Asymptotic runtime testing

November 25, 2020 Christopher Skane <chrisk3@umbc.edu>

Contents

1 Introduction

The goal of this is to provide explanation and demonstration of how one can approach the testing of the asymptotic efficiency of code. I will discuss a general approach to help determine the nature of the test, then go over many of the common efficiencies with examples.

2 Methods

2.1 General

The primary method I will discuss deals with choosing a scalar $\alpha \neq 0$, for some fixed N, and finding the sequence of ratios of the elapsed runtime.

Suppose we want to test the asymptotic runtime of a function f, which we expect to have an asymptotic efficiency of $O(g(N))$. Then let N be the number of times we call this function in a given trial. We should then expect the runtime of these N calls to have an efficiency of $O(N \cdot g(N))$. Let us define a function $t(N) := N \cdot g(N)$, which will represent the ideal time of these N function calls; ideal here means every operation takes exactly $g(N)$ units of time. Lastly, we define a sequence (β_k) such that

$$\beta_k = \frac{t(\alpha^k N)}{t(\alpha^{k-1} N)}, \quad k = 1, \ldots, M$$

where M is the total number of trials run. It may also be viable to instead define the sequence (β_k) as

$$\beta_k = t(N + \alpha k) - t(N + \alpha(k-1)), \quad k = 1, \ldots, M$$

with a linearly scaling value. A linear scaling value is only really useful if $t(N)$ is a linear map (read "function"), since you can separate the sums and scalars.

The goal is to give us a method to accurately estimate some β_{k+1}, given the sequence $\{\beta_1, \ldots, \beta_k\}$, through simple means.

2.2 Numeric

[To be completed. Deals with curve fitting]

3 Efficiencies

3.1 $O(1)$ - Constant

3.1.1 Math

Using the notation introduced in 2.1, we have some function, f, which we expect to have $O(1)$ (i.e constant) efficiency. So our time function $t(N) = N \cdot 1$. So the sequence (β_k) becomes

$$\beta_k = \frac{(\alpha^k N) \cdot 1}{(\alpha^{k-1} N) \cdot 1} = \alpha$$

in other words, we should expect a constant sequence as the total operation count rises.

Since $t(N)$ is linear, we could also have (β_k) be

$$\beta_k = (N + \alpha k) \cdot 1 - (N + \alpha(k-1)) \cdot 1 = \alpha$$

which is the same constant sequence as above.

Given that both sequences are equivalent, it would be preferrable to use the latter sequence to avoid exponentially large N values. The caviat being that the sequence may not be close to the exact value α, but will still be a constant sequence.

3.1.2 Pseudo-code

```
M := 5
N := 100
a := 10
T := [1..M] of double

//#%— Loop over M trials, collecting the times
for k = 1..M do
    start := time()

    for i = 1..N do
        fn(i)
    end

    stop := time()

    T[t] := stop - start
    N := N + a
end

//#%— Check that the difference in times is (nearly) constant
B := [1..(M-1)] of double
for k = 1..(M-1) do
    B[k] := T[k+1] - T[k]
    print(B[k])
end
```

3.2 $O(n)$ - linear

Using the notation introduced in 2.1, we expect our function f to have $O(n)$ efficiency. So our time function $t(N) = N \cdot N = N^2$. So the sequence (β_k) becomes

$$\beta_k = \frac{\alpha^{2k} N^2}{\alpha^{2(k-1)} N^2} = \alpha^2$$

and similar to above, we expect a constant sequence with values that are roughly α^2.

3.3 $O(n^p)$ - Generalized exponent

You often won't be testing much beyond linear, since $O(n^2)$ is not too frequent; it also sucks to test $O(n^2)$ because you'll have an $O(n^3)$ timing loop! But for completeness, and because it is simple, we can generalize the above cases.

We take our function, f, which we expect to have $O(n^p)$ efficiency. So our time function $t(N) = N \cdot N^p = N^{p+1}$. This makes the sequence (β_k)

$$\beta_k = \frac{\alpha^{(p+1)k} N^{p+1}}{\alpha^{(p+1)(k-1)} N^{p+1}} = \alpha^{p+1}$$

and like before, it is a constant sequence of some power of α.

WARNING: The total wall time for $p \geq 2$ can grow *very* rapidly, well into multiple minutes. You usually do not need to run for that long to determine a pattern. If you are testing anything $p \geq 3$ or larger, either the problem is inherently inefficient or you messed up badly; do your research to eliminate the former, as someone way smarter than you has probably figured it out.

3.4 $O(\log_b n)$ - Logarithmic

As before, we have f which we expect to have $O(\log_b n)$ efficiency. The time function is then $t(N) = N \cdot \log_b N$. So the sequence (β_k) is

$$\beta_k = \frac{(\alpha^k N) \log_b(\alpha^k N)}{(\alpha^{k-1} N) \log_b(\alpha^{k-1} N)}$$

$$\implies \beta_k = \alpha \frac{\log_b(\alpha^k) + \log_b(N)}{\log_b(\alpha^{k-1}) + \log_b(N)}$$

$$\implies \beta_k = \alpha \frac{k \log_b(\alpha) + \log_b(N)}{(k-1) \log_b(\alpha) + \log_b(N)}$$

$$\implies \beta_k = \alpha \frac{k}{k-1}$$

and in order to make our lives easier, we can just take the limit of β_k as $k \to \infty$ to get

$$\beta_k \to \alpha$$

so we can expect $\beta_k \approx \alpha$, which can likely just be further simplified to $\beta_k = \alpha$.

3.5 $O(n^p \log_b n)$ - Polynomial logarithmic

Again, the generalization for completeness. We have f which we expect to have $O(n^p \log_b n)$ efficiency. Then the time function is $t(N) = N \cdot N^p \log_b N = N^{p+1} \log N$. So the sequence (β_k) is

$$\beta_k = \frac{(\alpha^{(p+1)k} N^{p+1}) \log_b(\alpha^k N)}{(\alpha^{(p+1)(k-1)} N^{p+1}) \log_b(\alpha^{k-1} N)}$$

$$\implies \beta_k = \alpha^{p+1} \frac{\log_b(\alpha^k) + \log_b(N)}{\log_b(\alpha^{k-1}) + \log_b(N)}$$

$$\implies \beta_k = \alpha^{p+1} \frac{k}{k-1}$$

and just like before we can expect $\beta_k \approx \alpha$, which we will simplify to $\beta_k = \alpha^{p+1}$.

3.6 $O(n^p\sqrt{n})$ - Polynomial root

To preface, this case is encountered less than the above ones, but the pattern with this document is completeness. I'll skip the $p = 0$ case since it's easier to do the general case.

We have our f which we expect to be $O(n^p\sqrt{n})$, and so the time function becomes $t(N) = N \cdot N^p\sqrt{N} = N^{p+1}\sqrt{N}$. Then the sequence (β_k) is

$$\beta_k = \frac{(\alpha^{(p+1)k}N^{p+1})\sqrt{\alpha^k N}}{(\alpha^{(p+1)(k-1)}N^{p+1})\sqrt{\alpha^{k-1}N}}$$

$$\implies \beta_k = \alpha^{p+1}\sqrt{\frac{\alpha^k N}{\alpha^{k-1}N}}$$

$$\implies \beta_k = \alpha^{p+1}\sqrt{\alpha}$$

so in order to have nicer numbers, favor square numbers over others for α.

4 Code

Unless stated otherwise, using $\alpha \in [2, 4]$ is probably good enough for timing while also keeping the total wall time down. For anything logarithmic, choose α to be the base of the logarithm for nicer constants.

4.1 Pseudo

If you wish to automate the checking of the values in β_k (the array B in the code), I have found that the mean and standard deviation could be used to verify a **constant sequence**. For this, you will want some ϵ such that $|\mu - E| < \epsilon$; here μ is the standard mean, and E represents the expected value, as determined by the above math. You ideally want the deviation to be small, in order to verify that the values are actually clustered close to the mean; you could use $\epsilon/2$, or any other value, for this purpose.

```
M := 5
N := 100
a := 2
T := [1..M] of double

//#%-- Loop over M trials, collecting the times
S := N
for k = 1..M do
    start := time()

    for i = 1..S do
        fn(i)
    end

    stop := time()

    T[t] := stop - start
    S := N * a^(k+1)
end
```

```

//#%-- Check the ratio of times
B := [1..(M-1)] of double
for k = 1..(M-1) do
    B[k] := T[k+1] / T[k]
    print(B[k])
end
```

4.2 C++

```cpp
#include<ctime>
#include<cmath>
#include<iostream>

using std::cout;
using std::endl;

// O(1) function
long con(long x){
    return x*x - 2*x + 1.0; // Help avoid minor optimization
}

// O(n) function
long lin(long x){
    long total = 0;
    for(long i = 0;i < x;i++){
        total += 2*i; // Not just i, to hopefully avoid optimization
    }
    return total;
}

// O(log_2 n) function
long logn(long x){
    double s = x;
    long c = 0;
    while(s > 1.0){
        s /= 2;
        c += 1;
    }
    return c;
}

// Arithmetic mean
double mean(double *arr, long n){
    if(arr == nullptr || n == 0) return NAN;

    double total = 0.0;
    for(long i = 0;i < n;i++){
        total += arr[i];
    }
    return total / n;
}

// Standard deviation
```

```cpp
double stdev(double *arr, long n){
    if(arr == nullptr || n == 0) return NAN;

    double mu = mean(arr, n);
    double total = 0.0;
    for(long i = 0;i < n;i++){
        total += (mu - arr[i]) * (mu - arr[i]);
    }
    return sqrt(total / n);
}

int main(){
    const long M = 5; // Number of trials
    const long N = 1000; // Iterations per trial
    const long a = 2; // Scaling value
    double T[M]; // Result time array

    long (*fn)(long) = logn; // function to test
    // a^2 for linear, ~a for logarithmic, and a for constant
    double expect = a;

    /* Trial loop
    */
    long S = N; // Scratch variable to leave N unchanged
    long res;
    long a_it = a; // Used to avoid calling pow()
    clock_t start,stop;
    for(long k = 0;k < M;k++){
        cout << "Trial " << k+1 << " with N = " << S << endl;

        // Call loop
        start = clock();
        for(long i = 0;i < S;i++){
            res = fn(i);
        }
        stop = clock();

        // Get results
        T[k] = (stop - start);

        // Iterate things
        S = N * a_it;
        a_it *= a;
    }

    // T array dump
    cout << "T =" << endl;
    for(long i = 0;i < M;i++){
        cout << "  " << T[i] << endl;
    }

    // Calculate ratios
    double B[M-1];
    for(long k = 0;k < (M-1);k++){
        B[k] = (T[k+1] / T[k]);
    }
```

```

    // B array dump
    cout << "B =" << endl;
    for(long i = 0;i < M-1;i++){
        cout << "  " << B[i] << endl;
    }

    // Calculate the suggested values
    double mu = mean(B, M-1);
    double dev = stdev(B, M-1);
    cout << "Mean = " << mu << endl;
    cout << "Std dev = " << dev << endl;


    // Values subject to vary (wildly) depending on system and test
    double mu_eps = 0.5;
    double dev_eps = mu_eps;
    bool pass = abs(mu - expect) < mu_eps && dev < dev_eps;

    cout << "Test " << ((pass) ? "succeeded!" : "FAILED!") << endl;
}
```

4.3 Python

```
import numpy as np
from time import time

def const(x):
    return x*x - 2*x + 1 # Help avoid minor optimization

def lin(x):
    total = 0
    for i in range(x):
        total += 2*i # Not just i, to hopefully avoid optimization out
    return total

def logn(x):
    s = x
    c = 0
    while s > 1.0:
        s /= 2
        c += 1
    return c


# constant - N=16000 and a=4 is close to stdev below 0.1
# lin - N=500 and a=3 works, but is slow
# logn - N=4000 and a=3 gives consistent stdev below 0.1
M = 5
N = 500
a = 3
T = []
fn = logn

S = N
for k in range(M):
    print("Trial", k+1)
    start = time()

    for i in range(S):
        res = fn(i)

    stop = time()

    T.append(stop - start)
    S = N * a**(k+1)

print(T)

B = [(T[k+1] / T[k]) for k in range(M-1)]
print(B)

arr = np.array(B)
dev = np.std(arr)
mean = np.mean(arr)
print("Mean =", mean)
print("Std dev =", dev)
```