CSC110 Lecture 23: More Running-Time Analysis

David Liu and Tom Fairgrieve, Department of Computer Science

*Navigation tip for web slides: press **?** to see keyboard navigation controls.*

Announcements, Reminders and Today's Plan

Announcements

- Assignment 4 has been posted
  - Due in 9 days!
  - Check out the A4 FAQ (+ corrections)
  - Additional TA office hours start today!
  - Review advice on academic integrity
- The Final Exam schedule has been posted. Report any Exam Conflicts ASAP!

Before reading week ...

- learned to describe the growth of a function $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$
  - using Big-O, Omega and Theta
  - learned a few properties and The Function Growth Hierarcy
- started to analyze the running time of our programs
  - can't predict the required clock time ⌚
  - instead model the running time and describe it's growth
  - can then predict how the running time changes as the input size increases

Today you'll learn to...

1. Analyse the running time of code containing nested loops.
2. Analyse the running time of code containing comprehensions and while loops.
3. Perform running-time analyses by finding bounds on the running-time function (rather than finding an exact expression).

But first, more review.

Example 1 - no iteration

Analyse the running time of the following function.

```
def f(numbers: list[int]) -> bool:
    x = len(numbers) + 1
    y = x * 3
    return len(numbers) + y > 9000
```

Analysis. Let n be the length of the input list `numbers`.

All expressions and statements in the body of `f` are constant time operations, so we count the whole body as 1 step.

So $RT_{\mathtt{f}}(n) = 1$, which is $\Theta(1)$. We say `f` is a constant time function.

Note: Even if we count 3 steps, we still describe as $\Theta(1)$!

Example 2 - simple iteration

```
def print_items(numbers: list[int]) -> None:
    for number in numbers:
        print(number)
```

Analysis. Let n be the length of the input `numbers`.

- The for loop has n iterations
- A single iteration takes 1 step (since calling `print` on a number is constant time)

So the total number of steps is $n \cdot 1 = n$.

The total running time is $RT_{\text{print\_items}}(n) = n$, which is $\Theta(n)$.

Example 3 - simple iteration and more

When you see a mixture of constant-time and non-constant-time statements, calculate the number of steps for each statement separately and add the result.

```
def my_sum(numbers: list[int]) -> int:              # Line 1
    sum_so_far = 0                                  # Line 2
                                                    # Line 3
    for number in numbers:                          # Line 4
        sum_so_far = sum_so_far + number            # Line 5
                                                    # Line 6
    return sum_so_far                               # Line 7
```

Example 3 - simple iteration and more

```python
def my_sum(numbers: list[int]) -> int:               # Line 1
    sum_so_far = 0                                   # Line 2
                                                     # Line 3
    for number in numbers:                           # Line 4
        sum_so_far = sum_so_far + number             # Line 5
                                                     # Line 6
    return sum_so_far                                # Line 7
```

Analysis. Let n be the length of the input list `numbers`.

1. (Line 2) `sum_so_far = 0` takes 1 step (constant time)
2. (Line 4-5) the for loop takes n steps, because:
    - it takes n iterations
    - each iteration takes 1 step (constant time)
3. (Line 7) `return sum_so_far` takes 1 step (constant time)

So the total running time is $1 + n + 1 = n + 2$ steps, which is $\Theta(n)$.

Nested for loops

From Worksheet 22, Exercise 2

```
def f3(numbers: list[int]) -> None:
    for i in range(0, len(numbers) ** 2 + 5):
        for number in numbers:
            print(number * i)
```

When analyzing a nested loop, start with the inner loop first.

```python
def f3(numbers: list[int]) -> None:
    for i in range(0, len(numbers) ** 2 + 5):
        for number in numbers:
            print(number * i)
```

Analysis. Let n be the length of `numbers`.

For the inner loop:

- n iterations
- each iteration takes 1 step
- So, a total of $n \cdot 1 = n$ steps for a fixed iteration of the outer loop

For the outer loop:

- $n^2 + 5$ iterations
- each iteration takes n steps
- A total of $(n^2 + 5) \cdot n = n^3 + 5n$ steps

So the total running time is $RT_{\text{f3}}(n) = n^3 + 5n$, which is $\Theta(n^3)$.

```python
def f4(numbers: list[int]) -> None:
    for i in range(0, len(numbers) ** 2 + 5):
        for j in range(0, i):  # Note the range here!
            print(i + j)
```

Analysis. Let n be the length of `numbers`.

For the inner loop:

- i iterations
- each iteration takes 1 step
- So, a total of $i \cdot 1 = i$ steps for a fixed iteration of the outer loop

For the outer loop:

- $n^2 + 5$ iterations
- each iteration takes… i steps?

```python
def f4(numbers: list[int]) -> None:
    for i in range(0, len(numbers) ** 2 + 5):
        for j in range(0, i):  # Note the range here!
            print(i + j)
```

The total number of steps from all iterations of the outer loop is

$$0 + 1 + 2 + \cdots + (n^2 + 4) = \sum_{i=0}^{n^2+4} i$$

A summation formula (see Appendix C.1): $\sum_{i=0}^{m} i = \frac{m(m+1)}{2}$

So the total number of steps taken is

$$\sum_{i=0}^{n^2+4} i = \frac{(n^2+4)(n^2+5)}{2}, \text{which is } \Theta(n^4).$$

Running time of comprehensions

```python
def square_all(numbers: list[int]) -> list[int]:
    """Return a list containing the squares of
    the given numbers.
    """
    return [x ** 2 for x in numbers]
```

Same as for loops:

- the "number of iterations" is the length of the collection, and
- the "steps per iteration" is the steps for the expression `x ** 2`.

So using similar analysis to before, $RT_{\text{square\_all}} \in \Theta(n)$, where n is the length of `numbers`.

Running time of while loops

```python
def sum_powers_of_two(n: int) -> int:
    """Precondition: n > 1
    """
    sum_so_far = 0
    i = 1

    while i < n:
        sum_so_far = sum_so_far + i
        i = i * 2

    return sum_so_far
```

Analyzing a while loop

1. Identify a pattern for how the loop variable `i` changes.

```
i = 1

while i < n:
    ...
    i = i * 2
```

Iteration	`i`
0	1
1	2
2	4
3	8
4	16
...	...

In general, the value of `i` after k iterations is $i_k = 2^k$.

Analyzing a while loop

2. Find the smallest value of k such that i_k makes the loop condition False.

```
i = 1

while i < n:
    ...
    i = i * 2
```

Know: $i_k = 2^k$

Want: $i_k \geq n$

$$2^k \geq n$$

$$k \geq \log_2 n$$

Smallest value: $k = \lceil \log_2 n \rceil$

```python
def sum_powers_of_two(n: int) -> int:
    sum_so_far = 0
    i = 1

    while i < n:
        sum_so_far = sum_so_far + i
        i = i * 2

    return sum_so_far
```

Analysis.

- First two statements count as 1 step.
- While loop takes $\lceil \log_2 n \rceil$ iterations, with 1 step per iteration.
- Last statement counts as 1 step.

Total number of steps is $1 + \lceil \log_2 n \rceil + 1 = \lceil \log_2 n \rceil + 2$, which is $\Theta(\log_2 n)$.

Exercise 1: Analysing running time of while loops

Exercise 2: Analysing nested loops

A Twisted Example:

we can’t always determine the exact running time!

```python
def twisty(n: int) -> int:
    """Return the number of iterations it takes for this
    special loop to stop for the given n.
    """
    iterations_so_far = 0
    x = n
    while x > 1:
        if x % 2 == 0:              # x is even
            x = x // 2
        else:                       # x is odd
            x = 2 * x - 2
        iterations_so_far = iterations_so_far + 1

    return iterations_so_far
```

Analysis.

- First two lines are constant time (count as 1 step).
- Final return statement is 1 step.
- The loop body is constant time (1 step).
- The number of iterations is...?

Focus on the loop:

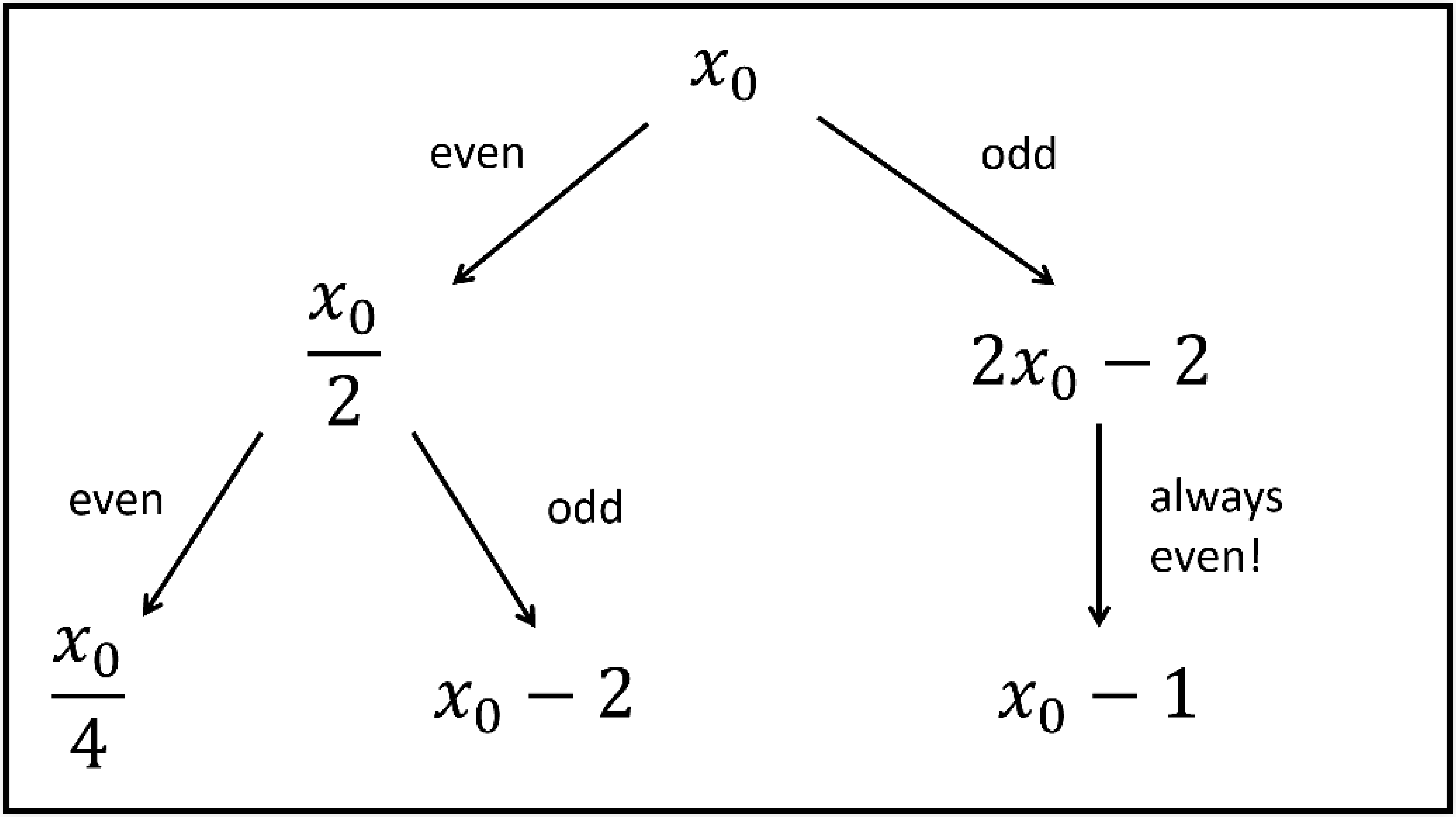
```
x = n
while x > 1:
    if x % 2 == 0:    # even
        x = x // 2
    else:             # odd
        x = 2 * x - 2
```

`twisty(11)`

Iteration	`x`
0	11
1	20
2	10
3	5
4	8
5	4
6	2
7	1

Problem: can't find a formula for x_k (the value of x after k iterations)

Idea: look at what happens after two iterations instead of one


x_0
even
odd
$\frac{x_0}{2}$
$2x_0 - 2$
even
odd
always even!
$\frac{x_0}{4}$
$x_0 - 2$
$x_0 - 1$

Claim. For any integer value of `x` greater than 2, after two iterations of the while loop in `twisty`, the value of `x` has decreased by at least one.

Or, as an inequality:

$$x_2 \leq x_0 - 1$$

where x_0 is the starting value of `x`, and x_2 is the value of `x` after two iterations.

Proof. Use a proof by cases. Can do cases based on even/odd (with subcases), or use four separate cases (based on $x_0 \ \% \ 4$).

See course notes for details.

How does this help?

- After 0 iterations, `x == n`
- After 2 iterations, `x <= n - 1`
- After 4 iterations, `x <= n - 2`
- After 6 iterations, `x <= n - 3`

$\boxed{\text{Let } x_{2k} \text{ be the value of } \texttt{x} \text{ after } 2k \text{ iterations. Then } x_{2k} \leq n - k.}$

The while loop stops when `x > 1` is `False`. That is, when $x_{2k} \leq 1$.

When $k = n - 1$, x_{2k} must be ≤ 1. (Since $n - (n - 1) = 1$.)

(And x_{2k} could be ≤ 1 much earlier.)

So the while loop must stop after at most $2(n - 1)$ iterations.

Back to our analysis

Analysis. First two lines are constant time (1 step), final return statement is 1 step.

The loop body is constant time (1 step). The number of iterations is at most $2(n-1)$.

So the total running time is at most $1 + 2(n-1) + 1 = 2n$ steps.

Or, $RT_{\mathrm{twisty}}(n) \leq 2n$.

This leads to an **upper bound** of $\mathcal{O}(n)$. (Not $\Theta(n)$!)

So… all that work, and we just got an upper bound of $RT_{\text{twisty}}(n) \in \mathcal{O}(n)$.

But we want Theta!

In fact, you can show:

$$RT_{\text{twisty}}(n) \in \mathcal{O}(\log n) \wedge RT_{\text{twisty}}(n) \in \Omega(\log n)$$

So:

$$RT_{\text{twisty}}(n) \in \Theta(\log n)$$

Deducing this requires a more careful analysis!

See the course notes for the exciting end to this twisty story.

Summary

Today you learned to...

1. Analyse the running time of code containing nested loops.
2. Analyse the running time of code containing comprehensions and while loops.
3. Perform running-time analyses by finding bounds on the running-time function (rather than an exact expression).

Homework

- Readings:
  - From today: 9.6
  - Tomorrow: : 9.7
- Work on Assignment 4