* **filename:lineno(function)** предоставляет соответствующие данные о каждой функции.

Рассмотрим пример, в котором создадим «слабые» места самостоятельно.

**Листинг 6. task\_6.py**

|  |
| --- |
| *"""Профилировка времени выполнения функций"""* **import** cProfile **import** time   **def** fast():  print(**"Я быстрая функция"**)   **def** slow():  time.sleep(3)  print(**"Я очень медленная функция"**)   **def** medium():  time.sleep(0.5)  print(**"Я средняя функция..."**)   **def** main():  fast()  slow()  medium()   cProfile.run(**'main()'**) |

**Продолжим освоение модуля timeit**

Закрепим полученные знания на практике и положим конец спорам относительно эффективности способов форматирования строк.

**Листинг 7. task\_7.py**

|  |
| --- |
| **import** timeit   **def** format\_concate():  param = 123  string = **"first "** + str(param) + **" second"   def** format\_percent():  param = 123  string = **"first %s second"** % param   **def** format\_format():  param = 123  string = **"first {} second"**.format(param)   **def** format\_f():  param = 123  string = **f"first {**param**} second"** print(timeit.timeit(**"format\_concate()"**, setup=**"from \_\_main\_\_ import format\_concate"**))  print(timeit.timeit(**"format\_percent()"**, setup=**"from \_\_main\_\_ import format\_percent"**))  print(timeit.timeit(**"format\_format()"**, setup=**"from \_\_main\_\_ import format\_format"**))  print(timeit.timeit(**"format\_f()"**, setup=**"from \_\_main\_\_ import format\_f"**))  **""" 0.33543220000000007 0.31619379999999997 0.37685690000000016 0.22341030000000006 """** |

**Замеры методов класса с помощью модуля timeit**

Такую задачу мы также можем выполнить. Рассмотрим, как ее решить на примере:

**Листинг 8. task\_8.py**

|  |
| --- |
| **from** time **import** sleep **from** timeit **import** repeat, default\_timer   **class** TestClass:   @staticmethod  **def** some\_slow\_method(loop\_count):  **for** i **in** range(loop\_count):  sleep(1)   @staticmethod  **def** some\_quick\_method(loop\_count):  **for** i **in** range(loop\_count):  sleep(0.1)   **if** \_\_name\_\_==**'\_\_main\_\_'**:  *# Для того что бы воспользоваться классом необходимого модуля,  # его необходимо импортировать в инициализирующем выражении setup  # а также создать экземпляр класса* setup = **""" from \_\_main\_\_ import TestClass test = TestClass()  """** statements = [**'test.some\_slow\_method(5)'**,  **'test.some\_slow\_method(3)'**,  **'test.some\_quick\_method(5)'**,  **'test.some\_quick\_method(3)'**]   **for** st **in** statements:  print(**f'{**st**}, {**min(repeat(st, setup, default\_timer, 3, 1))**}'**)  **""" test.some\_slow\_method(5), 5.002561600000001 test.some\_slow\_method(3), 3.000246899999997 test.some\_quick\_method(5), 0.5015900000000002 test.some\_quick\_method(3), 0.30135129999999677 """** |

А теперь попробуем самостоятельно определить время, но уже немного другим путем.

**Листинг 9. task\_9.py**

|  |
| --- |
| *"""Генерация целых чисел"""* **from** timeit **import** default\_timer, timeit   **def** gen\_prime(x):  multiples = []  results = []  **for** i **in** range(2, x+1):  **if** i **not in** multiples:  results.append(i)  **for** j **in** range(i\*i, x+1, i):  multiples.append(j)   **return** results   *# левая отсечка времени* start\_time = default\_timer() *# запуск функции* gen\_prime(3000) *# правая отсечка времени и результат* print(default\_timer() - start\_time)  *# сравним с привычным вариантом замеров* print(timeit(**"gen\_prime(3000)"**, setup=**"from \_\_main\_\_ import gen\_prime"**, number=1))  **""" 0.0578244 0.05766979999999999  Существенных расхождений не выявлено """** |

Вернемся к задаче вычисления чисел Фибоначчи и применим возможности модуля timeit к ее решению через цикл, рекурсию и оптимизированную рекурсию.

Цикл:

**Листинг 10. task\_10.py**

|  |
| --- |
| *"""Фибо через цикл"""* **import** timeit   **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   n = 8  print(timeit.timeit(**"f(n)"**, setup=**"from \_\_main\_\_ import f, n"**))  **"""0.6885986000000002"""** |

Рекурсия:

**Листинг 10. task\_10.py**

|  |
| --- |
| *"""Фибо через рекурсию"""* **import** timeit   **def** f(n):  **if** n < 2:  **return** n  **return** f(n - 1) + f(n - 2)   n = 8  print(timeit.timeit(**"f(n)"**, setup=**"from \_\_main\_\_ import f, n"**))  **"""8.6776222"""** |

Рекурсия c мемоизацией:

**Листинг 10. task\_10.py**

|  |
| --- |
| *"""Фибо через рекурсию с мемоизацией через декоратор"""* **import** timeit   **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   @memorize **def** f(n):  **if** n < 2:  **return** n  **return** f(n - 1) + f(n - 2)   n = 8  print(timeit.timeit(**"f(n)"**, setup=**"from \_\_main\_\_ import f, n"**))  **"""0.19176139999999997"""** |

Обычный рекурсивный 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-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.

Функция многократно вызывается для одного и того же значения параметра n. Поэтому на первом этапе применим мемоизацию. Мемоизация — это подход, который позволяет нам сохранять результаты промежуточных решений, для того чтобы в следующих расчетах не повторять тоже самое.

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

f(n), f(n-1), f(n-2), f(n-3), f(n-4), …, f(1), f(0), f(1), …, f(n-5), f(n-4), f(n-3), f(n-2).  
  
Добавляя мемоизирующий декоратор, удалось снизить сложность алгоритма с экспоненциальной (2n) до линейной (n).