Unit 22: Efficiency

Learning Objectives

After completing this unit, students should:

- be aware that efficiency translates to less (and hopefully minimal) computation and space being used by a program
- Understand that increasing efficiency typically requires that there are no redundancies and no duplication (i.e., not repeat work already done)
- understand that when analysing efficiency, we wish to bound the computation required and thus look at the worst-case scenario
- understand that Big-O bounds capture the highest order of growth for some algorithm/code
- be able to determine the Big-O bounds for most code used in CS1010
- be able to compare the efficiency between code using Big-O analysis

No Redundant Work

We have been asking you to write efficient code, but we have not been very formal about quantifying what is efficiency. In this unit, we will recap the various good habits to write efficient code, and introduces the notion of *time complexity* of a code, using the Big-O notation.

Writing efficient code basically means that we should not write code that runs unnecessarily. Let's consider the following two versions of `is_prime` function.

```
bool is_prime(long n)
{
  bool is_prime = true;
  for (long i = 2; i <= n - 1; i += 1) {
    if (n % i == 0) {
      is_prime = false;
    }
  }
  return is_prime;
}
```

```
1
```

```
bool is_prime(long n)
{
  for (long i = 2; i <= sqrt(n); i += 1) {
    if (n % i == 0) {
      return false;
    }
  }
  return true;
}
```

Both versions are correct -- they produces `true` when the input `n` is prime, and `false` otherwise. But they take a vastly different amount of time to run. On my machine, the slowest version of `is_prime` took ~100s, while the fast one runs ~3ms when invoked with `is_prime(10000000001)`. The time taken is five orders of magnitude difference!

The second version of `is_prime` is faster as it follows the following mantra:

"No redundant work"

We came across this when we discussed short-circuiting earlier -- we want to order our logical expression so that we don't do extra, unnecessary, work.

Here, we improve the efficiency of `is_prime` by (i) returning `false` as soon as we found a "proof" that the input is not a prime, and (ii) not checking for divisor that is redundant.

Worst Case Performance

The two techniques we employed above behave slightly differently. For (i), we *opportunistically* stop our computation once we know the answer when the input is not a prime. However, in the *worst case*, when the input is a prime, we still have to check through all $n - 2$ divisor, from 2 up to $n - 1$, before we can conclude that the input is a prime. Thus, for (i), we make our program faster for certain inputs but in the worst case, we are not able to speed up the program.

The second technique is more interesting and more fruitful. With a little math, we can show that we only need to check for the divisor up to $\sqrt{n}$. Here, we are improving the *worst case* performance of `is_prime`, so whether the input is a prime or not, we always have a speed up.

How much is the speed up in the worst case? In the input above, with the original slow code, in the worst case I need to check through ~10,000,000,001 divisors. With the faster version, I only need to check ~$\sqrt{10,000,000,001}$ = ~10,000 divisors. That's where the five orders of magnitude speed up comes from!

No Duplication

The second principle to improving the efficiency of the program is that "*No duplication*" -- we should not repeat work that has been done before.

Let's look at the example from earlier this semester, where you are asked to compute the range of a list. Recall that the range of a list is the absolute difference between the largest element and the smallest element. We use this problem to motivate the use of function, where we denote $range(L, k) = |max(L, k) - min(L, k)|$. If we are to implement this solution in C, however, we would end up scanning through the list twice: first to find the max, then to find the min.

Problem 17.1 shows that we could easily just go through the list once, use call-by-reference to output both the max and the min if we are willing to forgo the notion of pure functions and function-as-a-black-box.

The performance improvement for not scanning through a list twice is modest at most. Let's look at another example where, by not repeating ourselves, we can gain significant performance improvement.

Finding Fibonacci Numbers

Recall from your Exercise 2, the Fibonacci sequence of numbers are numbers formed by adding the previous two numbers to form the next numbers. In other words, the i-th Fibonacci number is computed as the sum of the previous two Fibonacci numbers, the $(i-2)$-th, and the $(i-1)$-th.

Here is one way we can find the Fibonacci numbers with a loop:

```
1  long fib(long n)
2  {
3    if (n == 1 || n == 2) {
4      return 1;
5    }
6    long first = 1;
7    long second = 1;
8    long third = 1;
9    for (long i = 2; i != n; i += 1) {
10     first = second;
11     second = third;
12     third = first + second;
13   }
14   return third;
15 }
```

To find the n-th Fibonacci number, we take n steps in a loop.

n-2 steps for loop only

Now, let's see the following elegant recursive solution:

```
long fib(long n)
{
  if (n == 1 || n == 2) {
    return 1;
  }
  return fib(n-1) + fib(n-2);
}
```

This solution is short and follows directly from the definition of Fibonacci numbers. Running this, however, reveals something very disturbing. Let's say we call `fib(n)`. This invocation in turns calls `fib(n-1)` and `fib(n-2)`. `fib(n-1)` then calls `fib(n-2)` and `fib(n-3)`. So, `fib(n-2)` will be called twice -- so we are repeating ourselves, violating the no duplication principles. In fact, we will invoke `fib()` a large number of times.

Big-O Notation

How do we characterize the number of times `fib()` is called? To answer this, we will introduce to you the Big-O notation.

The Big-O function is a mathematical function that computer scientists use to characterize the time and space efficiency of an algorithm. In CS1010, we will not go into the formal definition of the notation, but will introduce Big-O using the intuition that it is the "rate of growth" of a function.

> **Formal definition**
>
> For those mathematically inclined students who can't wait until CS2040C or CS3230 for a formal definition, here it is:
>
> Given two functions f and g, we say that $f(x) = O(g(x))$ if the exists a position real number c and a real number x_0 such that
>
> $$|f(x)| \leq cg(x), \ \forall x \geq x_0$$

To motivate Big-O, let's consider how we can count the number of "steps" taken by an algorithm. Let's consider this again:

```
  for (long i = 2; i != n; i += 1) {
    first = second;
    second = third;
```

```
4       third = first + second;
5     }
```

If we consider each of the fundamental operations: comparison, addition, and assignment, as a step, then we can see that, in each loop, there is one comparison (`i != n`), two additions (`i + 1`, `third = first + second`), four assignments. So we have seven operations per loop, with a total of $n - 1$ loops. So we have $7n - 7$ operations. In addition, we also need to count for the assignment `i = 2` and the additional comparison before we exit the loop (`i != n`). So, in total, we have $7n - 5$ operations.

As you can see, such detailed counting of the steps in tedious, and in fact, not very meaningful. For instance, we did not account for reading values from the memory and writing of values into the memory, the performance of which becomes dependant on the architecture underneath.

To free us from such low-level accounting of the number of steps, let's focus on the big picture. No matter how we count the number of steps, in the end, it is a linear function of n. In order words, the number of steps taken by the algorithm to compute Fibonacci in a loop grows linearly with n. Using the Big-O notation, we say that it takes $O(n)$ steps.

Given a mathematical function with multiple terms, the Big-O of this function is obtained by dropping any multiplicative constants and all terms, except for the one with the *highest rate of growth*. For instance, $O(\frac{n^4}{10} + 10000n^2 - n) = O(n^4)$.

Due to this focus on the term with highest rate of growth, and not bothering about other terms or multiplicative constants, it becomes very convenient for us to expression the time efficiency of an algorithm with $O()$ -- we no longer need to count the steps precisely but just focus on the number of times it takes to run the algorithm in terms of n.

Take the example of `is_prime`. The slow algorithm takes $O(n)$, the fast algorithm takes $O(\sqrt{n})$.

Take another example: to find the range of a list, both algorithms, regardless of whether we are taking two passes or one pass, take $O(n)$ time.

Comparing Rate of Growth

Given two functions $f(n)$ and $g(n)$, how do we determine which one has a higher rate of growth? We say that $f(n)$ grows faster than $g(n)$ if we can find a n_0, such that $f(n) > cg(n)$ for all $n \geq n_0$ and for some constant c.

For instance, which one grows faster? $f(n) = n^n$ or $g(n) = 2^n$? Pick $n = 1$, we have $f(1) < g(1)$[1]. Pick $n = 2$, we have $f(2)$ equals $g(2)$. Pick $n = 3$, we have $f(3) > g(3)$ now,

and we can see that for any $n > 3$, $n^n > 2^n$, so we can conclude that $f(n)$ grows faster than $g(n)$.

Running Time or Time Complexity of an Algorithm

Using the big-O notation, we can quantify how many steps an algorithm takes to run. This measurement is called the *running time* or the *time complexity* of the algorithm.

In CS1010, when we express the running time, we are interested in the worst case performance, or worst case running time. Computer scientists also find measuring the average running time useful in certain scenarios. You will learn about about analyzing the average running time in CS3230.

There are a couple of things to note when expressing the running of an algorithm in big-O notation:

- We should express it in its simplest form -- without multiplicative constants, and dropping all lower terms.
- Since we are specifying the worst case performance, we are measuring an upper bound on the running time. We should expression an upper bound that is as tight as possible.

Example: Kendall

Now, consider the algorithm to compute the Kendall-Tau distance. The code is given below.

```
1  long count_inversion(long i, long n, const long rank[])
2  {
3    long count = 0;          n-2 step
4    for (long j = i + 1; j < n; j += 1) {
5      if (rank[i] > rank[j]) {
6        count ++;
7      }
8    }
9    return count;
10 }
11
12 double kendall_tau(long n, const long rank[])
13 {
14   long count = 0;          n-1 step
15   for (long i = 0; i < n - 1; i += 1) {
16     count += count_inversion(i, n, rank);
17   }
18   return 2.0 * count/(n * (n - 1));
19 }
```

O(n)

O(n²)

need to add count-inversion efficiency here also

What is the running time of the function `kendall_tau`, expressed in Big-O notation in terms of n, the length of the input array?

To analyze the running time, we focus on the big picture, the part of the code that takes the most time, and we ignore all the other operations that take negligible time. In the function `kendall_tau` above, Line 16 is repeated many times, so let's focus on that. Looking at the looping conditions, we can conclude that Line 16 repeats $O(n)$ times.

It is tempting to conclude that `kentall_tau` takes $O(n)$ steps here, but it would be wrong.

Notice that Line 16 calls another function `count_inversion`. What is the running time of `count_inversion`? Inside `count_inversion`, there is another loop that repeats $n - i$ times. Each time we call `count_inversion`, `i` increases, so the loop in `count_inversion` takes fewer steps each time the function is called.

To calculate the total number of steps, we can compute the following sum $\sum_{i=0}^{n}(n-i)$, which is just $n + (n-1) + (n-2) + .. + 2 + 1$. This sum is the sum of an arithmetic series and equals to $n(n+1)/2$. Since we use the Big-O notation, we can focus on the term with the highest rate of growth, n^2, and ignore everything else. We have obtained the running time for `kentall_tau` function above as $O(n^2)$.

Example: Fibonacci

Let's get back to the recursive Fibonacci number example. What is the running time of `fib`?

```
1  long fib(long n)
2  {
3    if (n == 1 || n == 2) {
4      return 1;
5    }
6    return fib(n-1) + fib(n-2);
7  }
```

Answering this question is trickier since this is a recursive function. But, we can use the same "wishful thinking" approach we used when writing this code to analyze its running time.

Let $T(n)$ is the running time of `fib(n)`. Since `fib(n)` calls `fib(n-1)` and `fib(n-2)`, we can write $T(n) = T(n-1) + T(n-2)$. Just like the recursive function there is a base case: $T(n) = O(1)$ if $n \leq 2$.

Note that $O(1)$ is also known as constant running time[1].

The equation for $T(n)$ above is known as a *recurrence relation*. There are standard techniques to solve it (search for Master Theorem on the web), but we won't go into them in CS1010. We will solve it from the first principle, by expanding the equation and observing the pattern.

We know that $T(n-1) > T(n-2)$. So,

$$T(n-2) + T(n-2) < T(n) < T(n-1) + T(n-1),$$

therefore,

$$2T(n-2) < T(n) < 2T(n-1).$$

So

$$T(n) < 2T(n-1) < 4T(n-2) < 8T(n-3) < .. 2^{n-1}T(1) < O(\frac{1}{2}2^n)$$

and

$$T(n) > 2T(n-2) > 4T(n-3) > 8T(n-4) > .. 2^{n-2}T(1) > O(\frac{1}{4}2^n)$$

Since we can drop the multiplicative constant in Big-O notation, we have $T(n) = O(2^n)$.

This explains why the recursive solution to Fibonacci is so much slower than the iterative version, which is $O(n)$ -- one grows exponentially while the other linearly in terms of n.

Example: Collatz

Not every program has a running time that is easy to analyze. For instance, take `collatz`

```
long count_num_of_steps(long n)
{
  long num_of_steps = 0;
  while (n != 1) {
    n = collatz(n);
    num_of_steps += 1;
  }
  return num_of_steps;
}
```

Recall that this program is based on the Collatz conjecture, which has not been proven. So we cannot say that the loop above will always terminate after a given number of steps. The running time for the code above remains unknown.

Space Efficiency

We have focused mostly on efficiency in terms of time in this unit. Note that the notion of efficiency extends to space (memory usage) as well. Big-O notation can be used to quantify how much space is used as well.

Problem Set 22

Problem 22.1

Order the following functions in the increasing order of rate of growth:

- $n!$,
- 2^n,
- $\log_{10} n$,
- $\ln n$,
- n^4,
- $n \ln n$,
- n,
- n^2,
- e^n,
- $\sqrt{n}$

Problem 22.2

What is the Big-O running time of the following code, in terms of n?

a)

```
for (long i = 0; i < n; i += 1) {
  for (long j = 0; j < n; j += 2) {
    cs1010_println_long(i + j);
  }
}
```

b)

```
for (long i = 1; i < n; i *= 2) {
  for (long j = 1; j < n; j *= 2) {
    cs1010_println_long(i + j);
  }
}
```

c)

```
long k = 1;
for (long j = 0; j < n; j += 1) {
  k *= 2;
  for (long i = 0; i < k; i += 1) {
    cs1010_println_long(i + j);
  }
}
```

---

1. The 1 in $O(1)$ has nothing to do with the function returning `1`. But we write it this way because we can drop multiplicative constants in the big-O notation. ↩↩