CSC165H1: Mathematical Expression and Reasoning for Computer Science **Fall 2018**

Lecture 16: November 13

Lecturer: Tom Fairgrieve *Note-Taker: Jim Gao (jim.gao@mail.utoronto.ca)*

16.1 Example: Collatz Sequence

Since there is no proof that the sequence converges to 1, there would not be an upper bound for the sequence.

16.1.1 Modified Collatz Sequence

We define it as follows:

```
def f(n: int) -> int:
        steps = 0
        x = n
        while x > 1:
                if x % 2 == 0:
                        x = x // 2
                else:
                        x = 2 * x - 2
                steps += 1
        return steps
```

Our goal is to find $h_1(n)$, $h_2(n)$ such that:

- $RT_f(n) \in \mathcal{O}(h_1)$, and
- $RT_f(n) \in \Omega(h_2)$

By attempting some values of x, we can begin to see patterns for the algorithm:

- When x is a power of 2, x stays even
- When x is odd, we get to $(x-1)$ in 2 iterations: $x \to (2x-2) \to (x-1)$
- When x is even, $x \to x/2$. We know that $x/2$ is either even or odd.

It is important to see that, in either case, x decreases by at least 1 in 2 iterations.

Assuming the worse (i.e. x decreases by 1 per 2 iterations), the total number of iterations is less than or equal to $2(x-1)$.

We can therefore conclude that the runtime is in $\mathcal{O}(n)$.

In addition, by the first observation, the best possible case is when x is a power of 2. In that case, the number of iterations is equal to $\log_2 x$. Hence, the runtime is in $\Omega(n)$.

16.1.1.1 Writing a Formal Proof

The proof for upper bound is trivial.

For the proof for the lower bound, we claim that, after k iterations, x becomes $\frac{x}{2^k}$. As the algorithm terminates when $x \leq 1$, we need to find the greatest k such that:

$$\frac{n}{2^k} \leq 1$$

16.1.1.2 Additional Notes

- It is possible to tighten the upper bound for this example.
- In the general case, the upper bound is at the tightest when it reaches the lower bound.

16.2 Example: HasEven

Consider the following function:

```
def has_even(numbers: List[int]) -> bool:
        for n in numbers:
                if n % 2 == 0:
                        return True
        return False
```

The major qualitative difference of the above code is that, the runtime not only depends on the size of the list, but also depends on the content of the list.

We can define the worst case runtime: $\text{WC}_f(n)$, and the best case runtime: $\text{BC}_f(n)$.

For this case, we know $\text{WC}_f(n) \in \mathcal{O}(n)$, since the maximum number of iterations does not exceed the size of the list. We also know that $\text{WC}_f(n) \in \Omega(n)$.