A Brief Intro to Time Complexity

Announcements

- **EX05: Dictionary Utility Functions** due Monday (March 9) at 11:59pm
- Quiz 02 in one week (March 13)!
    - Practice problems will be posted to the site shortly (by tomorrow at the latest)
    - Hybrid Review Session next Wednesday (date and time TBA soon!)
    - If you take your quizzes with the UCO, reminder to check that you have scheduled your quiz
    - Have a university-approved absence? Email me to schedule an alternate time to take it!

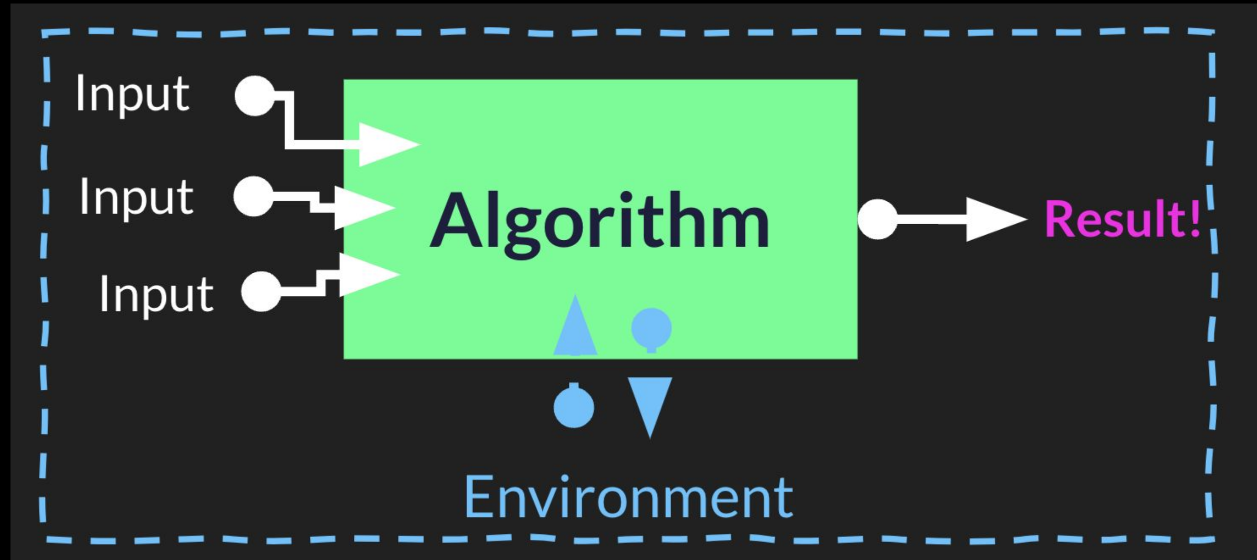
Review: Algorithms

Input is data given to an algorithm

An **algorithm** is a series of steps

An algorithm **returns** some **result**

An algorithm *may* be influenced by its **environment** and it *may* produce side-effects which influence its environment.



Warm-up:

With a partner, discuss which of the following factors you think are *most important* to consider when selecting or implementing an algorithm:

- Simplicity
- Ease of implementation
- Speed (how long does the code take to run?)
- Efficient use of memory
- Precision in answer

Is one factor *always* more important than the others, or would it vary by algorithm?

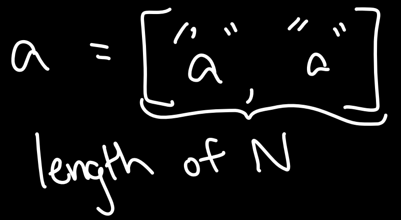
With a neighbor, try diagramming:

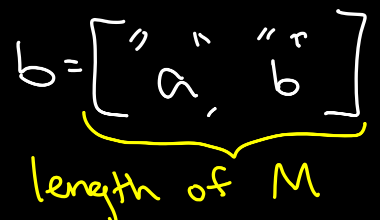
```python
def intersection(a: list[str], b: list[str]) -> list[str]:
    result: list[str] = []

    idx_a: int = 0
    while idx_a < len(a):
        idx_b: int = 0
        found: bool = False
        while not found and idx_b < len(b):
            if a[idx_a] == b[idx_b]:
                found = True
                result.append(a[idx_a])
            idx_b += 1
        idx_a += 1

    return result


foo: list[str] = ["a", "b"]
bar: list[str] = ["c", "b"]
print(intersection(foo, bar))
```

will loop always len(a) times (N)

at best, will loop 1 time
at worst, will loop M times

How many times will line 9 be reached?
At best: N times
At worst: N * M times





… and after diagramming:

Assume our unit of "operation" is the number of times the block of lines #9-12 are evaluated.

Q1. Can different values of a and b lead to a difference in the number of operations required for the intersection function evaluation to complete? Yes!

Q2. If so, provide example item values for a and b which require the fewest operations to complete? Then try for the maximal operations to complete?

Q3. Assuming the item values of a and b are random and unpredictable, about how many operations does this function take to complete?

```python
def intersection(a: list[str], b: list[str]) -> list[str]:
    result: list[str] = []

    idx_a: int = 0
    while idx_a < len(a):
        idx_b: int = 0
        found: bool = False
        while not found and idx_b < len(b):
            if a[idx_a] == b[idx_b]:
                found = True
                result.append(a[idx_a])
            idx_b += 1
        idx_a += 1

    return result


foo: list[str] = ["a", "b"]
bar: list[str] = ["c", "b"]
print(intersection(foo, bar))
```

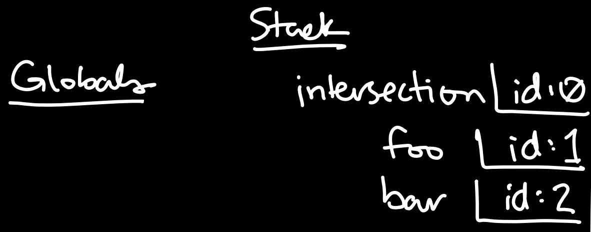
Stack

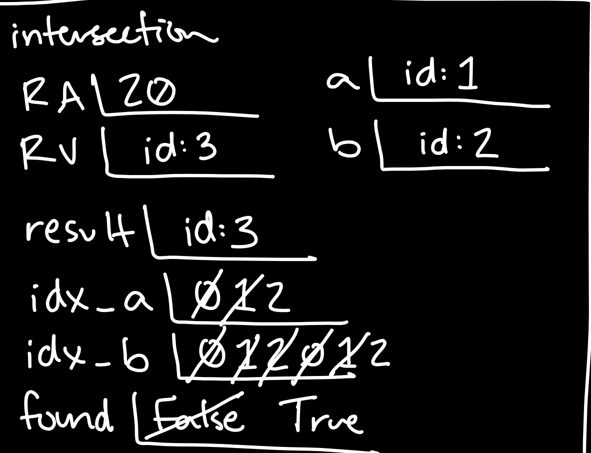
Globals

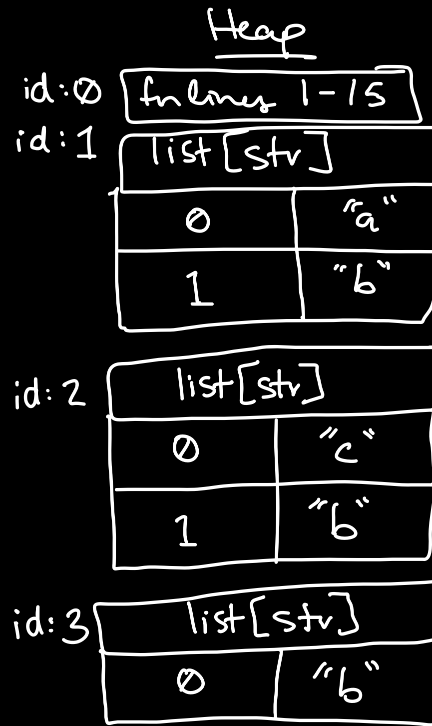
Heap

Output

```python
def intersection(a: list[str], b: list[str]) -> list[str]:
    result: list[str] = []

    idx_a: int = 0
    while idx_a < len(a):
        idx_b: int = 0
        found: bool = False
        while not found and idx_b < len(b):
            if a[idx_a] == b[idx_b]:
                found = True
                result.append(a[idx_a])
            idx_b += 1
        idx_a += 1

    return result


foo: list[str] = ["a", "b"]
bar: list[str] = ["c", "b"]
print(intersection(foo, bar))
```







Output

["b"]

As the lengths of `a` and `b` grow, the number of operations grows *quadratically*

```python
def intersection(a: list[str], b: list[str]) -> list[str]:
    result: list[str] = []

    idx_a: int = 0
    while idx_a < len(a):
        idx_b: int = 0
        found: bool = False
        while not found and idx_b < len(b):
            if a[idx_a] == b[idx_b]:
                found = True
                result.append(a[idx_a])
            idx_b += 1
        idx_a += 1

    return result


foo: list[str] = ["a", "b"]
bar: list[str] = ["c", "b"]
print(intersection(foo, bar))
```

- Outer while loop iterates through each element of `a`
  - *If there are N elements, we'll iterate N times*
- And within each iteration of the outer while loop…
- The inner while loop iterates through elements of `b` until either:
  - We find a value that == the current element in `a` OR,
  - We have "visited" (accessed) every element in `b`
    - *If there are M elements in* `b`*, we'll iterate up to M times*

Assuming `a` and `b` both have 3 elements…

1. Example of values of `a` and `b` that will cause the ***fewest*** operations to occur?
   `intersection(a=["a", "a", "a"], b=["a", "b", "c"])`
2. Example of values of `a` and `b` that will cause the ***most*** operations to occur?
   `intersection(a=["a", "b", "c"], b=["d", "e", "f"])`

If list `a` has N elements and list `b` has M elements, the "worst case scenario" is that this code will cause N*M operations to occur.

Comparing lists and sets

```python
def intersection(a: list[str], b: list[str]) -> list[str]:
    result: list[str] = []

    idx_a: int = 0
    while idx_a < len(a):
        if a[idx_a] in b:
            result.append(a[idx_a])
        idx_a += 1

    return result
```

```python
def intersection(a: list[str], b: set[str]) -> set[str]:
    result: set[str] = set()

    idx_a: int = 0
    while idx_a < len(a):
        if a[idx_a] in b:
            result.add(a[idx_a])
        idx_a += 1

    return result
```

Suppose `a` and `b` each had 1,000,000 elements. The worst case difference here is approximately 1,000,000 operations, versus 1,000,000**2 or 1,000,000,000,000 operations.

If your device can perform 100,000,000 operations per second, then...

A call to `a` will complete in 2.78 hours and `b` will complete in 1/100th of a second.

Recall: comparing lists and sets

```python
def intersection(a: list[str], b: list[str]) -> list[str]:
    result: list[str] = []

    idx_a: int = 0
    while idx_a < len(a):
        if a[idx_a] in b:
            result.append(a[idx_a])
        idx_a += 1

    return result
```

```python
def intersection(a: list[str], b: set[str]) -> set[str]:
    result: set[str] = set()

    idx_a: int = 0
    while idx_a < len(a):
        if a[idx_a] in b:
            result.add(a[idx_a])
        idx_a += 1

    return result
```

Suppose `a` and `b` each had 1,000,000 elements. The worst case difference here is approximately 1,000,000 operations, versus 1,000,000**2 or 1 trillion (1,000,000,000,000) operations.

If your device can perform 100,000,000 operations per second, then...

A call to `a` will complete in 2.78 hours and `b` will complete in 1/100th of a second.

Running time: how long does an algorithm take to run?

- Empirical analysis: write the code and test how long it takes to run!
    - Weaknesses:
        - You have to write the code for the whole algorithm and run it to see how long it will take
        - Different computers with different specs will have different runtimes
- Rather than using empirical analysis, computer scientists commonly consider the **number of operations (steps)** an algorithm requires
    - 1 operation == 1 step

Runtime analysis: Best, average, and worst case

Best case (lower bound):

- Minimum number of operations (running time) required for the algorithm to execute

Average case:

- Average running time among several different inputs

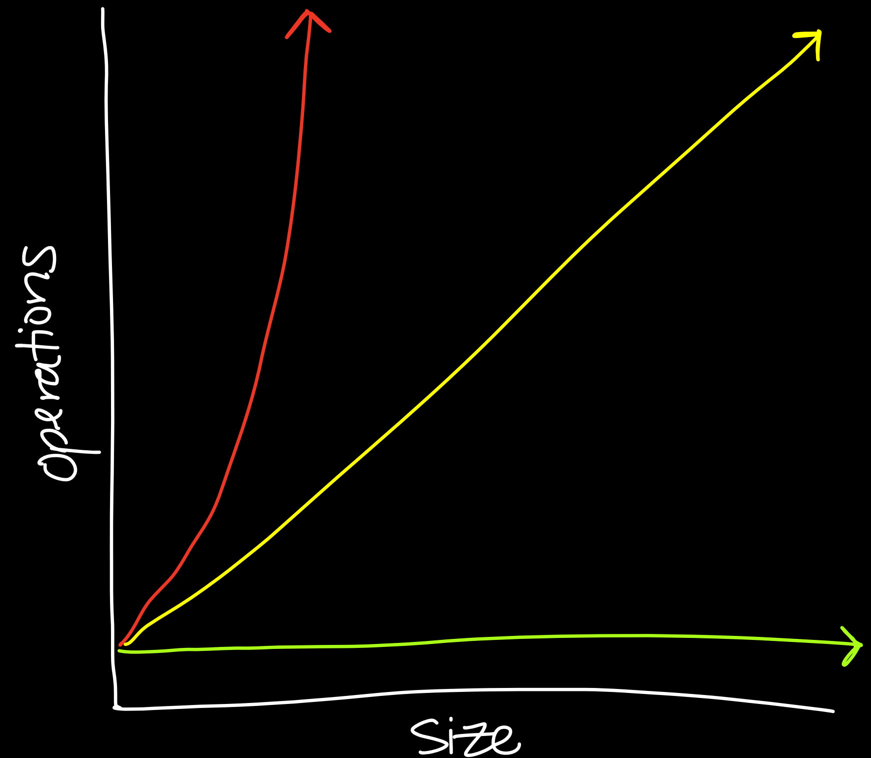
Worst case (upper bound) ✨:

- The maximum running time given an input
  - How does the number of operations grow as an input grows?
- Important to understand how our algorithm will perform in the *worst* case
  - Prepare for the worst case. If an input ends up requiring fewer operations, great!!

Runtime analysis: order of growth

We want our algorithms to scale well!

As the size (e.g. number of elements) of our inputs grow, we want to be careful about how the number of operations grows.



Runtime analysis: order of growth