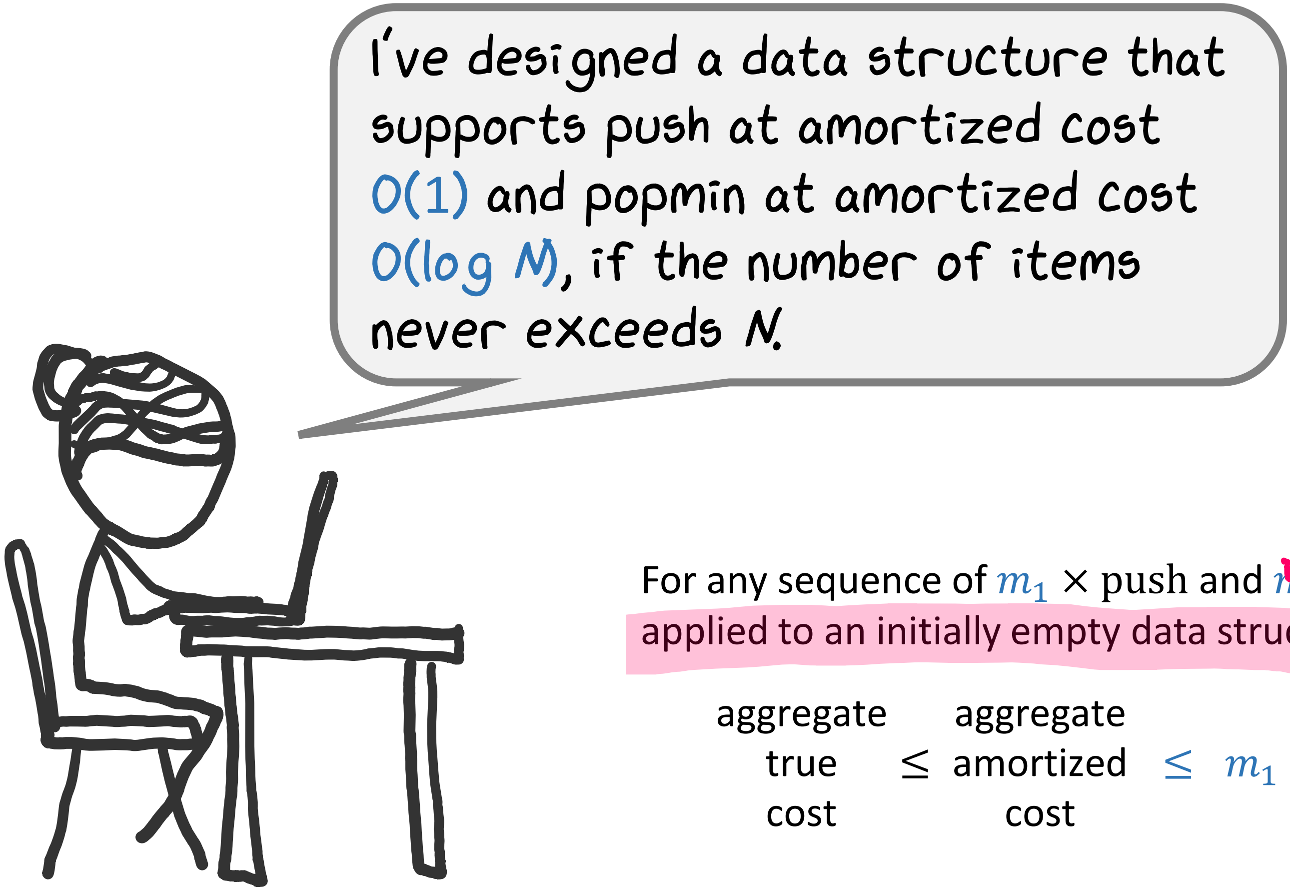
For advanced data structures like a Priority Queue

- We should care about the aggregate cost of a sequence of operations
- This might not be as bad as the per-operation worst cases suggest
- Amortized costs are a handy way to reason about aggregate costs

For advanced data structures like a Priority Queue

- We should care about the aggregate cost of a sequence of operations
- This might not be as bad as the per-operation worst cases suggest
- Amortized costs are a handy way to reason about aggregate costs

page 55



Ex. sheet 6 q.6 asks you to think through why this is a sensible restriction

For any sequence of $m_1 \times$ push and $m_2 \times$ popmin,
applied to an initially empty data structure,

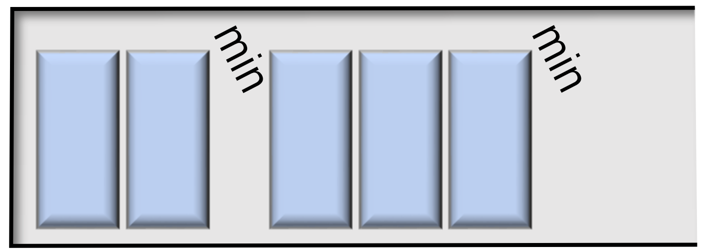
$$\text{aggregate true cost} \le \text{aggregate amortized cost} \le m_1\,O(1) + m_2\,O(\log N) = O(m_1 + m_2 \log N)$$

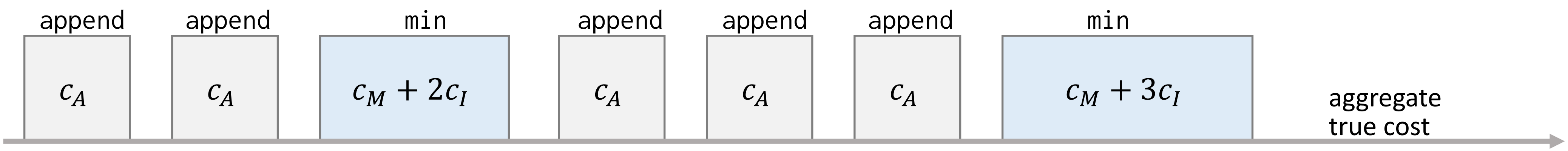
SECTION 7.4

Potential functions

or, how on earth do we come up with useful amortized costs?

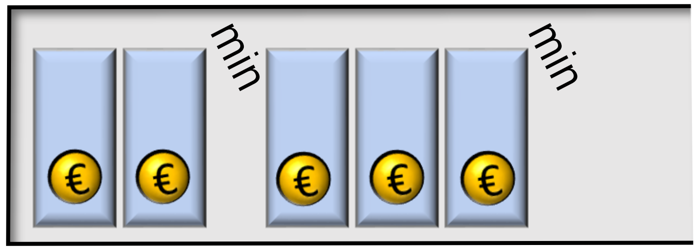
```
class MinList<T>:

    def append(T value):
        # append a new value

    def T min():
        # caches the result, so we
        # only need to iterate over
        # newly-appended items
```

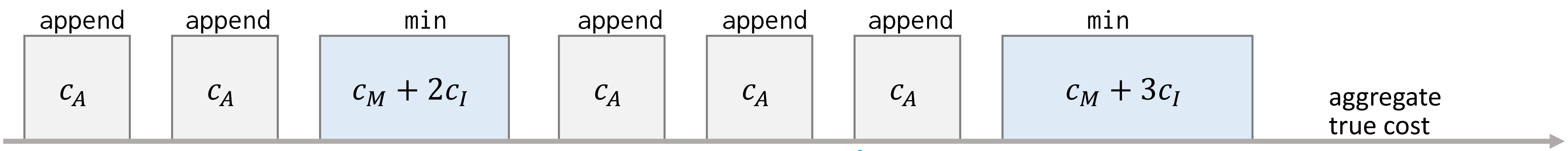




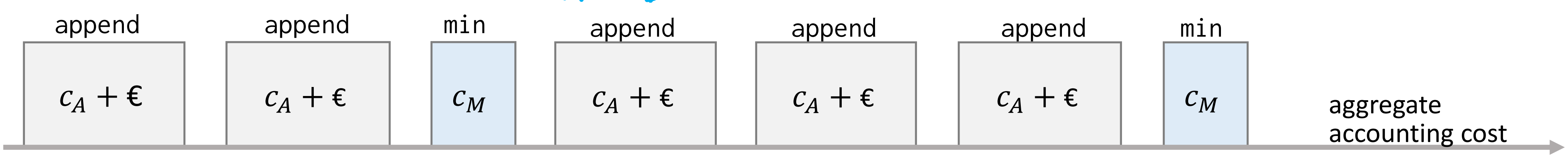
- Suppose we can store ‘credit’ in the data structure, and operations can either store or release credit
- Let the ‘accounting’ cost of an operation be: $\begin{pmatrix}\text{accounting}\\ \text{cost}\end{pmatrix} = \begin{pmatrix}\text{true}\\ \text{cost}\end{pmatrix} + \begin{pmatrix}\text{credit}\\ \text{it stores}\end{pmatrix} - \begin{pmatrix}\text{credit}\\ \text{it releases}\end{pmatrix}$
- Let’s ‘pay ahead’ for the potentially-costly operations


```
class MinList<T>:
    def append(T value):
        # append a new value
    def T min():
        # caches the result, so we
        # only need to iterate over
        # newly-appended items
```





acc. cost $= c_M + 2c_I + 0 - 2€ = c_M$



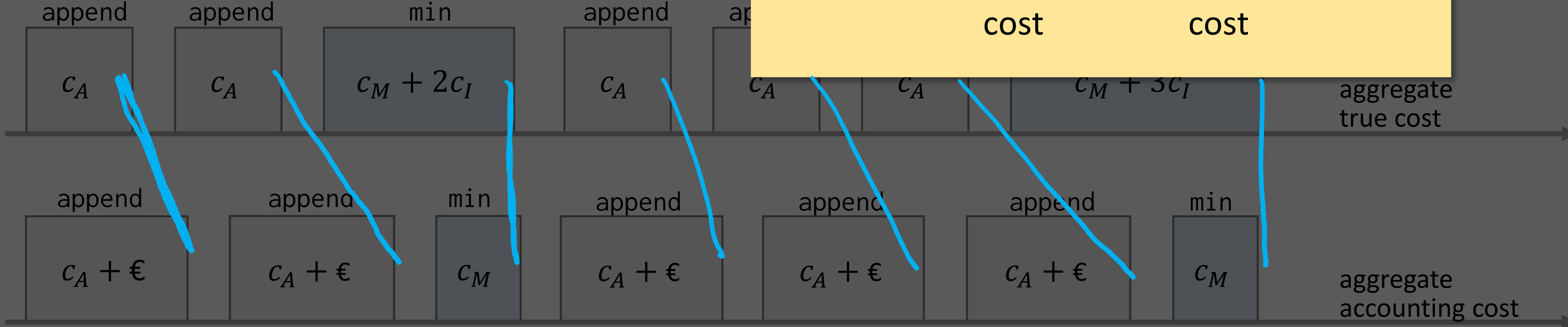
Store $1€ = c_I$

- Suppose we can store 'credit' in the data structure, and operations can either store or release credit
- Let the 'accounting' cost of an operation be: $\begin{pmatrix}\text{accounting}\\ \text{cost}\end{pmatrix} = \begin{pmatrix}\text{true}\\ \text{cost}\end{pmatrix} + \begin{pmatrix}\text{credit}\\ \text{it stores}\end{pmatrix} - \begin{pmatrix}\text{credit}\\ \text{it releases}\end{pmatrix}$
- Let's 'pay ahead' for the potentially-costly operations


```
class MinList<T>:
    def append(T value):
        # append a new value
    def T min():
        # caches the result, so we
        # only need to iterate over
        # newly-appended items
```

These are valid amortized costs
i.e. for any sequence of operations on an initially-empty data structure

$$\begin{matrix}\text{aggregate}\\ \text{true}\\ \text{cost}\end{matrix} \leq \begin{matrix}\text{aggregate}\\ \text{amortized}\\ \text{cost}\end{matrix}$$



page 57
page 61

Let Ω be the set of all states our data structure might be in.

A function $\Phi: \Omega \to \mathbb{R}$ is called a **potential function** if

$$\Phi(\mathcal{S}) \geq 0 \quad \text{for all} \quad \mathcal{S} \in \Omega$$

$$\Phi(\text{empty}) = 0$$

Φ = bank balance

= total amount of credit stored in the data structure.

For an operation $\underset{\text{state before}}{\mathcal{S}_{\text{ante}}} \to \underset{\text{state after}}{\mathcal{S}_{\text{post}}}$ with true cost c, define the **accounting cost** to be

$$c' = c + \Phi(\mathcal{S}_{\text{post}}) - \Phi(\mathcal{S}_{\text{ante}})$$

THE 'POTENTIAL' THEOREM: These are valid amortized costs.

PROOF: Consider an arbitrary sequence of operations, starting from empty: $\underset{\text{empty}}{\mathcal{S}_0} \xrightarrow{c_1} \mathcal{S}_1 \xrightarrow{c_2} \mathcal{S}_2 \to \cdots \xrightarrow{c_m} \mathcal{S}_m$

$$\begin{aligned}
\text{aggregate accounting cost} &= c_1' + c_2' + \cdots + c_m' \\
&= -\Phi(\mathcal{S}_0) + c_1 + \cancel{\Phi(\mathcal{S}_1)} \\
&\qquad - \cancel{\Phi(\mathcal{S}_1)} + c_2 + \cancel{\Phi(\mathcal{S}_2)} \\
&\qquad \cdots - \cancel{\Phi(\mathcal{S}_{m-1})} + c_m + \Phi(\mathcal{S}_m) \\
&= \underbrace{-\cancel{\Phi(\mathcal{S}_0)}}_{0} + \underbrace{c_1 + \cdots + c_m}_{\text{agg. true cost}} + \Phi(\mathcal{S}_m) \\
&\geq \text{aggregate true cost}
\end{aligned}$$

Example

Consider a dynamically-sized array to which we append items. It starts with capacity 1, and doubles its capacity whenever it becomes full.

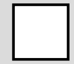
Suppose the cost of writing an item is 1, and the cost of doubling capacity from m to $2m$ (and copying across the existing items) is κm. $O(m)$

Show that the amortized cost of append is $O(1)$.

Φ = # items added since last doubling $\times 2\kappa$

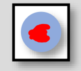
We want the coins that we've stored so far (2€) to pay for the doubling (cost 4κ). So we want 1€=2κ.

initially empty



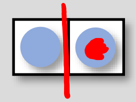
$\Phi = 0$

$c=1$ append() am. cost $c' = c + \Delta\Phi = 1 + 2\kappa$



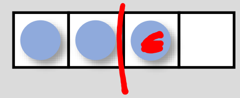
$\Phi = 2\kappa$

$c = \kappa + 1$ append(), requires doubling $c' = c + \Delta\Phi = 1 + \kappa$



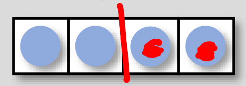
$\Phi = 2\kappa$

$c = 2\kappa + 1$ append(), requires doubling $c' = 1 + 2\kappa$



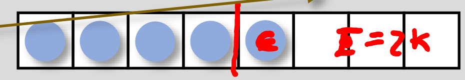
$\Phi = 2\kappa$

$c = 1$ append() $c' = 2\kappa + 1$



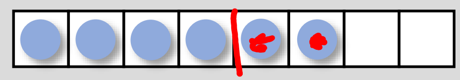
$\Phi = 4\kappa$

$c = 4\kappa + 1$ append(), requires doubling $c' = 4\kappa + 1 - 2\kappa = 1 + 2\kappa$



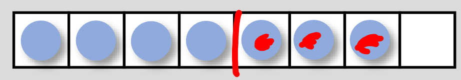
$\Phi = 2\kappa$

$c = 1$ append() $c' = 1 + 2\kappa$



$\Phi = 4\kappa$

$c = 1$ append() $c' = 1 + 2\kappa$



$\Phi = 6\kappa$

Am cost is always $\le 1 + 2\kappa = O(1)$

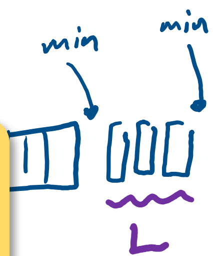
Example (sloppy style)

Consider a dynamically-sized array to which we append items. It starts with capacity 1, and doubles its capacity whenever it becomes full.

Suppose the cost of writing an item is $O(1)$, and the cost of doubling capacity from m to $2m$ (and copying across the existing items) is $O(m)$.

Show that the amortized cost of append is $O(1)$.

```
class MinList<T>:

    def append(T value):
        # append a new value

    def T min():
        # return the smallest
        # (without removing it)
```



QUESTION. What potential function might we use, to show that append and `min` both have amortized cost $O(1)$?

L = # items added since last min

$\Phi = L$

Stage 0

- Use a linked list
- `min` iterates over the entire list

Stage 1

- Use a linked list
- `min` caches its result, so that next time it only needs to iterate over newer values

Stage 2

- Use a linked list
- Store the current minimum, and update it on every append

Stage 3

- `min` caches its result, the same as Stage 1
- ... but we argue it's just as good as Stage 2

For one-shot algorithms such as sorting:

After we show that our algorithm is $O(n \log n)$, it's good manners to also show that the worst case is $\Omega(n \log n)$.

$\exists \kappa > 0$ such that, for all sufficiently large n,

$$\text{cost}_n \leq \kappa\, n \log n$$

$\exists \delta > 0$ and a sequence of example inputs with increasing n such that

$$\text{cost}_n \geq \delta\, n \log n$$

i.e. design a family of example inputs of increasing size n where

$$\text{cost}_n = \Omega(n \log n)$$

and, if we can't find matching O-Ω bounds, then maybe our O bound isn't as good as it could be.

For advanced data structures:

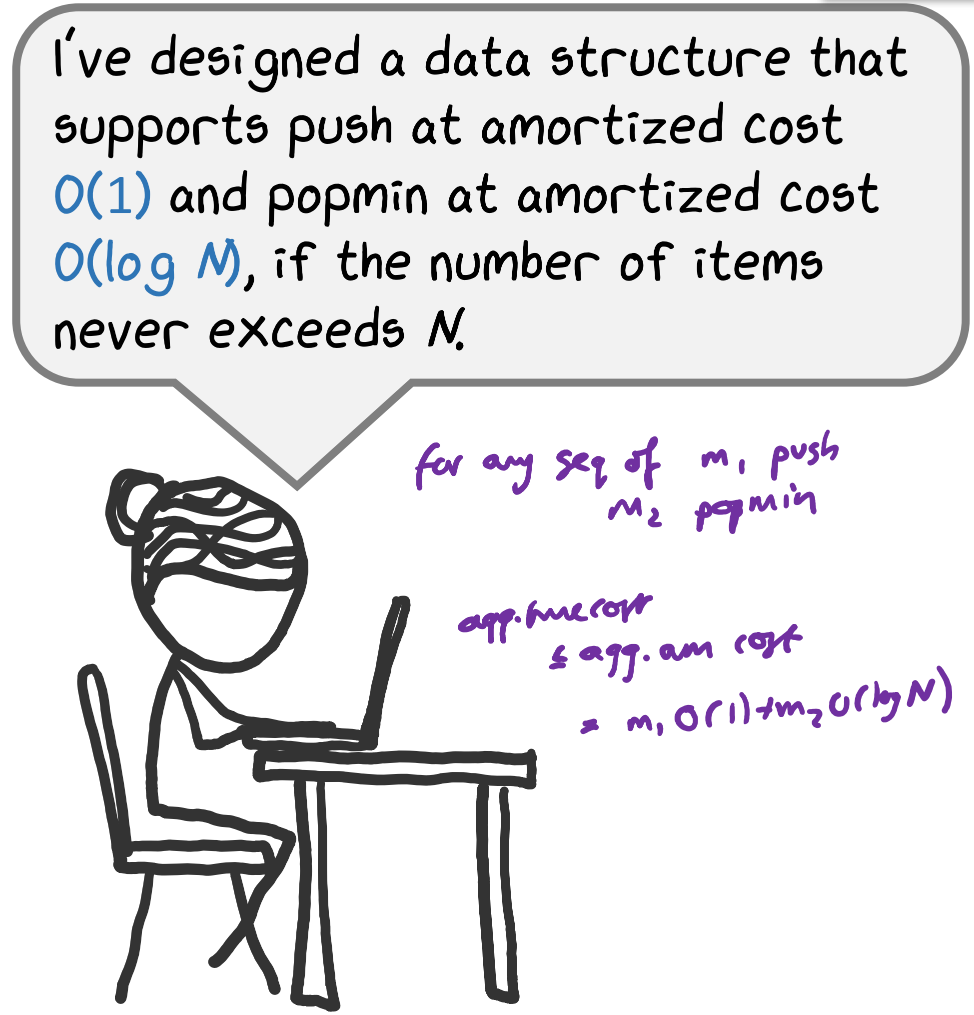
After we find big-O upper bounds for amortized costs, it's good manners to show matching worst-case performance.

$\exists \kappa > 0$ such that, for all sufficiently large N, and any operation-sequence s having $m_1 \times \text{push} + m_2 \times \text{popmin}$ such that #items is always $\leq N$,

$$\text{cost}_s \leq \kappa\,(m_1 + m_2 \log N)$$

Design a family of operation-sequences $s(N)$ having $m_1(N) \times \text{push} + m_2(N) \times \text{popmin}$ such that #items is always $\leq N$, and

$$\text{cost}_{s(N)} = \Omega(m_1(N) + m_2(N) \log N)$$



for any seq of m_1 push, m_2 popmin

agg. true cost $\leq$ agg. am cost $= m_1 O(1) + m_2 O(\log N)$

and, if we can't find matching O-Ω bounds, then maybe our amortized costs aren't as good as they could be.
[See ex.sheet 6 q.7] V. Important question.

Fully Connected 2023

Schedule

Thu Mar 16 2023 at 10:00 am to 05:30 pm
UTC+00:00

Location

Computer Laboratory, University of Cambridge | Cambridge, EN

Interested in ML research? Wondering what that ChatGPT thing is all about? Want a free lunch? Come to the CL!

About this Event

Interested in ML research but don't know where to start? Looking for a summer research project? Wondering what that ChatGPT thing is all about? Want a free lunch? Come along to **Fully Connected, Cambridge's flagship machine learning research event for undergraduates!**