DSC 40B

Theoretical Foundations II

Average Case

Average Case

- Best case and worst case can be **misleading**.
  - Depend on a **single good/bad input**.
- How long does algorithm take on a **typical input**?

Linear Search

- **Best Case**: target is right at the beginning, $\Theta(1)$.
- **Worst Case**: target is at end (or missing), $\Theta(n)$.
- **Typical Case**: target is somewhere in the middle, $\Theta(n/2) = \Theta(n)$.

More Formally

- Recall: the **expectation**.

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

- Expected winnings:

Expected Time

- We'll compute the **expected time**:

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

- Also called the **average case time complexity**.

Average Case Complexity

- 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).

Step 1: Determine the Cases

- Example: linear search.

$$\begin{aligned}
\text{Case 1:}\quad & \text{target is first element}\\
\text{Case 2:}\quad & \text{target is second element}\\
& \vdots\\
\text{Case } n\text{:}\quad & \text{target is } n\text{th element}\\
\text{Case } n+1\text{:}\quad & \text{target is not in array}
\end{aligned}$$

Step 2: Case Probabilities

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

- Often requires making **assumptions**.
  - Example: target is equally likely to be in any position, but *must* be in array.

- Often guarantee assumptions by **randomizing**.
  - Example: randomly shuffle input array.

Example

- **Assume**: target is in the array exactly once
- **Randomize** the array (in linear time)
- Each case is **equally likely**. Probability: $1/n$.

Step 3: Case Times

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

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

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 of **linear search** is $\Theta(n)$, as expected.

Note

- **Worst case** time complexity is still useful.
- Easier to calculate.
- Often same as average case (but not always!)
- Sometimes worst case is very important.
  - Real time applications, time complexity attacks

DSC 40B

Theoretical Foundations II

Average Case in Movie Problem

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 is the best case time complexity of `find_movies`? What about the worst case?

Time Complexity

- Best case: $\Theta(1)$
- Worst case: $\Theta(n^2)$
- Average case: $\Theta(?)$

Step 1: Determine the Cases

- Each possible 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

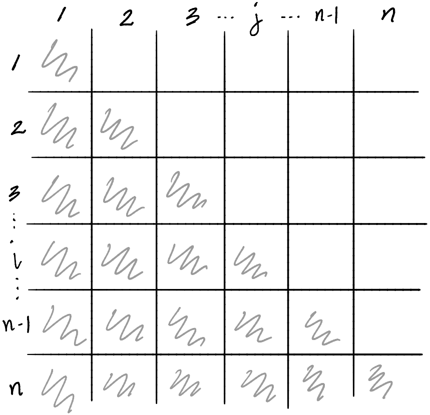
- How much time is taken for a particular case?
- Suppose the 4th and 11th movie sum to the target.
- How long does it take to find this pair?

```
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 how many times does line 5 execute if the 4th and 11th movies add to the target?

Visualization



Average Case

- The average case time complexity is $\Theta(n^2)$.

DSC 40B

Theoretical Foundations II

Lower Bound Theory

Problems and Algorithms

- There can be many **algorithms** for solving the same **problem**.
- Some have better time complexity than others.
- **Very important question**: For a given problem, what is the **best possible** time complexity?

Lower Bounds

- No algorithm can have a better (worst case) time complexity than a **theoretical lower bound**.
- E.g.: $\Omega(n)$ is a lower bound for movie problem.
- Any algorithm for the movie problem must take $\Omega(n)$ time in the worst case.

Definition

$f(n)$ is a **theoretical lower bound** for a problem if every possible algorithm's worst case complexity is $\Omega(f(n))$.

Main Idea

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

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.
- Theoretical Lower Bound: $\Omega(n)$.
  - Why? In the worst case, *every* algorithm has to look through all n numbers.

Useless Lower Bounds

- $\Omega(1)$ is also a theoretical lower bound.
- But it is useless... no algorithm exists.

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**.

The Question

- What is the **best possible** time complexity for an algorithm solving the movie problem?

- $\Omega(n)$ is a non-trivial lower bound.
  - Must read every movie duration in worst case.

- Is this a tight bound? Is there such an algorithm?

DSC 40B

Theoretical Foundations II

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

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

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

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.

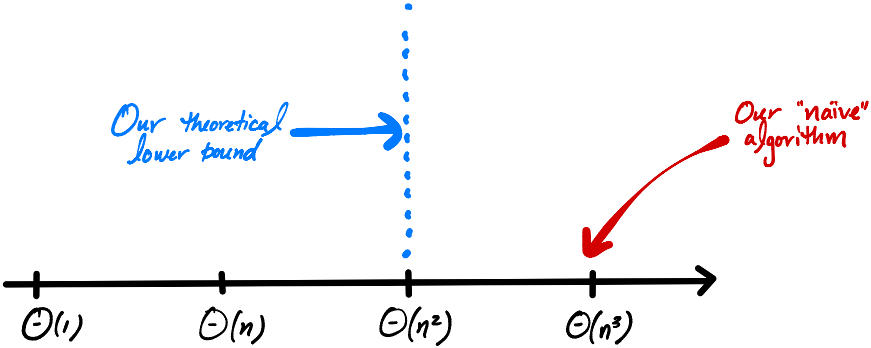
1 s	10 m	1 hr
1,000	6,694	15,326

The Question

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

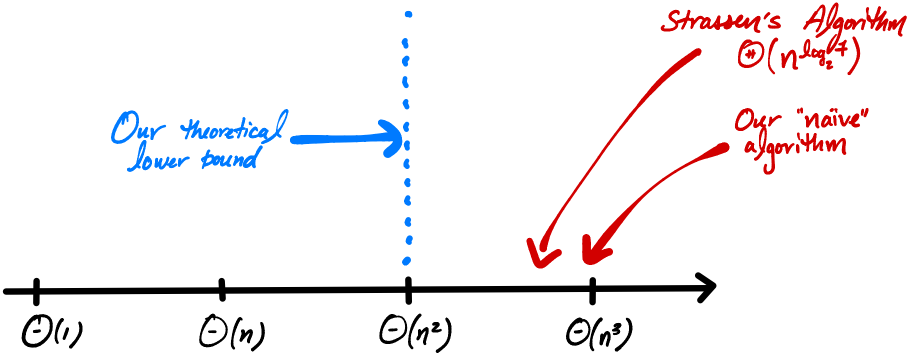
Theoretical Lower Bound

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

Strassen's Algorithm

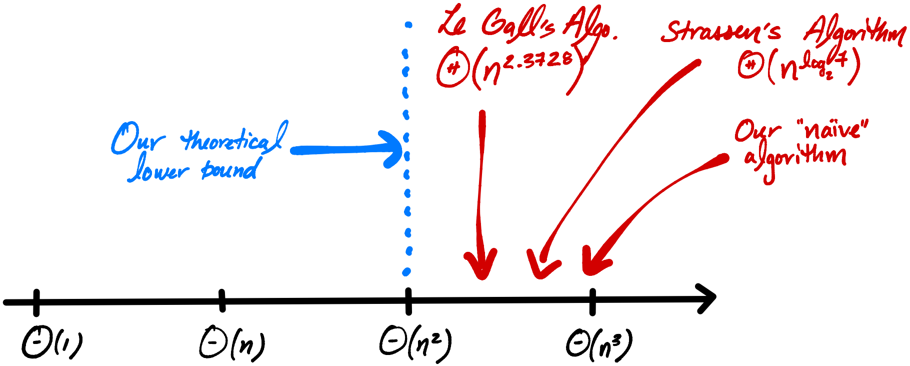
- Cubic was as good as it got...
- ...until Strassen, 1969.
- Time complexity: $\Theta(n^{\log_2 7}) = \Theta(n^{2.8073})$


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

Currently

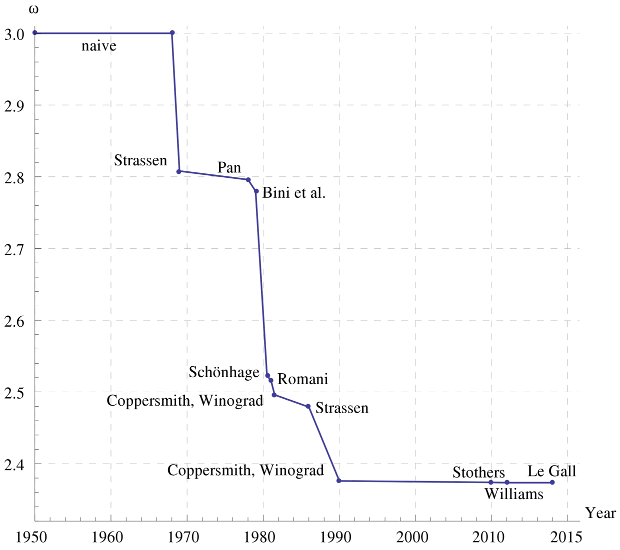
- The fastest[1] known matrix multiplication algorithm is due to Le Gall.
- $\Theta(n^{2.3728639})$ time.

[1]In terms of asymptotic time complexity.


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

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.


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

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.

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.