DSC 40B

Theoretical Foundations II

Lecture 3 | Part 1

Properties

Properties

- If one line of code takes $\Theta(n)$ time.
- And the next line of code takes $O(n)$ time.
- Do they take $\Theta(n)$ in total?

Properties of Θ

1. **Symmetry**:

$$\text{If } f(n) = \Theta(g(n)), \text{ then } g(n) = \Theta(f(n)).$$

2. **Transitivity**:

$$\text{If } f(n) = \Theta(g(n)), \text{ and } g(n) = \Theta(h(n)) \text{ then } f(n) = \Theta(h(n)).$$

3. **Reflexivity**:

$$f(n) = \Theta(f(n))$$

Proving Properties

- We want to prove **symmetry**: if $f(n) = \Theta(g(n))$, then $g(n) = \Theta(f(n))$.

- We need to find positive constants N, c_1, c_2 so that for all $n \geq N$:

$$c_1 f(n) \leq g(n) \leq c_2 f(n)$$

Step 1: State the assumption

- We know that $f(n) = \Theta(g(n))$.

- So there are constants M, b_1, b_2 so that for all $n \geq M$:

$$b_1 g(n) \leq f(n) \leq b_2 g(n)$$

Step 2: Use the assumption

$$b_1 g(n) \leq f(n) \leq b_2 g(n) \qquad (n \geq M)$$

- Dividing by b_1: $g(n) \leq \frac{1}{b_1} f(n)$
- Dividing by b_2: $g(n) \geq \frac{1}{b_2} f(n)$
- So: $\frac{1}{b_2} f(n) \leq g(n) \leq \frac{1}{b_1} f(n)$ for $n \geq M$.

Exercise

Show that if $f_1 = \Theta(g)$ and $f_2 = O(g)$, then $f_1 + f_2 = \Theta(g)$.

Theta, Big-O, and Big-Omega

- If $f(n) = \Theta(g(n))$...
- ...then $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Sums of Theta

- If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then

$$\begin{aligned} f_1(n) + f_2(n) &= \Theta(g_1(n) + g_2(n)) \\ &= \Theta(\max(g_1(n), g_2(n))) \end{aligned}$$

- Useful for sequential code.

Example

```
def foo(n):
    bar(n)
    baz(n)
```

- $T_{\text{foo}}(n) = T_{\text{bar}}(n) + T_{\text{baz}}(n)$
- If $T_{\text{bar}} = \Theta(n^2)$ and $T_{\text{baz}}(n) = \Theta(n^3)$...
- ...then $T_{\text{foo}}(n) = \Theta(n^3)$.
- `baz` is the **bottleneck**.

Products of Theta

- If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then

$$f_1(n) \cdot f_2(n) = \Theta(g_1(n) \cdot g_2(n))$$

Example

```
def foo(n):
    for i in range(3*n + 4, 5n**2 - 2*n + 5):
        for j in range(500*n, n**3):
            print(i, j)
```

Careful!

- If inner loop index depends on outer loop, you have to be more careful.

```
def foo(n):
    for i in range(n):
        for j in range(i):
            print(i, j)
```

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 2

Practicalities

In this part...

- Other ways asymptotic notation is used.
- Asymptotic notation *faux pas*.
- Downsides of asymptotic notation.

Not Just for Time Complexity!

- We most often see asymptotic notation used to express time complexity.
- But it can be used to express any type of growth!

Example: Combinatorics

- Recall: $\binom{n}{k}$ is number of ways of choosing k things from a set of n.

- How fast does this grow with n? For fixed k:

$$\binom{n}{k} = \Theta(n^k)$$

Example: Central Limit Theorem

- Recall: the CLT says that the sample mean has a normal distribution with standard deviation $\sigma_{\text{pop}}/\sqrt{n}$

- The **error** in the sample mean is: $O(1/\sqrt{n})$

Faux Pas

- Asymptotic notation can be used improperly.
  - Might be technically correct, but defeats the purpose.

- Don't do these in, e.g., interviews!

Faux Pas #1

- Don't include constants, lower-order terms in the notation.
- **Bad:** $3n^2 + 2n + 5 = \Theta(3n^2)$.
- **Good:** $3n^2 + 2n + 5 = \Theta(n^2)$.
- It isn't *wrong* to do so, just defeats the purpose.

Faux Pas #2

- Don't include base in logarithm.
- **Bad:** $\Theta(\log_2 n)$
- **Good:** $\Theta(\log n)$
- Why? $\log_2 n = c \cdot \log_3 n = c' \log_4 n = \ldots$

Faux Pas #3

- Don't misinterpret meaning of $\Theta(\cdot)$.

- $f(n) = \Theta(n^3)$ does **not** mean that there are constants so that $f(n) = c_3 n^3 + c_2 n^2 + c_1 n + c_0$.

Faux Pas #4

- Time complexity is not a **complete** measure of efficiency.
- $\Theta(n)$ is not always better than $\Theta(n^2)$.
- Why?

Faux Pas #4

- **Why?** Asymptotic notation "hides the constants".
- $T_1(n) = 1{,}000{,}000n = \Theta(n)$
- $T_2(n) = 0.00001n^2 = \Theta(n^2)$
- But $T_1(n)$ is **worse** for all but really large n.

Main Idea

Time complexity is not the **only** way to measure efficiency, and it can be misleading.

Sometimes even a $\Theta(2^n)$ algorithm is better than a $\Theta(n)$ algorithm, if the data size is small.

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 3

The Movie Problem

The Movie Problem

The Movie Problem

- **Given**: an array `movies` of movie durations, and the flight duration `t`
- **Find**: two movies whose durations add to `t`.
  - If no two movies sum to `t`, return `None`.

Exercise

Design a brute force solution to the problem. What is its time complexity?

```python
def find_movies(movies, t):
    n = len(movies)
    for i in range(n):
        for j in range(i + 1, n):
            if movies[i] + movies[j] == t:
                return (i, j)
    return None
```

Time Complexity

- It looks like there is a **best** case and **worst** case.
- How do we formalize this?

Exercise

Can you come up with a better algorithm?

Best Possible

- What is the *best possible time complexity* for an algorithm solving this problem?
- **Lower Bound Theory**

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 4

Best and Worst Cases

Recall: `mean`

```python
def mean(arr):
    total = 0
    for x in arr:
        total += x
    return total / len(arr)
```

Time Complexity of `mean`

- Linear time, $\Theta(n)$.
- Depends **only** on the array's **size**, n, not on its actual elements.

Linear Search

- **Given**: an array `arr` of numbers and a target `t`.
- **Find**: the index of `t` in `arr`, or `None` if it is missing.

```python
def linear_search(arr, t):
    for i, x in enumerate(arr):
        if x == t:
            return i
    return None
```

Exercise

What is the time complexity of `linear_search`?

The Best Case

- When `t` is the very first element.
- The loop exits after one iteration.
- $\Theta(1)$ time?

The Worst Case

- When `t` is not in the array at all.
- The loop exits after n iterations.
- $\Theta(n)$ time?

Time Complexity

- `linear_search` can take vastly different amounts of time on two inputs of the **same size**.
  - Depends on **actual elements** as well as size.
- There is no single, overall time complexity here.
- Instead we'll report **best** and **worst** case time complexities.

Best Case Time Complexity

- How does the time taken in the **best case** grow as the input gets larger?

Definition

Define $T_{\text{best}}(n)$ to be the **least** time taken by the algorithm on any input of size n.

The asymptotic growth of $T_{\text{best}}(n)$ is the algorithm's **best case time complexity**.

Best Case

- In `linear_search`'s **best case**, $T_{\text{best}}(n) = c$, no matter how large the array is.

- The **best case time complexity** is $\Theta(1)$.

Worst Case Time Complexity

- How does the time taken in the **worst case** grow as the input gets larger?

Definition

Define $T_{\text{worst}}(n)$ to be the **most** time taken by the algorithm on any input of size n.

The asymptotic growth of $T_{\text{worst}}(n)$ is the algorithm's **worst case time complexity**.

Worst Case

- In the worst case, `linear_search` iterates through the entire array.
- The **worst case time complexity** is $\Theta(n)$.

The Movie Problem

```python
def find_movies(movies, t):
    n = len(movies)
    for i in range(n):
        for j in range(i + 1, n):
            if movies[i] + movies[j] == t:
                return (i, j)
    return None
```

Exercise

What are the best case and worst case time complexities?

Best Case

- Best case occurs when movie 1 and movie 2 add to the target.
- Takes constant time, independent of number of movies.
- Best case time complexity: $\Theta(1)$.

Worst Case

- Worst case occurs when no two movies add to target.
- Has to loop over all $\Theta(n^2)$ pairs.
- Worst case time complexity: $\Theta(n^2)$.

Caution!

- The best case is never: “the input is of size one”.
- The best case is about the **structure** of the input, not its **size**.

Note

- An algorithm like `linear_search` doesn't have **one single** time complexity.
- An algorithm like `mean` does, since the best and worst case time complexities coincide.

Main Idea

Reporting **best** and **worst** case time complexities gives us a richer of the performance of the algorithm.

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 5

About Notation

A Common Mistake

- You'll sometimes see people equate $O(\cdot)$ with **worst case** and $\Omega(\cdot)$ with **best case**.
- This isn't right!

Why?

- $O(\cdot)$ expresses ignorance about a lower bound.
  - $O(\cdot)$ is like $\leq$
- $\Omega(\cdot)$ expresses ignorance about an upper bound.
  - $\Omega(\cdot)$ is like $\geq$
- Having both bounds is actually important here.

Example

- Suppose we said: "the worst case time complexity of `find_movies` is $O(n^2)$."
- Technically true, but not precise.
- This is like saying: "I **don't know** how bad it actually is, but it can't be worse than quadratic."
  - It could still be linear!"
- **Better**: the worst case time complexity is $\Theta(n^2)$.

Example

- Suppose we said: "the best case time complexity of `find_movies` is $\Omega(1)$."
- This is like saying: "I **don't know** how good it actually is, but it can't be better than constant."
  - It could be linear!
- **Correct**: the best case time complexity is $\Theta(1)$.

Put Another Way...

- It isn't **technically wrong** to say worst case for `find_movies` is $O(n^2)$...
- ...but it isn't **technically wrong** to say it is $O(n^{100})$, either!