Warmup (L23)

Write a version of `sortedSearch` (binary search) that returns the index of the *first* instance of a value in the list, rather than just returning True or False. If the value isn't in the list, return -1.

Efficiency

CS114 M10

How long does it take?

- Consider these two functions:

```
def foo(x: int) -> int:
    r = 0
    for i in range(x):
        r = r + i
    return r
```

```
def foo(x: int) -> int:
    r = 1
    for i in range(1000):
        r = r * x
    return r
```

- Which is faster?

“Faster” is hard

- With our previous two functions, which is faster depends on:
  - The value of `x`
  - How long addition takes
  - How long multiplication takes
  - Your particular machine and its performance

Something more causal

- Instead of trying to predict how long things take, we want this:

```
def foo(x: int) -> int:
    r = 0
    for i in range(x):
        r = r + i
    return r
```

As `x` gets bigger, this takes longer

```
def foo(x: int) -> int:
    r = 1
    for i in range(1000):
        r = r * x
    return r
```

This takes (roughly) the same amount of time no matter what `x` is

Efficiency is a function of input

CS114 M10

How to think about efficiency

- Not “how long does this take”, …
- not even “how many steps does this take”, …
- but, “how does the input affect how many steps this takes.”
- The actual duration of any step is too ephemeral to be a useful measure
- We want “bigger number takes more time”

Back to our example

- Here's our first example

```
def foo(x: int) -> int:
    r = 0
    for i in range(x):
        r = r + i
    return r
```

- Let's try to count the steps
  - What exactly a "step" is doesn't matter (we'll see why later)

Counting steps

- Outside the loop:
  - `r = 0`
  - Making the range up to `x`
  - `return r`
- Inside the loop:
  - Adding `r` and `1`
  - Assigning the result to `r`

Counting steps

- So, the number of steps is

$$2x + 3$$

(That is, two steps in the loop times *x* times through the loop, plus three steps outside the loop)

Counting steps

$$2x + 3$$

- Let's say we have no idea how long any single step takes (because we don't)
- What can we still say about the runtime?
  - If *x* is very big, then the 3 is negligible: 2*x* "dominates" 3
  - Since I don't know how long any of the steps take, the 2 is just "some number", and the 3 is just "some other number"

Abstracting our count

$$k_1 x + k_2$$

- Since the actual numbers don't matter, we can rewrite them as *k* (for "*k*onstant", because mathematicians can't spell)
- We don't actually care what the *k*s are!

Abstracting our count

$$k_1 x$$

- Since the *x* term dominates the k_2 term, we ignore the k_2
- Put differently, if I care how the amount of time *grows* in relation to the input, I don't care about a factor that's going to be insignificant when the input grows

Abstracting our count

$$1x$$

- Since k_1 is just some unknown number, this is describing the same relationship if it's 1

Abstracting our count

$$x$$

- Since k_1 is just some unknown number, this is describing the same relationship if it's 1
- And of course, if it's 1*x*, it's just *x*

Big-O notation

- The abstractions we've just made are the standard way of thinking about running time
- We call this abstracted relationship between the input and duration the *order of approximation* of the running time
- To show when we're approximating in this way, we use a big O: this was

  O(*x*) where *x* is the value of `x`

How to Big-O

1. Choose the input that will affect the execution time (value of a variable, length of a list, anything measurable)
2. Count steps and those steps' multipliers
3. Turn that into a mathematical expression
4. Remove dominated terms
5. Replace constants with 1s

How to Big-O

… and then, use "*n*" as a variable name instead of "*x*" or anything else, or people will get confused.

So, that one was O(*n*), where *n* is the value of `x`

In-lecture quiz (L23)

- https://student.cs.uwaterloo.ca/~cs114/quiz/
- Q1: Based on what you know about **`in`**, what do you think is the efficiency of `(x in lst)`, where *n* is the length of the list `lst`?

A. $O(1)$

B. $O(n)$

C. $O(n^2)$

D. $O(n+1)$

E. $O(2^n)$

In-lecture quiz (L23)

- https://student.cs.uwaterloo.ca/~cs114/quiz/
- Q2: What is the big-O efficiency of the following function, where *n* is the length of the list lst?

```
def foo(lst: list[float]) -> None:
    for idx in range(len(lst)):
        lst[idx] = 1
        for el in lst:
            lst[idx] *= el
```

A. $O(1)$
B. $O(n)$
C. $O(n^2)$
D. $O(n+1)$
E. $O(2^n)$

What counts as a step?

CS114 M10

What counts as a step?

- To count steps, we need to know what's considered a single step
- The short answer: "anything that doesn't hide repetition inside" (i.e., that doesn't hide a loop or recursion inside)
- Remember, we're replacing all constants with 1, so we don't care how complex a step is as long as it doesn't grow *more* complex with our input

What counts as a step

One step:

- All arithmetic on numbers
- Variable assignment
- `len(`list or array`)`
- Appending to a list
- `list[idx]` or `array[idx]` or `str[idx]`

n steps:

- Arithmetic on arrays
- Concatenating lists
- Slicing lists
- **`in`**
- String splitting, joining, etc.

Let’s measure efficiency

CS114 M10

```
def factorize(n: int) -> list[int]:
    assert n > 0, "Only positive
                   integers have
                   factors"

    r = []
    f = 1
    while f <= n:
        if n%f == 0:
            r.append(n)
        f = f + 1
```

```
def factorize(n: int) -> list[int]:
    assert n > 0, "Only positive
                   integers have
                   factors"

    r = []
    f =
    while f <= n:
        if n%f == 0:
            r.append(n)
        f = f + 1
```

O(*n*) where *n* is the value of `n`

```
def runningAverage(l: list[float]) -> float:
    sum = 0.0
    for idx in range(len(l)):
        sum = sum + l[idx]
        l[idx] = sum / (idx+1)
    return sum / len(l)
```

```
def runningAverage(l: list[float]) -> float:
    sum = 0.0
    for idx in range(len(l)):
        sum = sum + l[idx]
        l[idx] = sum / (idx+1)
    return sum / len(l)
```

O(*n*) where *n* is the length of the list `l`

```
def weirdProduct(
    lst: list[float]
) -> list[float]:
    r = []
    for idx in range(len(lst)):
        p = 1
        for pIdx in range(0, idx+1):
            p *= lst[pIdx]
        r.append(p)
    return r
```

```
def weirdProduct(
    lst: list[float]
) -> list[float]:
    r = []
    for idx in range(len(lst)):
        p = 1
        for pIdx in range(0, idx+1):
            p *= lst[pIdx]
        r.append(p)
    return r
```

This is a weird one, so let’s count carefully…

$$2n \cdot ? + 3n + 3$$

- The *?* is number of iterations of the inner loop
- But how many iterations is it?
- The first time around the loop, it's just 1, the second time it's 2, and the *x*th time it's *x*
- Since the number of iterations grows with *n*, it acts like another *n*

$$2n^2 + 3n + 3$$

$$k_1 n^2 + k_2 n + k_3$$

$$n^2 + n + 1$$

$$\boxed{O(n^2)}$$

```
def weirderProduct(
    lst: list[float]
) -> list[float]:
    r = []
    for idx in range(3, len(lst)):
        p = 1
        for pIdx in range(idx-3, idx):
            p *= lst[pIdx]
        r.append(p)
    return r
```

$$2(n-3) \cdot ? + 3(n-3) + 3$$

- The *?* is number of iterations of the inner loop
- But how many iterations is it now?
- It's always 3; it's not related to the length of the list!

$$6(n-3)+3(n-3)+3$$
$$6n-18+3n-9+3$$
$$9n-24$$
$$k_1 n + k_2$$
$$n+1$$
$$\boxed{O(n)}$$

Intuition

- Loops are a term of *n* **if** the number of loops is based on *n*
- A loop within a loop multiplies the inner factor by the outer factor
- Really, anything within a loop multiplies that anything by the loop

```
def contains(
    haystack: typing.Sequence,
    needle: typing.Any
) -> bool:
    for val in haystack:
        if val == needle:
            return True
    return False
```

Early return

- How many steps depends on whether it returned early or not!
- We can measure this in a number of different ways
  - *Expected* time, *average* time, *best-case* time, *worst-case* time…
- In practice (and if it's not said otherwise), we want the *worst-case* number of steps

```
def contains(
    haystack: typing.Sequence,
    needle: typing.Any
) -> bool:
    for val in haystack:
        if val == needle:
            return True
    return False
```

Assume early return doesn't happen, so O(*n*)

Recursion and efficiency

CS114 M10

Measuring recursion

- Recursion can be thought of as a cuboid:
  - Depth of the cuboid: how *deep* the recursion goes (how many calls down the chain you can get)
  - Width of the cuboid: how *wide* the recursion goes (e.g., if `fib` calls `fib` twice, then the total number of calls at every layer of recursion is 2^n)
  - Length of the cuboid: the order of the non-recursive part of the recursive function

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

- Depth: *n*
- Width: 1
- Length: 1
- Efficiency: O(*n*)

```
def fib(n: int) -> int:
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

- Depth: *n*
- Width: 2^n
- Length: 1
- Efficiency: 2^n (it dominates the *n*!)

```
def sortedSearch(lst: list, needle: typing.Any) -> bool:
    if len(lst) == 0:
        return False
    midpoint = len(lst) // 2
    if lst[midpoint] == needle:
        return True
    elif lst[midpoint] < needle:
        return sortedSearch(lst[midpoint+1:], needle)
    else: # lst[midpoint] > needle
        return sortedSearch(lst[:midpoint], needle)
```

- Depth: $\log n$ (the log base doesn't matter, as it's a *k*)
- Width: 1
- Length: *n* because we sliced ☹ (would've been 1 if we didn't slice)
- Efficiency: $n \log n$ (ew, slice) (would've been $\log n$ if we didn't slice)