Internet and Big Data

Review questions

What were the 2 searches we learned about last week?

A) Recursive and iterative
B) Linear and binary
C) Base case and recursive step
D) Use `for` and `range`

What is the requirement to perform a binary search on a list?

A) Data has no negative values
B) Data has no repeating values
C) Data has the value we are looking for
D) Data is sorted

Review
Binary Search

How would you do a binary search and return the **index** of the search term, or **-1** if not found?

```python
# Do binary search
def binary_search(data, item):
    # Track active search space
    low = 0
    # high = len(data) - 1   # Index of last element

    while low <= high:
        # Find the middle of the active search space
        middle_idx = (low + high) // 2
        middle_item = data[middle_idx]

        # If it's the middle, return true!
        if middle_item == item:
            return True
        else:
            # Otherwise, do we look to the left,
            # or do we look to the right of the middle?
            if item < middle_item:
                # ... look left of middle.
                high = middle_idx - 1
            else:
                # ... look right of middle.
                low = middle_idx + 1
    return False
```

http://interactivepython.org/courselib/static/pythonds/SortSearch/TheBinarySearch.html

A real-life example

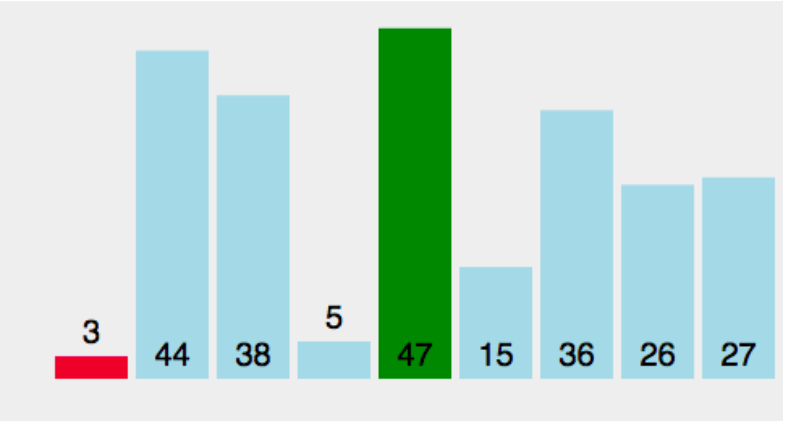
Let's say you want to sort Pokémon cards. You could sort by HP. How might you do it?

There are many sorting methods! And visualisations

on the internet, to help you understand the algorithm.

https://visualgo.net/bn/sorting

(choose "8. Selection Sort")



Pseudocode is also available in the link below (pp. 130-131)

http://www.cs.sfu.ca/CourseCentral/120/ggbaker/guide/parts/guide06

Swapping pattern

Let's say we want to swap the values at two different spots in a list. How would you do it?

We can use a temporary variable to swap a and b.

1. temp ← a
2. a ← b
3. b ← temp

a

temp

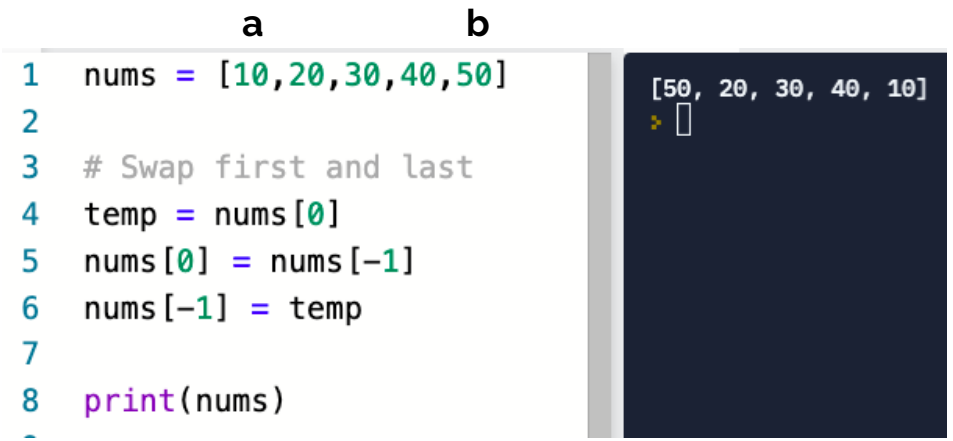
b

Swapping pattern

Let's say we want to swap the values at two different spots in a list. How would you do it?

We can use a temporary variable to swap a and b.

1. temp ← a
2. a ← b
3. b ← temp



Selection Sort

- For every element in the list:
  - Find the smallest element in the rest of the list
  - Swap the current element with that smallest element

Selection Sort

```python
# Input:  An unsorted list of numbers
# Output: Returns a sorted list of numbers
#         (Input list is unchanged)
def selection_sort(data_original):
    # Make a copy so we don't change the original
    data = data_original[:]

    # For each spot in the list, find the next smallest number
    # in the remaining sublist of our numbers
    for i in range(len(data)):
        # Start by assuming smallest element is first in sublist
        min_number = data[i]
        min_idx = i

        # Look through rest of sub-list for smallest element
        for j in range(i + 1, len(data)):
            if data[j] < min_number:
                min_number = data[j]
                min_idx = j

        # Swap the current element with the next smallest element
        temp = data[i]
        data[i] = data[min_idx]
        data[min_idx] = temp

    return data
```

```python
test = [5,3,6,2,1]
print(selection_sort(test))
print(test)
```

```
[1, 2, 3, 5, 6]
[5, 3, 6, 2, 1]
```

Selection Sort

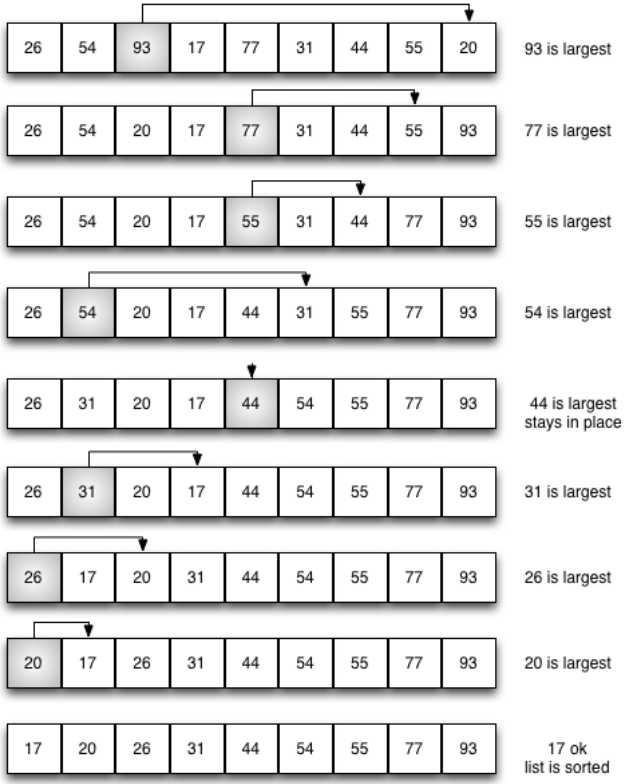
Will it work with words?

A) Yes
B) No

Could we sort from the back to front?

How about largest (first) to smallest (last)?

Could we sort (rearrange) letters in a string?



http://interactivepython.org/courselib/static/pythonds/SortSearch/TheSelectionSort.html

Timing Programs

What time is it?

You may have used the **time** module to **sleep**() to add pauses in your chatbot. With the time module, you can also find out what time it is, given in "seconds since the epoch".

January 1, 1970

main.py

```
1  # Calculating elapsed time
2  # Author: Angelica Lim
3  # Date: March 21, 2018
4
5  import time
6
7  # Get the time now
8  t0 = time.time()
9
10 # Sleep for one second
11 time.sleep(1)
12
13 # Get the time now
14 t1 = time.time()
15
16 print(t0)  # Time at t0
17 print(t1)  # Time at t1
18 print(t1-t0)  # Elapsed time (s)
```

```
1521646354.8800712
1521646355.8801394
1.000068187713623
```

time.time()

Use **time** to measure how long your program took. We will be using this in this unit!

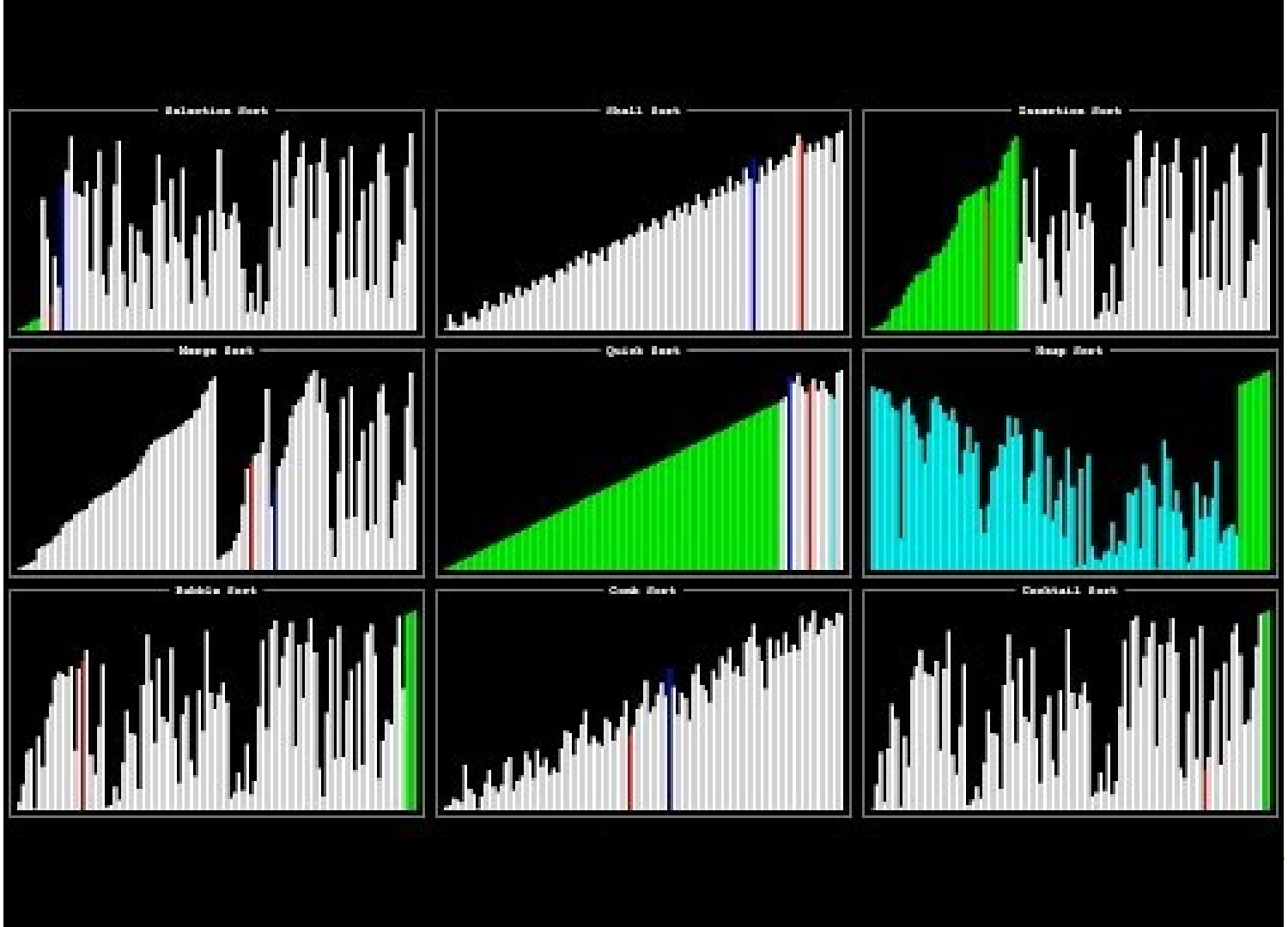
https://en.wikipedia.org/wiki/Unix_time#Encoding_time_as_a_number
https://docs.python.org/3.0/library/time.html

Timing our algorithms

Let's generate a big list of numbers and time our different algorithms.

```
32  import random
33  import time
34  num_list = random.sample(range(1, 1000000), 10000)
35
36  t0 = time.time()
37  my_sorted = selection_sort(num_list)
38  t1 = time.time()
39  num_list.sort()
40  t2 = time.time()
41
42  my_time = t1 - t0
43  py_time = t2 - t1
44  print (f"Built-in sort:  {py_time:.20f}")
45  print (f"Selection sort: {my_time:.20f}")
```

```
Built-in sort:  0.00000000000000000000
Selection sort: 1.18919777870178222656
```


Selection Sort
Shell Sort
Insertion Sort
Merge Sort
Quick Sort
Heap Sort
Bubble Sort
Comb Sort
Cocktail Sort

Goodness of an algorithm?

- **#1 Criteria: Correct** (i.e., it works. Though sometimes we need to compromise and accept an approximate solution)
- **#2 Desirable qualities:**
  - Clear to read code and to debug
  - Code easy to understand
  - Good user interface
  - Concise code
  - Modular (levels of abstraction, use of functions), structured
  - Robust (does not crash)
  - Easy to maintain and revise

2 Ways to Examine the "Goodness" of an algorithm

Time complexity: is time used efficiently?

- The algorithm executes efficiently with a realistic response time
- The more time it takes to produce the same result, the higher the complexity

Space complexity: is space used efficiently?

- The algorithm uses an optimal (or at least an acceptable amount) of memory
- The more space it needs to produce the same result, the higher the complexity

Efficiency is the essential quality to consider for **large** size problems!

Complexity

The amount of resources (in time, space) required to run an algorithm.

In this course, we'll focus on time

Measuring time complexity

- Generally speaking, we measure the **time** it takes for the algorithm to solve particular problem (with a considerable size)
- However, the **time depends on a lot of things**, e.g., CPU speed, RAM size. So while execution time gives us a good idea, it's not the most accurate measurement
- A better way is to **count the number of operations** that get carried out (i.e., executed), which is **independent of hardware** variations – we call this notion of time complexity the "**order of an algorithm**"

Order of an algorithm

- Gives a notion of the **Time complexity** of the algorithm

- This is a **theory** that is most relevant for problems and algorithms involving **large numbers of data** (large size of problems)

Like searching and sorting a million songs!

Order of an algorithm

- The **Order** gives an **“approximate”** measure of an algorithm in terms of number of **“critical operations”** that are executed.

- **“approximate”** is in fact very **precisely defined mathematically.**

Critical operations can be:

- **additions**
- **comparisons** (if statements)
- **transfer operations** (assignments) …

Order and Problem Input Size

- Since order is approximately the number of operations that get carried out, **the larger the problem input size is, the more operations are required**

- We can express **order** (i.e., time complexity) as a function of the problem **input size**, which can be:
  - dimensions of lists,
  - the number of values to be added,
  - the number of values among which we search
  - the number of values to be sorted…

Order of an algorithm

- An algorithm is **rated** in terms of some **reference function**. It is said to **be in the order (big-O) of** some reference function:
  - **O(n)**, **O(n^2)**, **O(log n)**, etc.

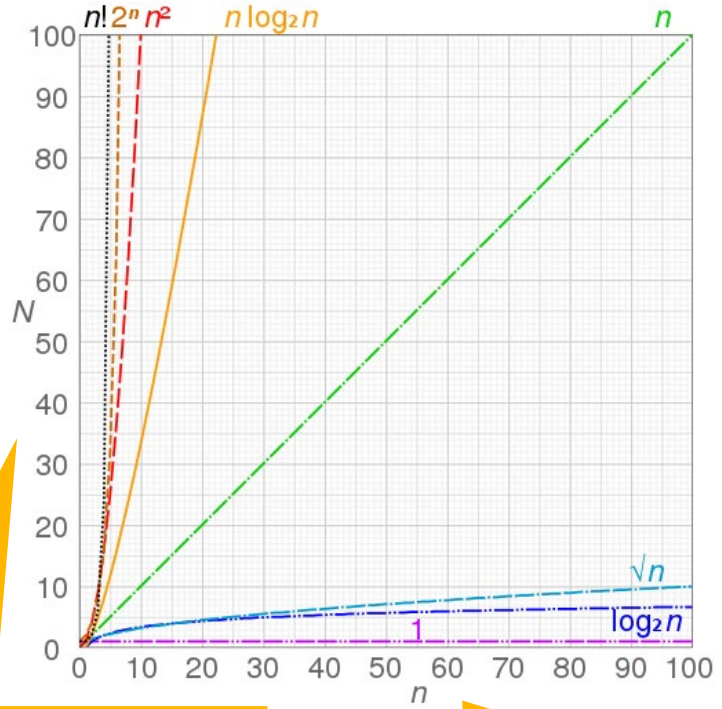
We say "Big O of n" or "Order n"

- Intuitively, that means that **for sufficiently large n**, the time that the algorithm will take (to execute the critical operations) will be **proportional** to **n**, **n^2**, **log n**, etc, where **n** is the **the problem input size**.

E.g. we have a list with n=1,000,000 elements that we want to sort

Standard reference functions

Category	Reference Function
Constant	1
Logarithmic	$\log_2(n)$
Linear	n
nlogn	$n\log_2(n)$
Quadratic	n^2
Cubic	n^3
Exponential	a^n, a>1



Number of operations

Number of elements

https://en.wikipedia.org/wiki/Big_O_notation

Order and Problem Input Size

Next we'll learn how to **analyze** our search and sort algorithms in terms of their order as the number of items to process gets large.

Let's review some concepts

What is the name of the of **sorting algorithm** we learned in this class?

Do we need to use comparison operators when sorting?