DSC 40B

Theoretical Foundations II

Lecture 4 | Part 1

News

News

- Homework 02 is posted
  - Due: Wednesday, April 17

DSC 40B
Theoretical Foundations II

Lecture 4 | Part 2

Best and Worst Cases

Example 1: `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.

Example 2: 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
```

Observation

- It looks like there are *two* extreme cases…

The Best Case

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

The Worst Case

- When the target, `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.

- It has no single, overall time complexity.

- 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 asymptotic 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 asymptotic time complexity**.

Worst Case

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

Exercise

What are the best case and worst case time complexities of the following code?

```
def foo(arr):
    n = len(arr)
    for x in arr:
        for y in arr:
            if x + y == 10:
                return sum(arr)
    return None
```

Best Case

- When the first element is 5, so `x + y == 10`.
- `sum(arr)` takes $\Theta(n)$ time.
- Exits, taking $\Theta(n)$ time in total.

Worst Case

- No two elements sum to 10.
- Has to loop over all $\Theta(n^2)$ pairs.
- Worst case time complexity: $\Theta(n^2)$.
- **Note**: it's not $\Theta(n^3)$, since the `sum(arr)` only runs once!

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.

DSC 40B

Theoretical Foundations II

Lecture 4 | Part 3

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:

$0 \times (0.5) + 1 \times (0.3) + 10 \times (0.18)$

$+ 50 \times (0.02) =$

$0 + 0.3 + 1.8 + 1 = 3.1$

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**:

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

- Best case? Worst case?

Example: Linear Search

- 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
	⋮
Case n:	target is nth element
Case $n+1$:	target is not in array

Example:

		Pr
Case 1:	"	$\frac{1}{2n}$
Case 2:	"	$\frac{1}{2n}$
⋮		⋮
Case n+1:	target not in array	$\frac{1}{2}$

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? $\frac{1}{n}$
- 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

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

$$= \sum_{i=1}^{n} \frac{1}{n} \cdot c \cdot i$$

$$= \frac{c}{n} \sum_{i=1}^{n} i$$

$$= \frac{c}{n} \cdot \frac{n(n+1)}{2} = \frac{c(n+1)}{2} = \Theta(n)$$

Average Case Time Complexity

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

[1]Under these assumptions on the input!

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

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?

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

$$= \sum_{i=1}^{n} \frac{1}{2n} \cdot c \cdot i + \frac{1}{2} \cdot c \cdot n = \frac{c}{2n} \sum_{i=1}^{n} i + \frac{c}{2} \cdot n = \Theta(n)$$

$$\frac{c}{2n} \sum_{i=1}^{n} i = \Theta(n), \qquad \frac{c}{2} \cdot n = \Theta(n)$$

DSC 40B

Theoretical Foundations II

Lecture 4 | Part 4

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
```

Exercise

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

```
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?

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

Time Complexity

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

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
```

Pair (i, j)
(0, 1)
(0, 2)
(0, 3)
⋮
(0, n−1)
⋮
(n−2, n−1)

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

pair (i,j)	Probability	time
(0, 1)	$\frac{2}{n(n-1)}$	c
(0, 2)	$\frac{2}{n(n-1)}$	$2 \cdot c$
(0, 3)	$\frac{2}{n(n-1)}$	$3 \cdot c$
⋮	⋮	⋮
(n-2, n-1)	$\frac{2}{n(n-1)}$	$\frac{n(n-1)}{2} \cdot c$

$$\sum_{\alpha=1}^{K} P(\text{case } \alpha) \cdot T(\text{case } \alpha)$$

$$= \sum_{\alpha=1}^{K} \frac{2}{n(n-1)} \cdot \alpha \cdot c$$

$$= \frac{2c}{n(n-1)} \sum_{\alpha=1}^{K} \alpha$$

$$= \frac{2}{n(n-1)} \cdot \frac{K(K+1)}{2}$$

$$K = \binom{n}{2} = \Theta(n^2)$$

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

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.

DSC 40B

Theoretical Foundations II

Lecture 4 | Part 5

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

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

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

- $P(\text{Case } 1) = 1/n$
- $P(\text{Case } 2) = (n-1)/n$

$$= 1 - \frac{1}{n} = \frac{n-1}{n}$$

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:

$$P(\text{Case } 1) \cdot T(\text{Case } 1) + P(\text{Case } 2)\, T(\text{Case } 2)$$

$$= \frac{1}{n} \cdot \Theta(n) + \frac{n-1}{n} \cdot \Theta(1)$$

$$= \Theta(1) + \Theta(1) = \Theta(1)$$

Expected Time

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

DSC 40B

Theoretical Foundations II

Lecture 4 | Part 6

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 DSC 40B.

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**!

DSC 40B

Theoretical Foundations II

Lecture 4 | Part 7

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

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

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

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.

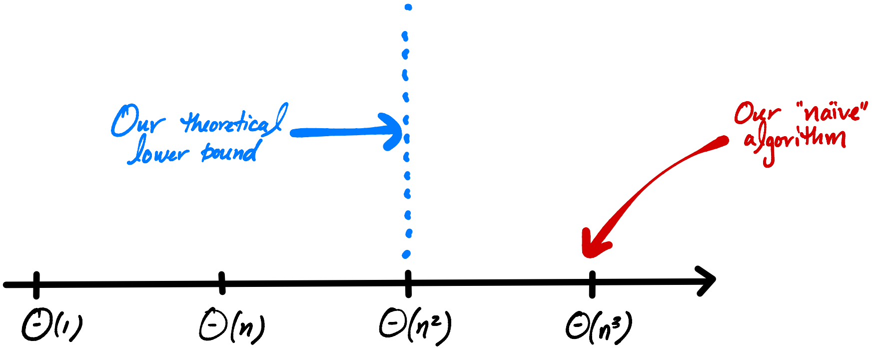
1 s	10 m	1 hr
1,000	6,694	15,326

The Question

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

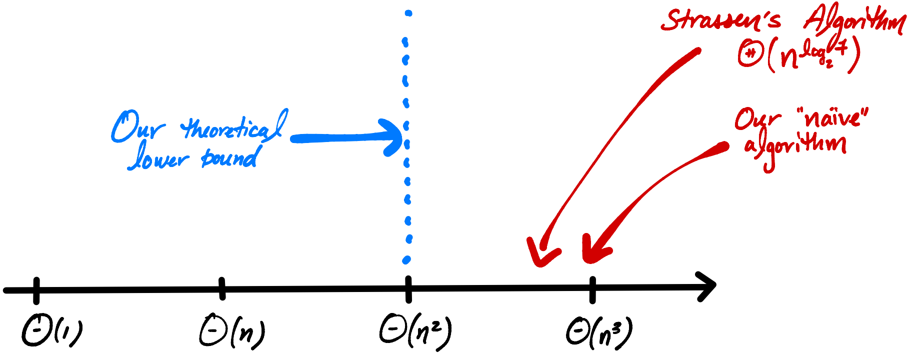
Theoretical Lower Bound

- 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?

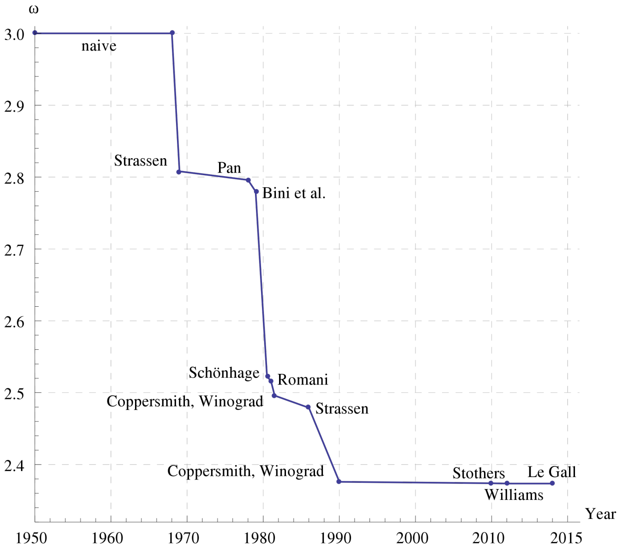

Our theoretical lower bound
Our "naïve" algorithm
Θ(1)
Θ(n)
Θ(n²)
Θ(n³)

Strassen's Algorithm

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



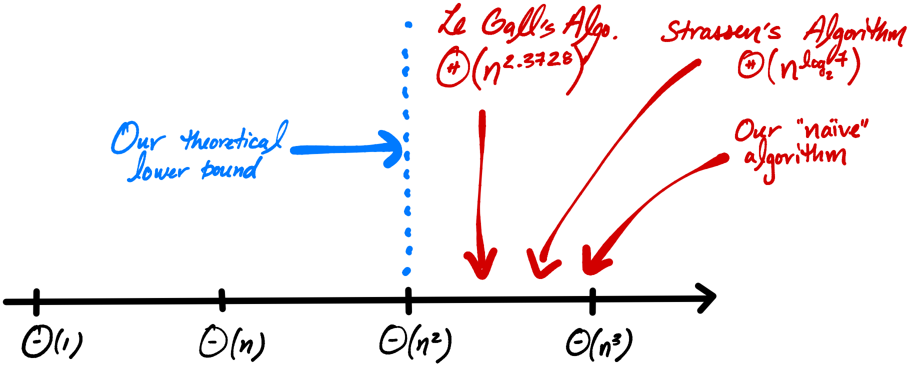
Strassen's Algorithm
$\Theta(n^{\log_2 7})$
Our "naïve" algorithm
Our theoretical lower bound
$\Theta(1)$
$\Theta(n)$
$\Theta(n^2)$
$\Theta(n^3)$


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
Williams
Le Gall
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 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.

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: $999{,}999n^2$ 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.