For advanced data structures like a Python list or 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

**TODAY**

❖ **Amortized costs and potential functions** are a handy way to reason about aggregate costs
class MinList<T>:
    def append(T value):
        # append a new value
    def flush():
        # empty the list
    def foreach(f):
        # do f(x) for each item
    def T min():
        # return the smallest
        # (without removing it)

let \( n = \text{#items} \)

Stage 0
- Use a linked list
- \( \text{min} \) iterates over the entire list
  
append is \( \Theta(1) \)

min is \( \Theta(n) \)

Stage 1
- Use a linked list
- \( \text{min} \) caches its result, so that next time it only needs to iterate over newer values

min is \( \Theta(n) \)
in the worst case

Stage 2
- Use a linked list
- Store the current minimum, and update it on every append

append is \( \Theta(1) \)

min is \( \Theta(1) \)

Stage 3
- \( \text{min} \) caches its result, the same as Stage 1
- ... but we argue it’s just as good as Stage 2
Stage 3

- $\min$ caches its result, the same as Stage 1
- ... but we argue it’s just as good as Stage 2
FUNDAMENTAL INEQUALITY OF AMORTIZATION

Let there be a sequence of $m$ operations, applied to an initially-empty data structure, whose true costs are $c_1, c_2, \ldots, c_m$. Suppose someone invents $c'_1, c'_2, \ldots, c'_m$. These are called amortized costs if

$$c_1 + \cdots + c_j \leq c'_1 + \cdots + c'_j$$

for all $j \leq m$. In words: for any sequence of operations, aggregate true cost $\leq$ aggregate amortized cost.

We'd like to be able to reason about aggregate true costs. If someone cleverer can think up amortized costs for us, it makes it easy to do so.
FUNDAMENTAL INEQUALITY OF AMORTIZATION

Let there be a sequence of $m$ operations, applied to an initially-empty data structure, whose true costs are $c_1, c_2, \ldots, c_m$. Suppose someone invents $c_1', c_2', \ldots, c_m'$. These are called amortized costs if

$$c_1 + \cdots + c_j \leq c_1' + \cdots + c_j' \quad \text{for all} \quad j \leq m$$

Ex. sheet 6 q.6 asks you to think through why this is a sensible restriction
I’ve designed a data structure that supports push at amortized cost $O(1)$ and popmin at amortized cost $O(\log N)$, assuming the number of items never exceeds $N$.

Amortized costs make it easy for the user to reason about aggregate costs.

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

$$\text{aggregate cost} \leq m_1 O(1) + m_2 O(\log N) = O(m_1 + m_2 \log N)$$
I've designed a data structure that supports push at amortized cost $O(1)$ and popmin at amortized cost $O(\log N)$, assuming the number of items never exceeds $N$.

Amortized costs make it easy for the user to reason about aggregate costs.
For any sequence of $m_1 \times$ push and $m_2 \times$ popmin, applied to an initially empty data structure,

$$\text{aggregate cost} \leq m_1 O(1) + m_2 O(\log N) = O(m_1 + m_2 \log N)$$
SECTION 7.4
Potential functions
or, how on earth do we come up with useful amortized costs?
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:

\[
\text{accounting cost} = \text{true cost} + (\text{credit it stores}) - (\text{credit it releases})
\]

Let’s ‘pay ahead’ for the potentially-costly operations.
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: \( \text{(accounting cost)} = \text{(true cost)} + \text{(credit it stores)} - \text{(credit it releases)} \)

Let’s ‘pay ahead’ for the potentially-costly operations.

```python
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: 
  \[
  \left( \text{accounting cost} \right) = \left( \text{true cost} \right) + \left( \text{credit it stores} \right) - \left( \text{credit it releases} \right)
  \]
- Let’s ‘pay ahead’ for the potentially-costly operations

```python
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
```

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

\[
\text{aggregate true cost} \leq \text{aggregate amortized cost}
\]

- Let's 'pay ahead' for the potentially-costly operations
Let $\Omega$ 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

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

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

For an operation $S_{\text{ante}} \rightarrow S_{\text{post}}$ with true cost $c$, define the accounting cost to be

$$c' = c + \Phi(S_{\text{post}}) - \Phi(S_{\text{ante}}) = c + \Delta\Phi$$

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

**PROOF:** Consider an arbitrary sequence of operations, starting from empty:

$$S_0 \xrightarrow{c_1} S_1 \xrightarrow{c_2} S_2 \rightarrow \ldots \xrightarrow{c_m} S_m$$

aggregate accounting = $c_1' + c_2' + \ldots + c_m'$

cost

$$= -\Phi(S_0) + c_1 + \Delta\Phi - \Phi(S_1) + c_2 + \Phi(S_2) - \Phi(S_1) + c_2 + \Phi(S_2) \ldots - \Phi(S_{m-1}) + c_m + \Phi(S_m)$$

$$= 0 \quad \text{by defn. of } \Phi$$

$$= -\Phi(S_0) + c_1 + \ldots + c_m + \Phi(S_m)$$

$$\geq c_1 + \ldots + c_m \quad \text{aggregate true cost}$$
EXAMPLE: DYNAMIC ARRAY

A Python list is implemented as a dynamically-sized arrays. 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 \( km \).

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

Let's set \( \Delta \Phi = \# \text{newly added items since last doubling} \times \in \)

\[ c' = c + \Delta \Phi \]

Let's set \( 1 \in = 2k \). (This way, \( \Delta \Phi \) "pays off" for the variable amount of work involved in doubling.)

Observe that the amortized costs are slightly different (some \( 1+k \), some \( 1+2k \)), but in all cases, the amortized cost of append \( \leq 1+2k \).

In other words, the cost of append is \( O(1) \), asymptotic in \( N=\# \text{items in array} \).

\[ \exists k'>0, \forall N \geq N_0 \quad \text{am. cost of append} \leq k' \]
A Python list is implemented as a dynamically-sized arrays. 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 $O(1)$.

There are two ways that `append()` might play out:

1. If the array size has to double from $n$ to $2n$,
   
   $$c = O(n) + O(1) \quad \Phi_{\text{ante}} = 2n - n \quad \Phi_{\text{post}} = 2(n+1) - 2n \quad \Delta \Phi = 2 - n$$
   
   $$\Rightarrow c' = c + \Delta \Phi = O(n) + O(1) + 2 - n = O(1)$$

   [What we really mean here is: define $\Phi = (2n - \text{size}) \times \epsilon$, and set the exchange rate equal to the constant hidden inside $O(n)$.

2. If the array doesn't need doubling,
   
   Then, $-n \epsilon$ really does cancel out $O(n)$.

   $$c = O(1) \quad \Delta \Phi = 2 \quad \Rightarrow c' = c + \Delta \Phi = O(1) + 2 = O(1)$$

In both cases, am-cost is $O(1)$.

[Technically, this $\Phi$ isn’t a potential function. A potential function must be $\geq 0$, and $=0$ when empty, but $\Phi(\text{empty}) = -1$. We can create a proper potential function $\Phi'$ that’s like $\Phi$ except at empty. This changes some of the amortized costs, but only finitely many, so big-O results remain true, just with a possibly-larger $K$.]

Let $\Phi = 2 \times \# \text{items in array} - \text{size of array}$. 