*Work in pairs. For each task, predict the Big-O on paper first, then check with* %timeit.

**Task 1.** What is the time complexity (Big-O) of computing the Fibonacci of n?

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

**Task 2.** Which is more readable? Which runs faster? What is the time complexity of each? *(Assume arr is sorted.)*

**Example 1**

```
def ex1(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
```

**Example 2**

```
def ex2(arr, x):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] < x:
            low = mid + 1
        elif arr[mid] > x:
            high = mid - 1
        else:
            return mid
    return -1
```

**Task 3.** What does each algorithm do, which is most readable, and what is its time complexity?

```
def ex1(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                t = arr[j]
                arr[j] = arr[j+1]
                arr[j+1] = t
```

```
def ex2(arr):
    if len(arr) <= 1:
        return arr
    p = arr[len(arr)//2]
    lo = [x for x in arr if x<p]
    mid = [x for x in arr if x==p]
    hi = [x for x in arr if x>p]
    return ex2(lo)+mid+ex2(hi)
```

```
def ex3(arr):
    if len(arr) <= 1:
        return
    m = len(arr)//2
    L, R = arr[:m], arr[m:]
    ex3(L); ex3(R)
    i = j = k = 0
    while i<len(L) and j<len(R):
        if L[i] < R[j]:
            arr[k] = L[i]; i += 1
        else:
            arr[k] = R[j]; j += 1
        k += 1
    while i < len(L):
        arr[k] = L[i]
        i += 1; k += 1
    while j < len(R):
        arr[k] = R[j]
        j += 1; k += 1
```

**Task 4.** For each search (Task 2) and sort (Task 3), what input produces its best-case running time, and what input its worst-case? State the Big-O of each case.

**Task 5.** What is the worst-case auxiliary space complexity (memory used beyond the input itself, including the recursion stack) of each algorithm in Tasks 1–3?

**Task 6.** You will test membership a million times. Predict the complexity of one lookup in each, then time both on a large collection. Why does the built-in data structure change the Big-O?

| List | Set |
|---|---|
| `data = list(range(200_000))`<br>`x in data` | `data = set(range(200_000))`<br>`x in data` |

**Task 7.** Both build one string of length n. Both look like a single pass. Time them for growing n (10k, 20k, 40k) and plot — do both curves have the slope your Big-O predicted?

| Concatenate in a loop | Join |
|---|---|
| `s = ""`<br>`for w in words:`<br>`  s = s + w` | `s = "".join(words)` |

**Task 8.** Prepare for the rest of the day.