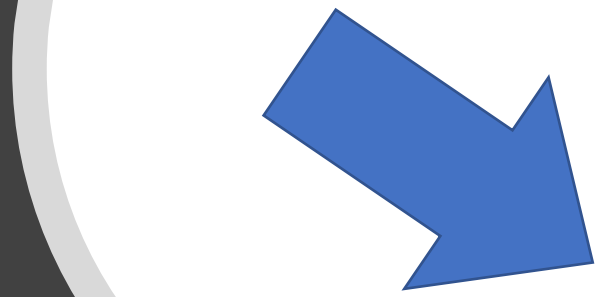
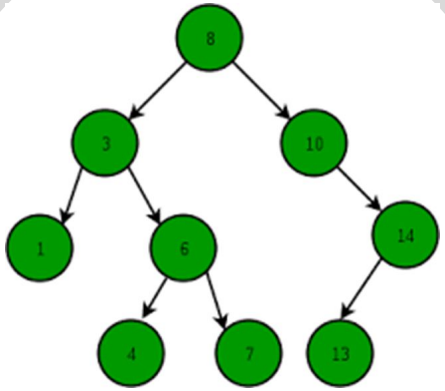
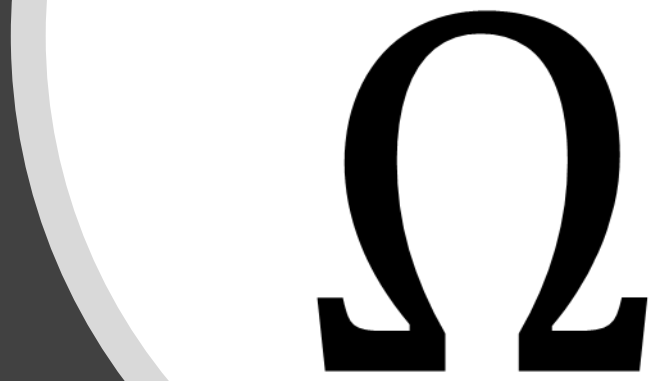
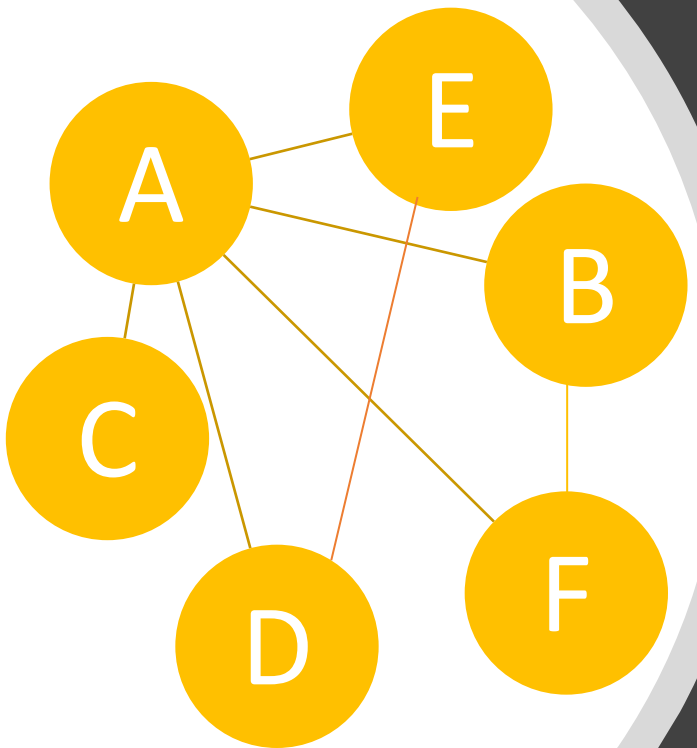








An overview of critical

Data Structures and Algorithms

The Big O

Time complexity

space complexity

"An algorithm* is characterized by its running time (run-time), whether in terms of space or time."

Big O Notation

Big Ω

- Best case scenario.
- How fast will the algo run under best circumstances.

Big O

- Worst Case Scenario.
- This is our biggest concern, an algorithm sometimes is only as strong as its weakest link .

Big θ

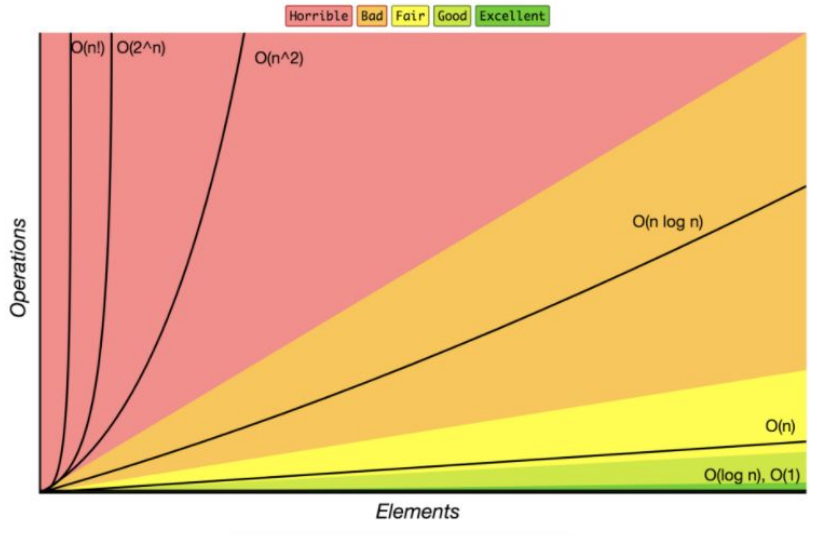
- This notation is used when Big Omega and Big O are the same.
- Run time is the same in both cases. (sometimes we use this causally to mean what the average is.)

Evaluate Your Run Time

- **O(1)** - Constant Time → Always takes the same amount of time (or space) to execute regardless of input size.
  eg. Get third value from list, value = y[3]

- **O(N)** - Linear Time → Time or (space) complexity grows in direct proportion to the input size.
  Larger input = more time/space.
  eg.for item in list:
  print(item)

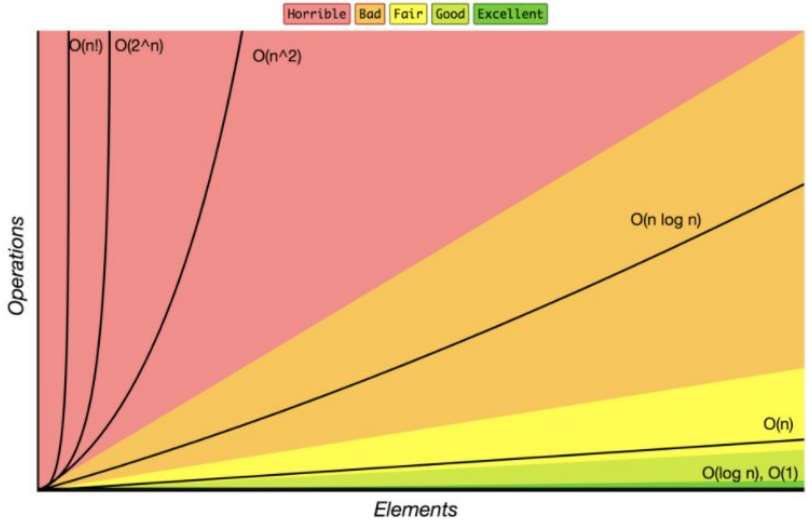
- **O(logN)** - Logarithmic Time → Time or (space) complexity grows in direct proportion to logarithm of the input size.
  - As input increases multiplicatively, time/space increases linearly.
    eg. binary search .



Evaluate Your Run Time

- **O(N²)** - Quadratic Time → Time or (space) complexity grows in direct proportion to the square of the input size.
  eg. double for loop , for item in list:
  for letter in item:
  do something …..

- **O(2ⁿ)** - Exponential Time → Growth doubles with each added element in the input.



Simple search: go through every single name in the phone book to find your person.
(not a big deal if there are only 10 names, but let's say there are thousands?!?!)
As the phonebook (dataset) increases in size (grows) the maximum time to search increases linearly.
Worst case scenario O(n) : n is number of elements.

Big Ω

Big O

Binary Search

- We are looking for the last name "Pfeffer"
- We open the phonebook to M (roughly the middle of the phone book)
- We can totally disregard all the names from A-M – we split our search into half the time!
- Can continue to divide in half

Binary search, searches sorted data and finds the midpoint

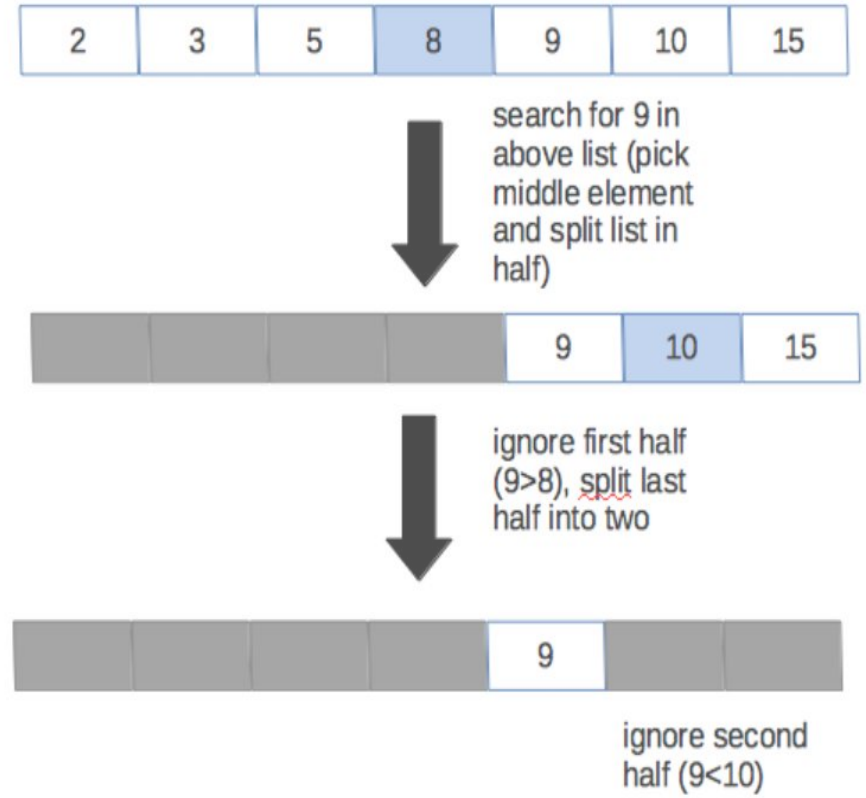
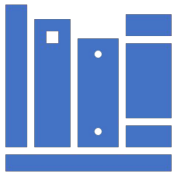


Diagram of binary search with a list of numbers

Sorting

- Selection Sort
- Quick Sort
- Merge Sort



Selection Sort

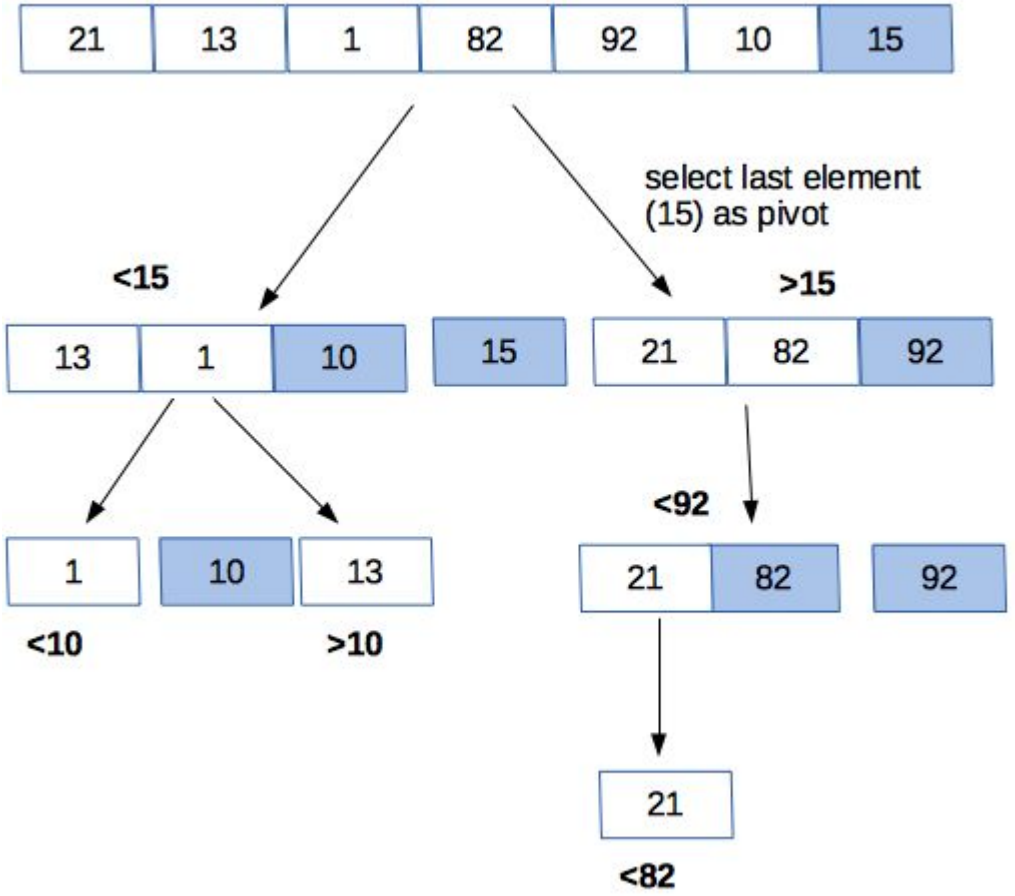
- Brute force algo (like simple search)
- Go thru every element in a list and append each element to a new list in order

1. Search through the list to find the largest number
2. Add that number to a new list
3. Go to the original list, search through it again to find the next largest number
4. Add that number to the new list and so on...

What's the big O?

Quicksort

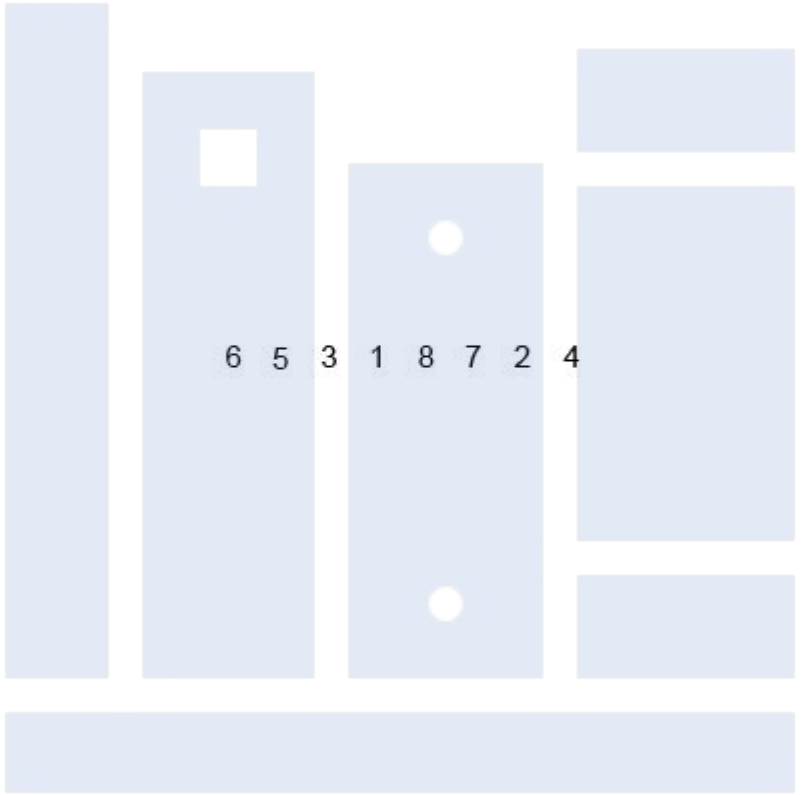
1. Pick an element from your list, known as the pivot. The selection of a pivot is important in determining how quickly a quicksort algorithm will run. For now, we can select the last element each time as the pivot.
2. Partition the list so that all numbers smaller than the pivot are to its left and all numbers greater than the pivot are to its right.
3. For each 'half' of the list, you can treat it as a new list with a new pivot and rearrange each half until it is sorted



Mergesort

1. Break down list into individual elements
2. Ordered pairs are created from element (smaller numbers to the left)
3. No order pairs are ordered into groups of 4, and so on until final merged and the list is sorted

(check out the cool animation)
(Animated overview of mergesort algorithm (credit: By Swfung8 — Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=14961648))



Recursion

A function calling itself

why use recursion?

Complex tasks can be broken down into simpler problems.

Code using recursion is usually shorter and more elegant.

Sequence generation is cleaner with recursion than with iteration.

```
def factorial(n):
    if n < 1:          #base case
        return 1
    else:               #recursive case
        return n * factorial(n-1)
```