Programowanie
po 1 wykładzie

Andrzej Giniewicz

01.03.2024

W dzisiejszym bloku materiałów po wykładzie krótka notka o mierzeniu czasu algorytmów.

1 Dlaczego nie stoper?

Pomiar czasu działania algorytmu w konkretnych warunkach teoretycznie mógłby być zmierzony stoperem. Niestety, zakładając nawet, że istnieje stoper, umiejący zmierzyć mikrosekundy w sposób dokładny, należy pamiętać, że nasz program nie jest jedyną aplikacją działającą na komputerze. W każdej chwili system operacyjny lub antywirus mogą stwierdzić, że chcą wykonać aktualizację. Przeglądarka może w tym samym czasie odtwarzać film na jakimś portalu. Każda z tych operacji zajmuje czas i komputer dzieli swoje zasoby pomiędzy różne rzeczy działające w tym samym momencie. Z tego powodu zadanie dokładnego zmierzenia czasu z dokładnością do milisekund jest skomplikowane.

Podsumowując, mamy trzy problemy, przez które zmierzenie algorytmu raz nie jest wystarczające:

1. jeśli coś trwa krótko, poniżej 1ms, precyzja pomiaru będzie niewielka,
2. może zdarzyć się, że komputer raz na jakiś czas robi coś innego i pomiar może nie być reprezentatywny,
3. jeśli w kodzie następuje jakaś forma losowania lub interakcja z warunkami zewnętrznymi, trudno ustalić, co tak naprawdę jest czasem wykonania funkcji.

Pierwszy problem związany jest z dokładnością naszego „stopera". Rozwiązaniem, które możemy zastosować, jest zmierzenie czasu większej liczby wykonań funkcji. Jeśli jedno wywołanie trwa $0{,}25ms$ a nasz stoper ma dokładność $0{,}1ms$, wynik to $0{,}3ms$, czyli mamy spory błąd. Wykonując funkcję dziesięć razy, odnotujemy, że wykonanie trwa $10 \cdot 0{,}25ms = 2{,}5ms$, co jesteśmy w stanie odnotować w sposób dokładny stoperem o precyzji $0{,}1ms$. Następnie wiedząc, że było to dziesięć pomiarów, możemy wynik podzielić przez liczbę wywołań funkcji. Ten pomysł powinniśmy stosować tylko do kodu, który trwa bardzo krótko, ponieważ

jeśli czas działania jest znacznie dłuższy niż precyzja pomiarów, problem nie występuje i stracimy czas na niepotrzebne, wielokrotne obliczenia czasu działania kodu. W szczególności, jeśli działanie kodu trwa godzinę, nie ma sensu liczyć go dziesięć razy, ponieważ godzina będzie zmierzona bardzo dokładnie na stoperze dowolnej rozsądnej precyzji.

Drugi problem można tak naprawdę rozwiązać tylko, jeśli nie zachodzi trzeci. Jeśli mierzymy czas wykonania kodu, który nie zależy od losowania, interakcji z użytkownikiem, nie pobiera nic z Internetu i ogólnie, jesteśmy pewni, że przy każdym wywołaniu funkcji powinien trwać tyle samo, możemy policzyć działanie kodu wielokrotnie i wziąć najkrótszy czas działania. Wybieramy najkrótszy czas, ponieważ z założenia wiemy, że nasz kod zawsze trwa tyle samo a wszelkie opóźnienia są spowodowane „wtrącaniem się" innych programów. Wobec tego najkrótszy czas ma największą szansę być czasem działania naszego kodu, obranym ze wszystkich zakłóceń, na które nie mamy wpływu. Niestety, jeśli w algorytmie następuje losowanie, nie możemy już wprost użyć minimum — w takim przypadku mierzylibyśmy optymistyczny czas wykonania algorytmu, który może być dużo niższy niż przeciętny.

Spróbujemy teraz zaimplementować funkcję mierzącą czas działania kodu, która powinna działać w możliwie największej liczbie sytuacji. Postaramy się również zadbać o to, aby dobrze zmierzyć funkcje wykorzystujące losowanie.

2 Mała precyzja pomiaru

Przygotujemy funkcję, która będzie mierzyć czas dla większej liczby wywołań na raz i uśredni ten czas. Dzięki temu, jeśli zmierzymy coś 100 razy i podzielimy czas przez 100, otrzymamy oszacowanie czasu jednego wykonania z lepszą precyzją, niż gdy zmierzymy to raz. Czy jednak 100 to najlepsza liczba? Jeśli coś trwa długo, np.: sekundę, nie ma problemu z precyzją pomiaru, a niepotrzebnie będziemy na wynik czekać ponad minutę. Aby temu zaradzić, zastosujemy metodę adaptacyjną. Zmierzymy czas 1 raz, jeśli będzie to trwało zbyt krótko, zmierzymy czas kolejnych 9 razy (w sumie 10). Jeśli to wciąż będzie za mało, zmierzymy czas kolejnych 90 razy (w sumie 100) — i tak dalej.

Do wykonywania wielu pomiarów moglibyśmy użyć pętli z `range` jak na wykładzie, jednak użyjemy `repeat` z modułu `itertools`, ponieważ jest szybszy, a nie zależy nam na wiedzy, w którym z wielu wywołań jesteśmy. Dodatkowo podczas mierzenia małych fragmentów czasu wyłączymy garbage collector, abyśmy mieli pewność, że Python nie będzie się w tym czasie wtrącał w nasze obliczenia — pozwolimy mu usunąć zbędne obiekty dopiero po zakończeniu pomiaru czasu. Nasza funkcja będzie pobierać funkcję `f` (bezargumentową), którą mierzymy, oraz minimalny czas w sekundach, który uznajemy, że jesteśmy w stanie dobrze zmierzyć.

Na początek inicjalizujemy zmienne oraz zapamiętujemy stan garbage collectora, aby przywrócić go po zakończeniu pętli. Pętla działa tak długo, aż suma czasu wykonania nie przekroczy ustalonego limitu. Początkowo uruchamiamy pętlę raz, potem wielokrotnie w pętli. Zwróćmy uwagę, że pomiędzy zmiennymi `start` i `stop` jest absolutne minimum kodu potrzebnego do wykonania pomiaru, aby nie zacierać czasu potrzebnego do wykonania programu. Na koniec zwracamy uśredniony czas działania jednego wykonania.

```
from time import perf_counter
from itertools import repeat
import gc

def zmierz_raz(f, min_time=0.2):
    czas = 0
    ile_razy = 0
    ile_teraz = 1
    stan_gc = gc.isenabled()
    gc.disable()
    while czas < min_time:
        if ile_teraz == 1:
            start = perf_counter()
            f()
            stop = perf_counter()
        else:
            iterator = repeat(None, ile_teraz)
            start = perf_counter()
            for _ in iterator:
                f()
            stop = perf_counter()
        czas = stop-start
        ile_teraz *= 2
    if stan_gc:
        gc.enable()
    return czas/ile_teraz
```

Zobaczmy jak uruchomić kod. Pierwszy argument to funkcja bezargumentowa, którą będziemy mierzyć. Drugi argument jest opcjonalny i zawiera ilość czasu, który musi upłynąć, abyśmy uznali, że precyzja pomiaru jest wystarczająca. Przykładowe wywołanie mierzące czas wywołania funkcji wbudowanej `max` znajduje się poniżej. Zwróćmy uwagę na wyrażenie lambda, aby kod „zamknąć w funkcji bezargumentowej".

```
x = [1, 2, 1, 4, 1, 5, 1, 6, 1, 7, 7, 6, 45, 8, 2, 10]

zmierz_raz(lambda: max(x))
```

Dzięki takiemu zabiegowi rozwiązujemy problem z precyzją pomiaru, jednocześnie licząc czas tylko raz, jeśli funkcja liczy się długo.

3 Nietypowe pomiary czasu

Niestety wciąż nie rozwiązuje to problemu nietypowego czasu. Wciąż antywirus lub przeglądarka lub inna aplikacja może nam przeszkodzić. Wykonamy kilka pomiarów i weźmiemy minimum ich czasów. Aby zabezpieczyć się przed tym, że funkcja może wykonywać

losowanie, musimy dowiedzieć się, jak działają liczby „losowe" w komputerze. W rzeczywistości w komputerze losowość nie występuje, a liczby, które otrzymujemy, są tak zwanymi liczbami pseudo-losowymi. Są one wyliczane przez generatory liczb pseudo-losowych w ten sposób, aby przypominały losowe. W tym celu większość generatorów wykorzystuje koncepcję ziarna. Jeśli ustawimy takie samo ziarno, które jest stanem generatora liczb pseudo-losowych, otrzymamy ten sam zestaw liczb. W ten sposób, mierząc funkcję kilkukrotnie, ustawiając to samo ziarno tuż przed jej wykonaniem, upewnimy się, że wewnątrz „wylosują się" te same liczby, czyli każde wykonanie funkcji będzie trwało tyle samo czasu.

Przygotujmy funkcję, która wykona serię pomiarów, ustawiając identyczne ziarno przy każdym przejściu pętli. Na koniec, zwracamy minimalny czas działania. Podobnie jak w przypadku garbage collectora, zapamiętujemy stan generatora liczb losowych, aby przywrócić go po zakończeniu pomiarów.

```
from random import seed, randrange, setstate, getstate

def zmierz_min(f, serie_min=5, min_time=0.2):
    pomiary = []
    generator = getstate()
    seed()
    my_seed = randrange(1000)
    for _ in repeat(None, serie_min):
        seed(my_seed)
        pomiary.append(zmierz_raz(f, min_time=min_time))
    setstate(generator)
    return min(pomiary)
```

Oczywiście pomiar ten odpowiada jedynie za zmierzenie fizycznie jednego wariantu, który mógł być wariantem optymistycznym, ponieważ ustawiamy to samo ziarno — a co za tym idzie, wylosują się za każdym razem dokładnie te same liczby. Jeśli więc chcemy uwzględnić fakt, że pomiary mogą zależeć od wewnętrznie wylosowanych liczb, musimy w jakiś sposób uwzględnić wiele wywołań funkcji `zmierz_min`.

Poradzimy sobie z tym, mierząc czas wiele razy, na przykład 10, i zwracając medianę tych czasów. Mediana to wartość, od której 50% obserwacji jest mniejszych i 50% obserwacji większych, jest więc obserwacją w pewnym sensie „przeciętną". Jeśli mierzymy czas nieparzystą liczbę razy, nie ma problemu z wyznaczeniem wartości w środku. Jeśli mamy parzystą liczbę pomiarów, wartość „w środku" będzie średnią arytmetyczną dwóch środkowych liczb.

```
def zmierz(f, serie_median=10, serie_min=5, min_time=0.2):
    pomiary = []
    for _ in repeat(None, serie_median):
        pomiary.append(zmierz_min(f, serie_min=serie_min, min_time=min_time))
    pomiary.sort()
    if serie_median%2==0:
        return (pomiary[serie_median//2-1]+pomiary[serie_median//2])/2
```

```
    else:
        return pomiary[serie_median//2]
```

Oczywiście, wprowadzenie takiego sposobu obliczeń prowadzi do znacznego wydłużenia procedury mierzenia czasu, ale tym samym pozwala na precyzyjne pomiary i uniknięcie wielu możliwych problemów prowadzących do nieoczekiwanych rezultatów. Przykład wywołania całej funkcji znajduje się poniżej.

```
x = [1, 2, 1, 4, 1, 5, 1, 6, 1, 7, 7, 6, 45, 8, 2, 10]

zmierz(lambda: max(x))
```