CS-GY 6033

Design and Analysis of Algorithms I

Lecture 2 | Part 1

News

News

- Homework 01 posted on
  - Due Feb 5 @ 11:59 pm PST on Gradescope.
  - LaTeX template available.
  - 10% extra credit if use LaTeX.
- Quiz: next week.

Agenda

1. What is Θ notation, really?
2. Worst case vs average case.
3. Lower bounds

CS-GY 6033

Design and Analysis of Algorithms I

Lecture 2 | Part 2

Big Theta, Formalized

So Far

- Time Complexity Analysis: a picture of how an algorithm **scales**.
- Can use Θ-notation to express time complexity.
- Allows us to **ignore** details in a rigorous way.
  - **Saves us work!**
  - **But what exactly can we ignore?**

Now

- A deeper look at **asymptotic notation**:
- What does $\Theta(\cdot)$ mean, exactly?
- Related notations: $O(\cdot)$ and $\Omega(\cdot)$.
- How these notations save us work.

Theta Notation, Informally

- $\Theta(\cdot)$ forgets constant factors, lower-order terms.

$$5n^3 + 3n^2 + 42 = \Theta(n^3)$$

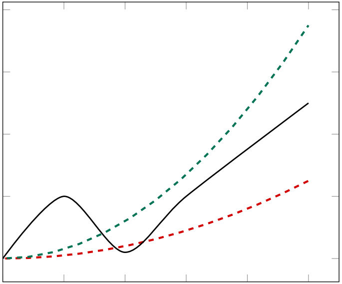
- $f(n) = \Theta(g(n))$ if $f(n)$ "grows like" $g(n)$.

- More examples:
  - $4n^2 + 3n - 20 = \Theta(n^2)$
  - $3n + \sin(4\pi n) = \Theta(n)$
  - $2^n + 100n = \Theta(2^n)$

Definition

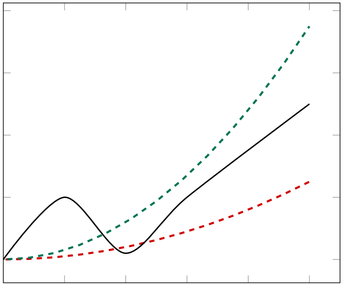
We write $f(n) = \Theta(g(n))$ if there are positive constants N, c_1 and c_2 such that for all $n \geq N$:

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



Main Idea

If $f(n) = \Theta(g(n))$, then when n is large f is "sandwiched" between copies of g.



Proving Big-Theta

- We can prove that $f(n) = \Theta(g(n))$ by finding these constants.

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad (n \geq N)$$

- Requires an upper bound and a lower bound.

Strategy: Chains of Inequalities

- To show $f(n) \leq c_2 g(n)$, we show:

$$f(n) \leq (\text{something}) \leq (\text{another thing}) \leq \ldots \leq c_2 g(n)$$

- At each step:
  - We can do anything to make value **larger**.
  - But the goal is to simplify it to look like $g(n)$.

Example

- Show that $4n^3 - 5n^2 + 50 = \Theta(n^3)$.

- Find constants c_1, c_2, N such that for all $n > N$:

$$c_1 n^3 \quad \leq \quad 4n^3 - 5n^2 + 50 \quad \leq \quad c_2 n^3$$

- They don’t have to be the “best” constants! Many solutions!

Example

$$c_1 n^3 \quad \leq \quad 4n^3 - 5n^2 + 50 \quad \leq \quad c_2 n^3$$

- We want to make $4n^2 - 5n^2 + 50$ "look like" cn^3.
- For the upper bound, can do anything that makes the function **larger**.
- For the lower bound, can do anything that makes the function **smaller**.

Example

$$c_1 n^3 \quad \leq \quad 4n^3 - 5n^2 + 50 \quad \leq \quad c_2 n^3$$

- Upper bound:

Example

$$c_1 n^3 \quad \leq \quad 4n^3 - 5n^2 + 50 \quad \leq \quad c_2 n^3$$

- Lower bound:

Example

$$c_1 n^3 \quad \leq \quad 4n^3 - 5n^2 + 50 \quad \leq \quad c_2 n^3$$

- All together:

Upper-Bounding Tips

- "Promote" lower-order **positive** terms:

$$3n^3 + 5n \le 3n^3 + 5n^3$$

- "Drop" **negative** terms

$$3n^3 - 5n \le 3n^3$$

Lower-Bounding Tips

- "Drop" lower-order **positive** terms:

$$3n^3 + 5n \geq 3n^3$$

- "Promote and cancel" negative lower-order terms if possible:

$$4n^3 - 2n \geq 4n^3 - 2n^3 = 2n^3$$

Lower-Bounding Tips

- "Cancel" negative lower-order terms with big constants by "breaking off" a piece of high term.

$$
\begin{aligned}
4n^3 - 10n^2 &= (3n^3 + n^3) - 10n^2 \\
&= 3n^3 + (n^3 - 10n^2)
\end{aligned}
$$

$n^3 - 10n^2 \geq 0$ when $n^3 \geq 10n^2 \implies n \geq 10$:

$$\geq 3n^3 + 0 \qquad (n \geq 10)$$

Caution

- To upper bound a fraction A/B, you must:
  - Upper bound the numerator, A.
  - *Lower* bound the denominator, B.

- And to lower bound a fraction A/B, you must:
  - Lower bound the numerator, A.
  - *Upper* bound the denominator, B.

Exercise

Let $f(n) = [3n + (n\sin(\pi n) + 3)]n$. Which one of the following is true?

- $f = \Theta(n)$
- $f = \Theta(n^2)$
- $f = \Theta(n\sin(\pi n))$

CS-GY 6033
Design and Analysis of Algorithms I

Lecture 2 | Part 3

Big-Oh and Big-Omega

Other Bounds

- $f = \Theta(g)$ means that f is both **upper** and **lower** bounded by factors of g.
- Sometimes we only have (or care about) upper bound or lower bound.
- We have notation for that, too.

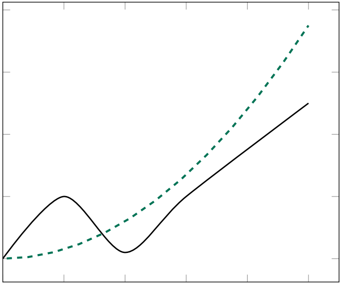
Big-O Notation, Informally

- Sometimes we only care about **upper bound**.
- $f(n) = O(g(n))$ if f "grows at most as fast" as g.
- Examples:
  - $4n^2 = O(n^{100})$
  - $4n^2 = O(n^3)$
  - $4n^2 = O(n^2)$ and $4n^2 = \Theta(n^2)$

Definition

We write $f(n) = O(g(n))$ if there are positive constants N and c such that for all $n \geq N$:

$$f(n) \leq c \cdot g(n)$$



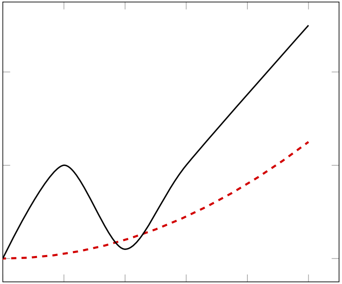
Big-Omega Notation

- Sometimes we only care about **lower bound**.
- Intuitively: $f(n) = \Omega(g(n))$ if f "grows at least as fast" as g.
- Examples:
  - $4n^{100} = \Omega(n^5)$
  - $4n^2 = \Omega(n)$
  - $4n^2 = \Omega(n^2)$ and $4n^2 = \Theta(n^2)$

Definition

We write $f(n) = \Omega(g(n))$ if there are positive constants N and c such that for all $n \geq N$:

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



Theta, Big-O, and Big-Omega

- If $f = \Theta(g)$ then $f = O(g)$ and $f = \Omega(g)$.
- If $f = O(g)$ and $f = \Omega(g)$ then $f = \Theta(g)$.
- Pictorially:
  - $\Theta \implies (O \text{ and } \Omega)$
  - $(O \text{ and } \Omega) \implies \Theta$

Analogies

- Θ is kind of like $=$
- O is kind of like $\leq$
- Ω is kind of like $\geq$

Big-Oh

- Often used when another part of the code would dominate time complexity anyways.

Exercise

What is the time complexity of `foo`?

```python
def foo(n):
    for a in range(n**4):
        print(a)

    for i in range(n):
        for j in range(i**2):
            print(i + j)
```

Big-Omega

- Often used when the time complexity will be so large that we don't care what it is, exactly.

Example: Big-Omega

```python
best_separation = float('inf')
best_clustering = None

for clustering in all_clusterings(data):
    sep = calculate_separation(clustering)
    if sep < best_separation:
        best_separation = sep
        best_clustering = clustering

print(best_clustering)
```

Other Notations

- $f(n) = o(g(n))$ if f grows “much slower” than g.
  - Whatever c you choose, eventually $f < cg(n)$.
  - Example: $n^2 = o(n^3)$

- $f(n) = \omega(g(n))$ if f grows “much faster” than g
  - Whatever c you choose, eventually $f > cg(n)$.
  - Example: $n^3 = \omega(n^2)$

- We won’t really use these.

CS-GY 6033

Design and Analysis of Algorithms I

Lecture 2 | Part 4

Properties

Properties

- We don't usually go back to the definition when using Θ.
- Instead, we use a few basic **properties**.

Properties of Θ

1. **Symmetry**: If $f = \Theta(g)$, then $g = \Theta(f)$.
2. **Transitivity**: If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.
3. **Reflexivity**: $f = \Theta(f)$

Exercise

Which of the following properties are true?

- T or F: If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
- T or F: If $f = \Omega(h)$ and $g = \Omega(h)$, then $f = \Omega(g)$.
- T or F: If $f_1 = \Theta(g_1)$ and $f_2 = O(g_2)$, then $f_1 + f_2 = \Theta(g_1 + g_2)$.
- T or F: If $f_1 = \Theta(g_1)$ and $f_2 = \Theta(g_2)$, then $f_1 \times f_2 = \Theta(g_1 \times g_2)$.

Proving/Disproving Properties

- Start by trying to disprove.
- Easiest way: find a counterexample.
- Example: If $f = \Omega(h)$ and $g = \Omega(h)$, then $f = \Omega(g)$.
  - **False!** Let $f = n^3$, $g = n^5$, and $h = n^2$.

Proving the Property

- If you can't disprove, maybe it is true.

- Example:
  - Suppose $f_1 = O(g_1)$ and $f_2 = O(g_2)$.
  - Prove that $f_1 \times f_2 = O(g_1 \times g_2)$.

Step 1: State the assumption

- We know that $f_1 = O(g_1)$ and $f_2 = O(g_2)$.
- So there are constants c_1, c_2, N_1, N_2 so that for all $n \geq N$:

$$f_1(n) \leq c_1 g_1(n) \qquad (n \geq N_1)$$
$$f_2(n) \leq c_2 g_2(n) \qquad (n \geq N_2)$$

Step 2: Use the assumption

- Chain of inequalities, starting with $f_1 \times f_2$, ending with $\leq cg_1 \times g_2$.
- Using the following piece of information:

$$f_1(n) \leq c_1 g_1(n) \qquad (n \geq N_1)$$
$$f_2(n) \leq c_2 g_2(n) \qquad (n \geq N_2)$$

Analyzing Code

- The properties of Θ (and O and Ω) are useful when analyzing code.
- We can analyze pieces, put together the results.

Sums of Theta

- **Property**: If $f_1 = \Theta(g_1)$ and $f_2 = \Theta(g_2)$, then $f_1 + f_2 = \Theta(g_1 + g_2)$
- Used when analyzing **sequential** code.

Example

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

- Say `bar` takes $\Theta(n^3)$, `baz` takes $\Theta(n^4)$.
- `foo` takes $\Theta(n^4 + n^3) = \Theta(n^4)$.
- `baz` is the **bottleneck**.

Products of Theta

- **Property**: If $f_1 = \Theta(g_1)$ and $f_2 = \Theta(g_2)$, then

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

- Useful when analyzing nested **loops**.

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)
```

CS-GY 6033

Design and Analysis of Algorithms I

Lecture 2 | Part 5

Asymptotic Notation Practicalities

In this part...

- Asymptotic notation *faux pas*.
- Downsides of asymptotic notation.

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.

CS-GY 6033

Design and Analysis of Algorithms I

Lecture 2 | Part 6

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?

CS-GY 6033

Design and Analysis of Algorithms I

Lecture 2 | Part 7

Best and Worst Cases

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**.

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**.

Exercise

What are the best case and worst case time complexities of `find_movies`?

```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
```

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**.
- Not always constant time! Example: sorting.

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 understanding of the performance of the algorithm.

CS-GY 6033

Design and Analysis of Algorithms I

Lecture 2 | Part 8

Average Case

Time Taken, Typically

- Best case and worst case can be **misleading**.
  - Depend on a **single good/bad input**.

- How much time is taken, typically?

- **Idea:** compute the average time taken over all possible inputs.

Recall: The Expectation

- The expected value of a random variable X is:

$$\sum_x x \cdot P(X = x)$$

winnings	probability
\$ 0	50%
\$ 1	30%
\$ 10	18%
\$ 50	2%

Expected winnings:

Average Case

- We'll compute the expected time over all cases:

$$T_{\text{avg}}(n) = \sum_{\text{case} \in \text{all cases}} P(\text{case}) \cdot T(\text{case})$$

- Called the **average case time complexity**.

Strategy for Finding Average Case

- **Step 0:** Make assumption about distribution of inputs.
- **Step 1:** Determine the possible cases.
- **Step 2:** Determine the probability of each case.
- **Step 3:** Determine the time taken for each case.
- **Step 4:** Compute the expected time (average).

Example: Linear Search

- Recall **linear search**:

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

- Best case? Worst case?
- What is the **average case time complexity** of **linear search**?

Step 0: Assume input distribution

- We must assume something about the input.
- Example: Target must be in array, equally-likely to be any element, no duplicates.
- This is usually given to you.

Step 1: Determine the Cases

- Example: linear search.

Case 1:	target is first element
Case 2:	target is second element
	$\vdots$
Case n:	target is nth element
Case $n+1$:	target is not in array

Step 2: Case Probabilities

- What is the probability that we see each case?
  - Example: what is the probability that the target is the kth element?

- This is where we use assumptions from Step 0.

Example

- **Assume**: target is in the array exactly once, equally-likely to be any element.

- Each case has probability $1/n$.

Step 3: Case Times

- Determine time taken in each case.

- Example: linear search.
  - Let's say it takes time c per iteration.

$$
\begin{aligned}
\text{Case 1:} &\quad \text{time } c \\
\text{Case 2:} &\quad \text{time } 2c \\
&\quad \vdots \\
\text{Case i:} &\quad \text{time } c \cdot i \\
&\quad \vdots \\
\text{Case } n\text{:} &\quad \text{time } c \cdot n
\end{aligned}
$$

Step 4: Compute Expectation

$$T_{\text{avg}}(n) = \sum_{i=1}^{n} P(\text{case } i) \cdot T(\text{case } i)$$

Average Case Time Complexity

- The **average case** time complexity[1] of **linear search** is $\Theta(n)$.

[1] Under these assumptions on the input!

Note

- **Hard** to make realistic assumptions on input distribution.
- Example: linear search.
  - Is it realistic to assume t is in array?
  - If not, what is the probability that it *is* in the array?

Exercise

Suppose we change our assumptions:

- The target has a 50% chance of being in the array.
- If it is in the array, it is equally-likely to be any element.

What is the average case complexity now?

CS-GY 6033

Design and Analysis of Algorithms I

Lecture 2 | Part 9

Average Case in Movie Problem

The Movie Problem

Recall: 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`.

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
```

“Average” Case?

- Best case: $\Theta(1)$
  - When the first pair of movies checked equals target.
- Worst case: $\Theta(n^2)$
  - When no pair of movies equals target.
- The best and worst cases are **extremes**.
- How much time is taken, *typically*?
  - That is, when the target pair is not the first checked nor the last, but somewhere in the middle.

Exercise

How much time do you expect `find_movies` to take on a typical input?

- $\Theta(1)$
- $\Theta(n^2)$
- Something in between, like $\Theta(n)$

Step 0: Assume input distribution

- Suppose we are told that:
  - There is a unique pair of movies that add to t.
  - All pairs are equally likely.

Step 1: Determine the Cases

- Case α: the αth pair checked sums to t.
- Each pair of movies is a case.
- There are $\binom{n}{2}$ cases.

Step 2: Case Probabilities

- **Assume**: there is a *unique* pair that adds to `t`.
- **Assume**: all pairs are equally likely.
- Probability of any case: $\frac{1}{\binom{n}{2}} = \frac{2}{n(n-1)}$

Step 3: Case Time

- How much time is taken for a particular case?
- Example, suppose the movies a and b sum to the target.
- How long does it take to find this pair?

```
1 def find_movies(movies, t):
2     n = len(movies)
3     for i in range(n):
4         for j in range(i + 1, n):
5             if movies[i] + movies[j] == t:
6                 return (i, j)
7     return None
```

Exercise

Roughly much time is taken (how many times does line 5 run) if the αth pair checked sums to the target?

Step 4: Compute Expectation

Average Case

- The average case time complexity of `find_movies` is $\Theta(n^2)$.
- Same as the worst case!

Note

- We’ve seen two algorithms where the average case = the worst case.
- Not always the case!
- Interpretation: the worst case is not too extreme.

CS-GY 6033

Design and Analysis of Algorithms I

Lecture 2 | Part 10

Expected Time Complexity

Example: Contrived Algorithm

```python
def wibble(n):
    # generate random number between 0 and n
    x = np.random.randint(0, n)

    if x == 0:
        for i in range(n):
            print('Unlucky!')
    else:
        print('Lucky!')
```

Exercise

How much time does `wibble` take *on average*?

Random Algorithms

- This algorithm is *randomized*.
- The time it takes is also *random*.
- What is the **expected time**?

Average Case vs. Expected Time

- With average case complexity, a probability distribution on inputs is specified.
- Now, the randomness is *in the algorithm itself*.
- Otherwise, the analysis is very similar.

Step 1: Determine the cases

```
def wibble(n):
    x = np.random.randint(0, n)

    if x == 0:
        for i in range(n):
            print('Unlucky!')
    else:
        print('Lucky!')
```

- Case 1: `x == 0`
- Case 2: `x != 0`

Step 2: Determine case probabilities

```
def wibble(n):
    x = np.random.randint(0, n)

    if x == 0:
        for i in range(n):
            print('Unlucky!')
    else:
        print('Lucky!')
```

- P(Case 1) = $1 / n$
- P(Case 2) = $(n - 1) / n$

Step 3: Determine case times

```
def wibble(n):
    x = np.random.randint(0, n)

    if x == 0:
        for i in range(n):
            print('Unlucky!')
    else:
        print('Lucky!')
```

- Case 1: $\Theta(n)$
- Case 2: $\Theta(1)$

Step 4: Compute expectation

- Compute expected time:

Expected Time

- This was a contrived example.

- Some important algorithms involve randomness!
  - Quicksort
  - We'll see alg. for median with $\Theta(n)$ expected time.

CS-GY 6033

Design and Analysis of Algorithms I

Lecture 2 | Part 11

Lower Bound Theory

Imagine...

- You write a simple algorithm to solve a problem.
- You analyze time complexity and find it is $\Theta(n^2)$.
- You ask yourself: *can I do better than* $\Theta(n^2)$*?*
- Or: *What is the best time complexity possible?*

Doing Better

- How can you know what you don’t know?
- You can argue that *any* algorithm for solving the problem *must* take at least a certain amount of time in the worst case.

Example: Minimum

- Problem: Find minimum in array of length n.
- *Any* algorithm has to check all n numbers in the worst case.
  - Or else the number not checked could have been the smallest!
- Takes at least linear ($\Omega(n)$) time.
  - **No algorithm** for the min can have worst case of < linear time.

Definition

A **theoretical lower bound** is a lower bound on the worst-case time complexity of **any algorithm** solving a particular problem.

Main Idea

No algorithm's worst case can be better than theoretical lower bound.

Loose Lower Bounds

- $\Omega(\log n)$, $\Theta(\sqrt{n})$ and $\Theta(1)$ are also theoretical lower bounds for finding the minimum.
- But no algorithm can exist which has a worst case of $\Theta(\log n)$, $\Theta(\sqrt{n})$, or $\Theta(1)$.
- This bound is **loose**. Not super useful.

Tight Lower Bounds

- A lower bound is **tight** if there exists an algorithm with that worst case time complexity.
- That algorithm is (in a sense) **optimal**.

How to find a TLB

- Argument from completeness:
  - The algorithm might not be correct if it doesn't check k things, so the time is $\Omega(k)$.
- Argument from I/O:
  - If the output is an array of size k, time taken is $\Omega(k)$
- More sophisticated arguments...

Tight Bounds can be difficult to find

- Often require sophisticated combinatorial arguments outside of the scope of CS-GY 6033.

Assumptions make problems easier

- The TLB for finding a minimum changes if we assume that the array is sorted.

Exercise

Consider these two problems:

1. Find the min of a sorted array.
2. Given a target t and a sorted array, determine whether t is in the array.

Find tight theoretical lower bounds for each problem.

Main Idea

When coming up with an algorithm, first try to find a tight TLB. Then try to make an algorithm which has that worst-case complexity. If you can, it's **optimal**!

CS-GY 6033
Design and Analysis of Algorithms I

Lecture 2 | Part 12

Case Study: Matrix Multiplication

It’s Important

- Matrix multiplication is a *very* common operation in machine learning algorithms.
- **Estimate**: 75% - 95% of time training a neural network is spent in matrix multiplication.

Recall

- If A is $m \times p$ and B is $p \times n$, then AB is $m \times n$.
- The ij entry of AB is

$$(AB)_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$

Naïve Algorithm

- This algorithm is relatively straightforward to code up.

```python
def mmul(A, B):
    """
    A is (m x p) and B is (p x n)
    """
    m, p = A.shape
    n = B.shape[1]

    C = np.zeros((m, n))

    for i in range(m):
        for j in range(n):
            for k in range(p):
                C[i,j] += A[i,k] * B[k, j]

    return C
```

Time Complexity

- The naïve algorithm takes time $\Theta(mnp)$.
- If both matrices are $n \times n$, then $\Theta(n^3)$ time.
- **Cubic!**

Cubic Time Complexity

- The largest problem size that can be solved, if a basic operation takes 1 nanosecond.

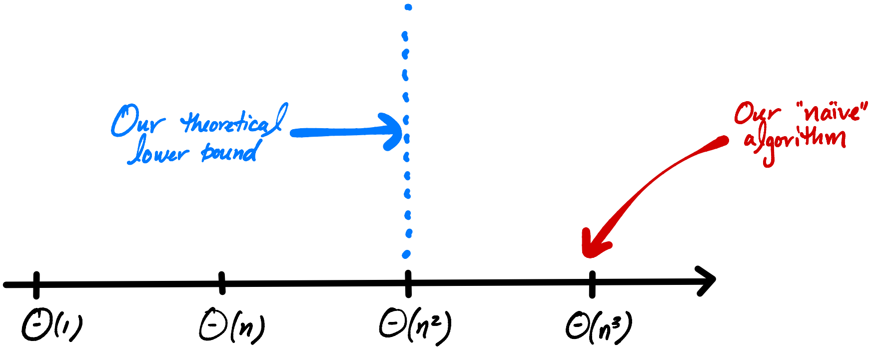
1 s	10 m	1 hr
1,000	6,694	15,326

The Question

- Can we do better?
- How fast can we possibly multiply matrices?

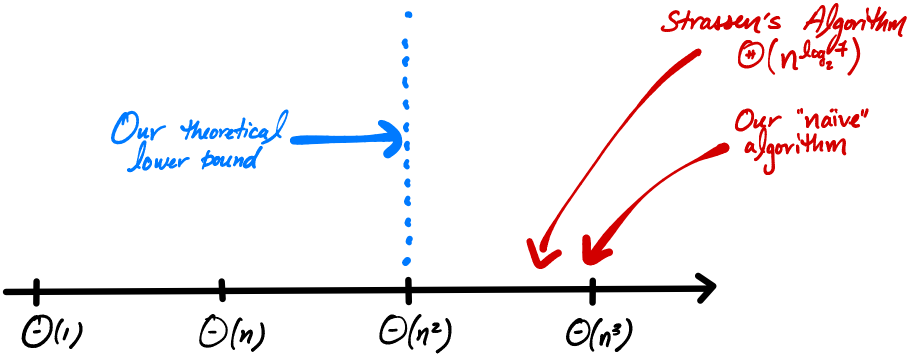
Theoretical Lower Bound

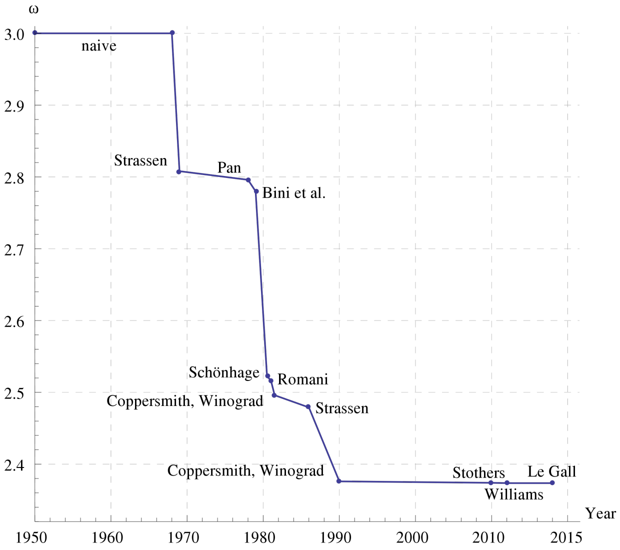
- If A and B are $n \times n$, C will have n^2 entries.
- Each entry must be filled: $\Omega(n^2)$ time.
- That is, matrix multiplication must take at least quadratic time.
- Is this bound **tight**? Can it be increased?


Our theoretical lower bound
Our "naïve" algorithm
Θ(1)
Θ(n)
Θ(n²)
Θ(n³)

Strassen's Algorithm

- Cubic was as good as it got...
- ...until Strassen, 1969.
- Time complexity: $\Theta(n^{\log_2 7}) = \Theta(n^{2.8073})$


Strassen's Algorithm
$\Theta(n^{\log_2 7})$
Our "naïve" algorithm
Our theoretical lower bound
$\Theta(1)$
$\Theta(n)$
$\Theta(n^2)$
$\Theta(n^3)$

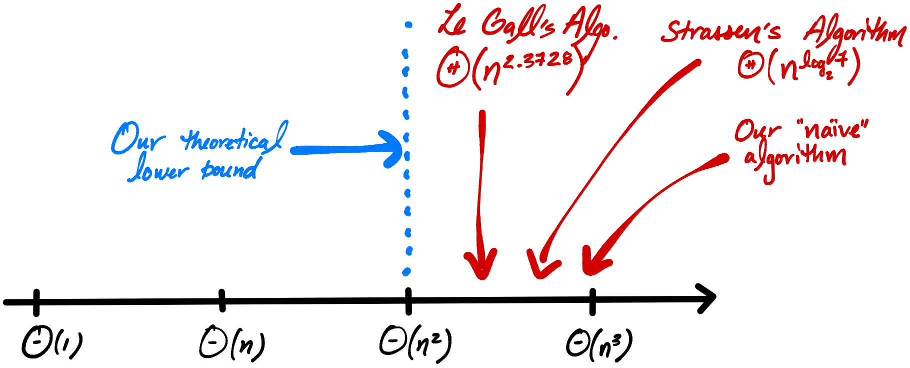


ω
3.0
2.9
2.8
2.7
2.6
2.5
2.4
naive
Strassen
Pan
Bini et al.
Schönhage
Romani
Coppersmith, Winograd
Strassen
Coppersmith, Winograd
Stothers
Le Gall
Williams
Year
1950
1960
1970
1980
1990
2000
2010
2015

Currently

- The fastest[2] known matrix multiplication algorithm is due to Le Gall.
- $\Theta(n^{2.3728639})$ time.

[2] In terms of asymptotic time complexity.


Le Gall's Algo.
$\Theta(n^{2.3728})$
Strassen's Algorithm
$\Theta(n^{\log_2 7})$
Our "naive" algorithm
Our theoretical lower bound
$\Theta(1)$
$\Theta(n)$
$\Theta(n^2)$
$\Theta(n^3)$

Interestingly...

- No one knows what the lowest possible time complexity is.
- It could be $\Theta(n^2)$!
- The “best” matrix multiplication algorithm is probably still undiscovered.

Irony

- There are many matrix multiplication algorithms.
- How fast is numpy's matrix multiply?
- $\Theta(n^3)$.

Why?

- Strassen *et al.* have better asymptotic complexity.
- But much (much!) larger “hidden constants”.
- Remember, which is better for small n: $999{,}999n^2$ or n^3?

Optimization

- Numpy, most others use **highly optimized** cubic time algorithms[3]

[3] The constant c in $T(n) = cn^3 + \ldots$ is actually much less than 1, as can be verified by timing.

Main Idea

No one knows what the lowest possible time complexity of matrix multiplication is, and some algorithms are approaching $\Theta(n^2)$.

But most useful implementations take $\Theta(n^3)$ time.