DSC 40B

Lecture 7 :Average, Expected, Lower Bound theory

Agenda

Plan for the lecture

- Average case, one more example.
- Expected Time
- Lower Bound Theory (can you do better?)

Average Case Time Complexity

- The **average case** time complexity of **linear search** is $\Theta(n)$.
  - Under *the 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?

Average Case in 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.

```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

- **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.

“Average” Case?

- 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?

A: $\Theta(1)$

B: $\Theta(n^2)$

C: Something in between, like $\Theta(n)$

The Movie Problem

```
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
```

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 (pairs of movies)

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?

Exercise

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

```
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
```

(m_1, m_2)

Exercise

```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
```

α	T

(m_1, m_2)

Exercise

```
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
```

α	T
1	1

(m_1, m_3)

Exercise

```
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
```

α	T
1	1
2	2

(m_1, m_4)

Exercise

```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
```

α	T
1	1
2	2
3	3

Exercise

```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
```

α	T
1	1
2	2
3	3
4	4

Exercise

```
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
```

α	T
1	1
2	2
3	3
4	4

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

Step 4: Compute Expectation

$$T_{avg} =$$

Step 4: Compute Expectation

$$T_{avg} = \sum_{\alpha = 1}$$

Step 4: Compute Expectation

$$T_{avg} = \sum_{\alpha = 1}^{\binom{n}{2}}$$

Step 4: Compute Expectation

$$T_{avg} = \sum_{\alpha = 1}^{\binom{n}{2}} P(case\ \alpha)$$

Step 4: Compute Expectation

$$T_{avg} = \sum_{\alpha=1}^{\binom{n}{2}} P(case\ \alpha) \cdot T(case\ \alpha)$$

$$T_{avg} = \sum_{\alpha=1}^{\binom{n}{2}} P(case\ \alpha) \cdot T(case\ \alpha)$$

$$\sum_{\alpha=1}^{\binom{n}{2}} \cdot \frac{1}{\binom{n}{2}} \cdot T(case\ \alpha)$$

$$T_{avg} = \sum_{\alpha=1}^{\binom{n}{2}} P(case\ \alpha) \cdot T(case\ \alpha)$$

$$\sum_{\alpha=1}^{\binom{n}{2}} \cdot \frac{1}{\binom{n}{2}} \cdot T(case\ \alpha) = \sum_{\alpha=1}^{\binom{n}{2}} \cdot \frac{1}{\binom{n}{2}} \cdot \alpha$$

$$\sum_{\alpha=1}^{\binom{n}{2}} \cdot \frac{1}{\binom{n}{2}} \cdot \alpha = \frac{1}{\binom{n}{2}} \cdot \sum_{\alpha=1}^{\binom{n}{2}} \alpha$$

$$\sum_{\alpha=1}^{\binom{n}{2}} \cdot \frac{1}{\binom{n}{2}} \cdot \alpha = \frac{1}{\binom{n}{2}} \cdot \sum_{\alpha=1}^{\binom{n}{2}} \alpha$$

$$\sum_{\alpha=1}^{\binom{n}{2}} \alpha = \sum_{\alpha=1}^{t} \alpha = \frac{t(t+1)}{2}$$

$$\sum_{\alpha=1}^{\binom{n}{2}} \cdot \frac{1}{\binom{n}{2}} \cdot \alpha = \frac{1}{\binom{n}{2}} \cdot \sum_{\alpha=1}^{\binom{n}{2}} \alpha$$

$$\sum_{\alpha=1}^{\binom{n}{2}} \alpha = \sum_{\alpha=1}^{t} \alpha = \frac{t(t+1)}{2} = \frac{\binom{n}{2}\left[\binom{n}{2}+1\right]}{2}$$

$$\sum_{\alpha=1}^{\binom{n}{2}} \cdot \frac{1}{\binom{n}{2}} \cdot \alpha = \frac{1}{\binom{n}{2}} \cdot \sum_{\alpha=1}^{\binom{n}{2}} \alpha$$

$$\sum_{\alpha=1}^{\binom{n}{2}} \alpha = \sum_{\alpha=1}^{t} \alpha = \frac{t(t+1)}{2} = \frac{\binom{n}{2}\left[\binom{n}{2}+1\right]}{2}$$

$$\binom{n}{2} = \frac{n!}{2!\,(n-2)!} = \frac{n(n-1)}{2} = \Theta(n^2)$$

$$\binom{n}{2} = \frac{n!}{2!\,(n-2)!} = \frac{n(n-1)}{2} = \Theta(n^2)$$

$$\frac{\binom{n}{2}\left[\binom{n}{2}+1\right]}{2} = \Theta(?)$$

$$\binom{n}{2} = \frac{n!}{2!\,(n-2)!} = \frac{n(n-1)}{2} = \Theta(n^2)$$

$$\frac{\binom{n}{2}\left[\binom{n}{2}+1\right]}{2} = \Theta(n^4)$$

$$\frac{1}{\binom{n}{2}} \cdot \sum_{\alpha = 1}^{\binom{n}{2}} \alpha$$

$$\frac{1}{\binom{n}{2}} \cdot \sum_{\alpha=1}^{\binom{n}{2}} \alpha = \frac{1}{\binom{n}{2}} \cdot \frac{\binom{n}{2}\left[\binom{n}{2}+1\right]}{2}$$

$$\frac{1}{\binom{n}{2}} \cdot \sum_{\alpha = 1}^{\binom{n}{2}} \alpha = \frac{1}{\binom{n}{2}} \cdot \frac{\binom{n}{2}\left[\binom{n}{2}+1\right]}{2}$$

$$= \frac{1}{\binom{n}{2}} \cdot \Theta(n^4)$$

$$\frac{1}{\binom{n}{2}} \cdot \sum_{\alpha=1}^{\binom{n}{2}} \alpha = \frac{1}{\binom{n}{2}} \cdot \frac{\binom{n}{2}\left[\binom{n}{2}+1\right]}{2}$$

$$= \frac{1}{\binom{n}{2}} \cdot \Theta(n^4)$$

$$= \frac{1}{\Theta(n^2)} \cdot \Theta(n^4)$$

$$\frac{1}{\binom{n}{2}} \cdot \sum_{\alpha=1}^{\binom{n}{2}} \alpha = \frac{1}{\binom{n}{2}} \cdot \frac{\binom{n}{2}\left[\binom{n}{2}+1\right]}{2}$$

$$= \frac{1}{\binom{n}{2}} \cdot \Theta(n^4)$$

$$= \frac{1}{\Theta(n^2)} \cdot \Theta(n^4)$$

$$= \Theta(n^2)$$

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.

T/F?

$O(\cdot)$ is worst case,

$\Omega(\cdot)$ is best case,

$\Theta(\cdot)$ is average case.

A: True

B: False

A Common Mistake

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

Note!

- $O(\cdot)$ expresses ignorance about a lower bound.
    - $O(\cdot)$ is like ≤ ("at most")
- $\Omega(\cdot)$ expresses ignorance about an upper bound.
    - $\Omega(\cdot)$ is like ≥ ("at least")
- $\Theta(\cdot)$ expresses ignorance about an upper bound.
    - $\Theta(\cdot)$ is like = ("roughly the same")

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!

Yet another way

- Best, worst, and average cases can each have their own bouds
  - Upper (big-O)
  - lower, (big-Omega)
  - tight (Big Theta)
- as these are distinct concepts: "case" describes a *specific* input characteristic.

Expected Time Complexity

Example: Contrived Algorithm

```
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!')
```

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):
    # 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!')
```

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

Step 2: Determine case probabilities

```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!')
```

- **Case 1**: `x == 0`
  - `P(Case 1) = 1/n`
- **Case 2**: `x != 0`
  - `P(Case 2)=1-1/n`

Step 3: Determine case times

```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!')
```

- **Case 1:** x == 0
  - T(Case 1): = **Θ(n)**
- **Case 2**: x != 0
  - T(Case 1): = **Θ(1)**

Step 4: Compute expectation

Compute expected time:

Step 4: Compute expectation

Compute expected time:

```
P(Case 1) * T(Case 1) + P(Case 2) * T(Case 2)
```

Step 4: Compute expectation

Compute expected time:

P(Case 1) * T(Case 1) + P(Case 2) * T(Case 2) =

1/n * Θ(n)

Step 4: Compute expectation

Compute expected time:

P(Case 1) * T(Case 1) + P(Case 2) * T(Case 2) =

1/n * Θ(n) + (1-1/n) * Θ(1)

Step 4: Compute expectation

Compute expected time:

P(Case 1) * T(Case 1) + P(Case 2) * T(Case 2) =

1/n * Θ(n) + **(1-1/n)** * Θ(1) =

1/n * Θ(n) + (**(n-1)/n**))* Θ(1) =

Step 4: Compute expectation

Compute expected time:

$$P(\text{Case } 1) * T(\text{Case } 1) + P(\text{Case } 2) * T(\text{Case } 2) =$$

$$1/n * \Theta(n) + \mathbf{(1-1/n)} * \Theta(1) =$$

$$1/n * \Theta(n) + (\mathbf{(n-1)/n})) * \Theta(1) =$$

$$\Theta(1) + \Theta(1) = \mathbf{\Theta(1)}$$

Expected Time

This was a *contrived* example.

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

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 possibly 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**.

Definition

A **tight theoretical lower bound** for a problem is the **fastest** possible worst-case time complexity of any algorithm solving that problem.

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 DSC 40B.

Assumptions make problems easier

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

Exercise

A: **Constant**

B: **n**

C: $\mathbf{n^2}$

D: **Something else**

- Consider these **two** problems:
  - Find the min of a sorted array.
  - 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**!

Practice makes perfect

- dsc40b.com/practice has a dozen more examples of finding theoretical lower bounds.

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

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

Recall

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

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 5 & -1 \\ 1 & 7 \\ -2 & -3 \end{pmatrix} = \begin{pmatrix} \mathbf{x} & \\ & \\ & \end{pmatrix}$$

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.

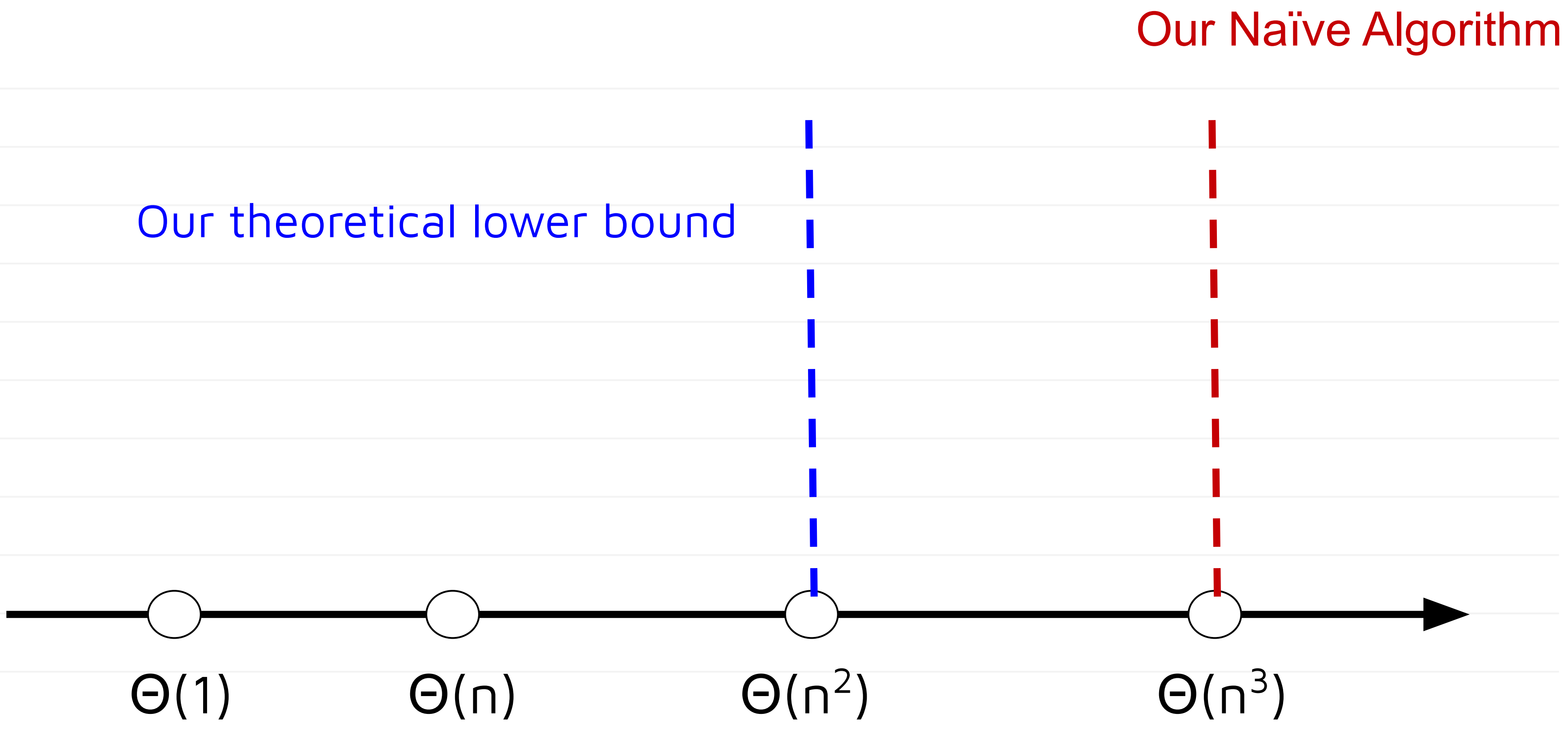
1 s	10 m	1 hr
1,000	6,694	15,326

The Question

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

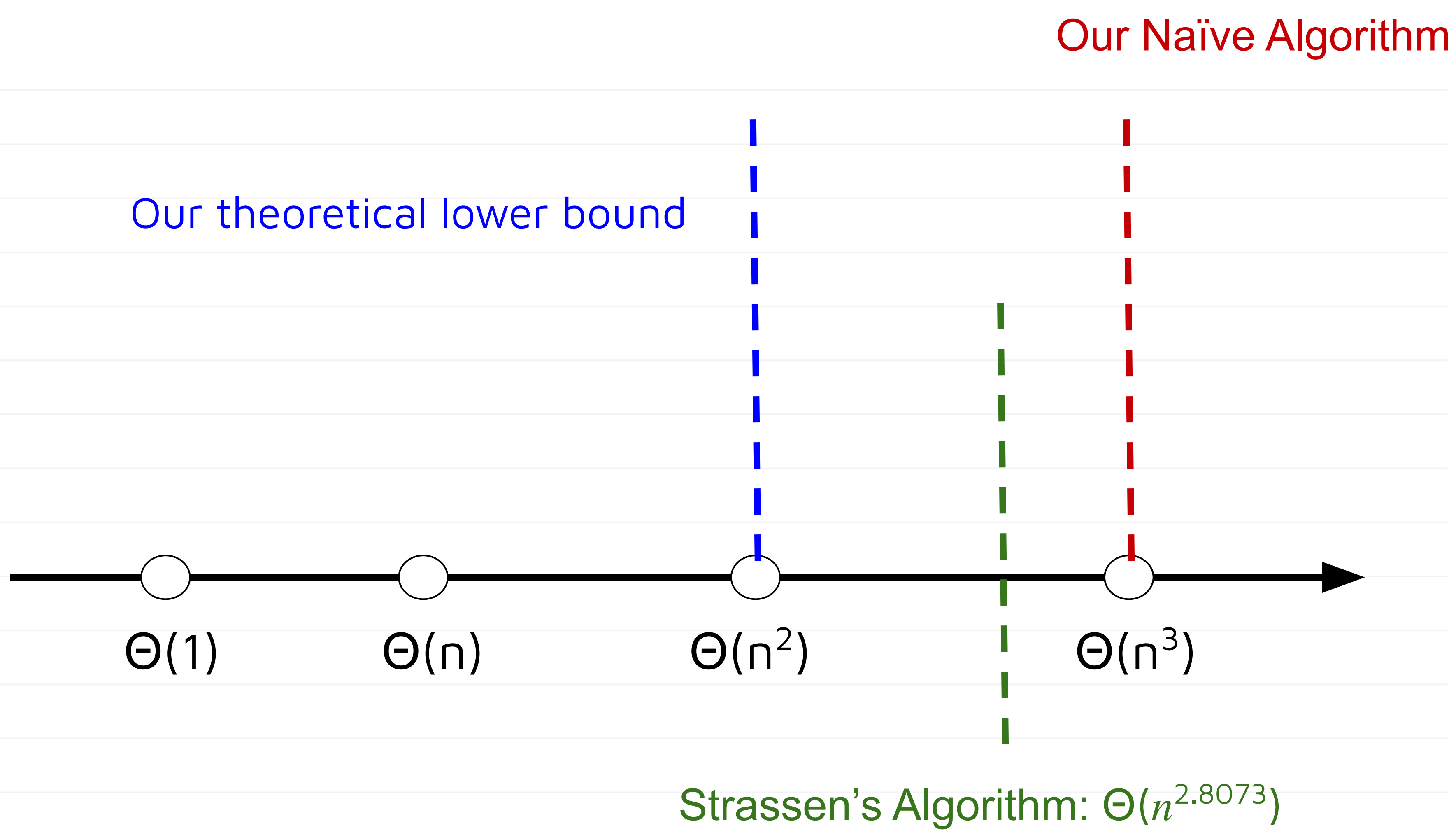
Theoretical Lower Bound

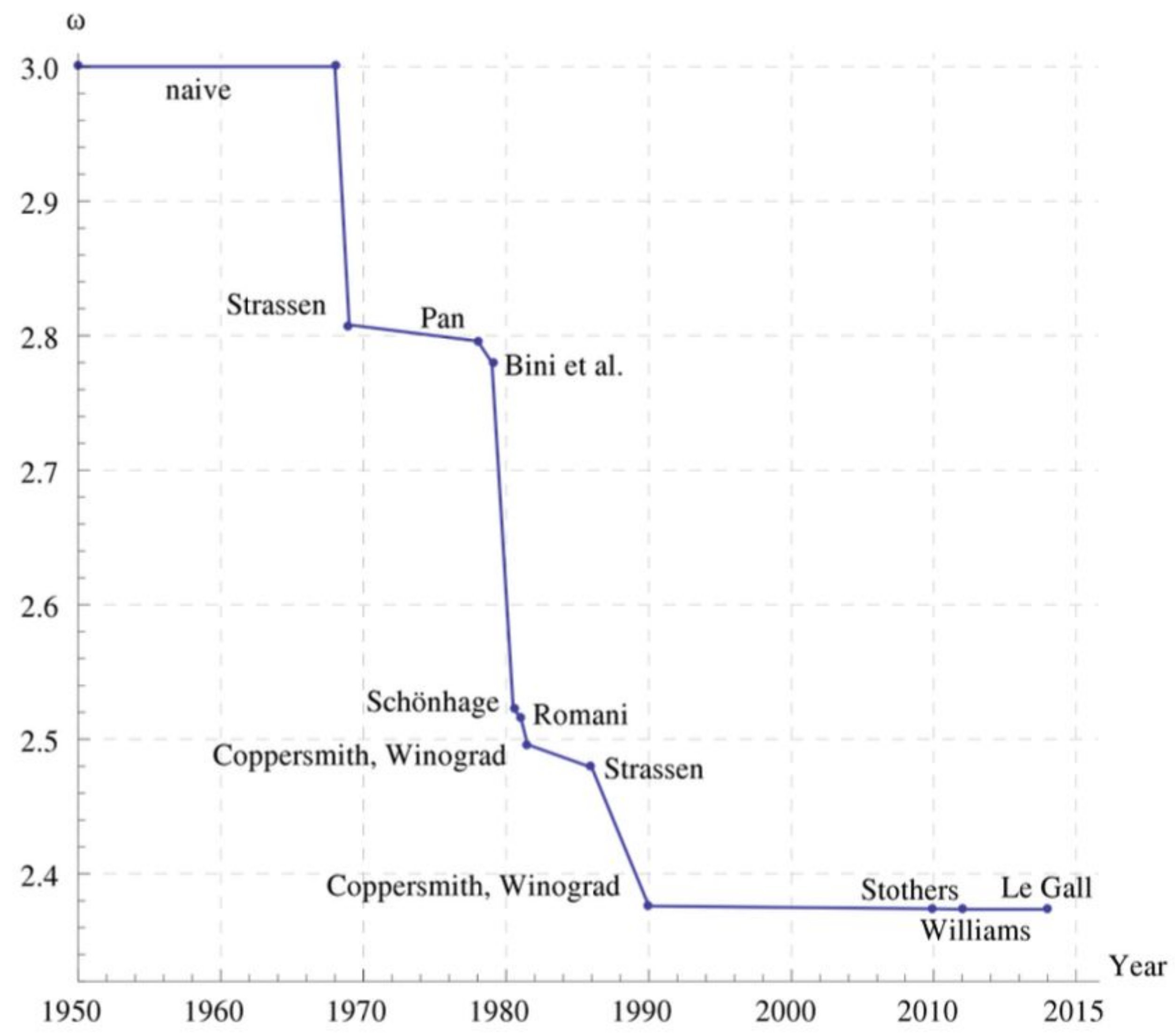
- If A and B are $n \times n$, C will have n^2 entries.
- Each entry must be filled: $\Omega(n^2)$ time.
- That is, matrix multiplication must take at least quadratic time.
- Is this bound **tight**? Can it be increased?


Our Naïve Algorithm
Our theoretical lower bound
Θ(1)
Θ(n)
Θ(n^2)
Θ(n^3)

Strassen's Algorithm

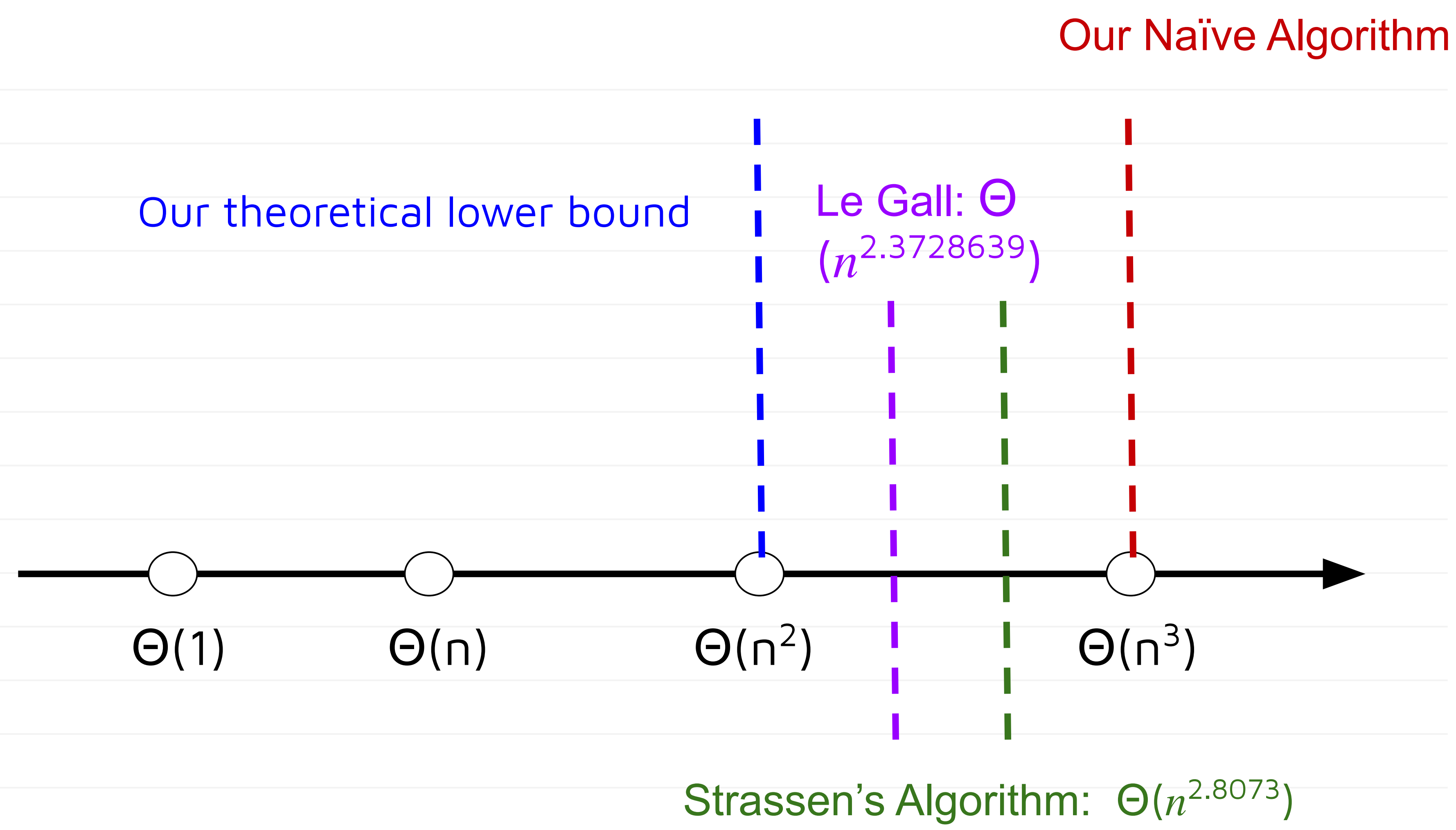
- Cubic was as good as it got…
- …until Strassen, 1969.
- Time complexity: $\Theta(n^{\log_2 7}) = \Theta(n^{2.8073})$


Our Naïve Algorithm
Our theoretical lower bound
Θ(1)
Θ(n)
Θ(n²)
Θ(n³)
Strassen's Algorithm: Θ(n^2.8073)


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 known matrix multiplication algorithm is due to Le Gall
  - In terms of asymptotic time complexity.
- $\Theta(n^{2.3728639})$ time.


Our Naïve Algorithm
Our theoretical lower bound
Le Gall: Θ ($n^{2.3728639}$)
Θ(1)
Θ(n)
Θ(n^2)
Θ(n^3)
Strassen's Algorithm: Θ($n^{2.8073}$)

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?

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:

$$999{,}999n^2 \text{ or } n^3?$$

Thank you!

Do you have any questions?

CampusWire!