**Readings**

1. **Balci, O., et al. (n.d.). Lesson IV: Sorting Algorithms in Virginia Tech’s *Algorithms*.**

Retrieved from <http://courses.cs.vt.edu/~csonline/Algorithms/Lessons/index.html>

There are many different algorithms that can be used to sort a list, each with advantages and disadvantages. How do we choose the best algorithm for a given task? This reading prepares students to answer this question for **simple sort, insertion sort, and others**. Make sure you **understand the steps for simple sort and insertion sort by reading sections A "Basic Operations" through E "The Insertion Sort"**. Use the arrows in the top right to navigate through all the sections**. Optional: finish the sections in lesson IV and then read lesson V on Algorithm Analysis.**

* Algorithms – set of instructions to solve a problem
  + Well ordered
    - Need for instruction of algo
  + Unambiguous operations
    - No need to simplify operation in algo, need to be clear
    - Need to be written in computer primitives to make it unambiguous for execution
  + Effectively computable operations
    - Don’t put division by zero or imaginary numbers
  + Produce a result
    - Need to have result to check if solution is correct
  + Halt in a finite amount of time
    - Should complete execution and not go into infinite loop

1. **Muhammad, R. (n.d.). Divide-and-Conquer Algorithms.**

Retrieved from <http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/divide.htm>

Just read the very short introductory section for a definition of **divide-and-conquer algorithms**, which we will discuss in class. Optional: read through the search examples that follow, written in pseudo-code.

1. **101 Computing. (January 21, 2018). Big O notation.**

Retrieved from <http://www.101computing.net/big-o-notation/>

This resource describes the most **common way to measure the performance of an algorithm**. Pay attention to how to the categories and what Big O notation tells you about an algorithms' performance. Click through to **see the examples of O(1), O(N), O(N^2), O(2^N), and O(log(N))**, noting how the loop structure relates to the algorithm's efficiency. As you read, think about the efficiency of the sorting algorithms introduced in the first reading. Note that we'll use "efficiency" synonymously with "peformance."

1. **Danish. (2014). Basic sorting algorithms implemented in python.**

Retrieved from <http://danishmujeeb.com/blog/2014/01/basic-sorting-algorithms-implemented-in-python/>

This reading provides some **code in Python for a variety of common sorting algorithms**. Although you need not know about all of these different sorting algorithms, read over the code to get a sense of how to implement the sorting algorithms in Python. Note how the loop structure relates to the algorithm's efficiency.

1. **(Optional). John, C. T. (Nov 28, 2016). What's the fastest way to alphabetize your bookshelf? *TED-Ed*.**

Retrieved from <https://www.youtube.com/watch?v=WaNLJf8xzC4>

This resource introduces a few different sorting algorithms at an accessible level by providing visualizations motivated by a practical example. **Bubble sort and quick sort will not be discussed in today's class.**

**Study Guide**

1. Comparison operation – determine which item come first in the list to arrange it in order
2. **Swap** operation – swap small and large items (one way of shorting)
   1. A = Temp
   2. B = A
   3. Temp = B
3. Describe the steps of **simple sort** and **insertion sort** in your own words.
   1. **Simple/ selection sort** 
      1. Select smallest in **group1 (unsorted)** and move to **group2 (sorted)** until all group1 is empty
      2. Need to **compare cards to select the smallest number** (eg: smaller of two numbers, then compare with next number, until no numbers left)
   2. **Insertion/bubble sort**
      1. Give list, divide sorted and unsorted section
      2. Compare **first unsorted with numbers of sorted**. If smaller than sorted, then **swap** them (**insert in position**), keep comparing with next one until it is larger than the unsorted number (time consuming)
      3. Eg: 9|467 -> 4|967 -> 49|67 -> 46|97 -> 469|7 -> 467|9 -> 4679|
4. **Practice** carrying out simple sort and insertion sort by hand on example lists. Demonstrate each step clearly.
5. Explain the difference between **constant, linear, logarithmic, and quadratic algorithms**. Give an example algorithm of each type. Explain how the big O notation relates to the *efficiency* of an algorithm.
6. Define and explain how a **divide-and-conquer algorithm works**.
   1. Divide problem into smaller subproblems, solutions easier to find then combine solution
   2. **Binary search** – **middle element of sorted array** to search for specific element (number), move either upper / lower (**achieve in logarithmic time**)
      1. Eg: finding 22 out of 100 numbers
      2. 100-> 50-> 25 -> 12->18->21 -> 23 -> 22 (instead of guessing so many times)
      3. Find q in array. Compare q with [n/2] element
   3. Sequential search -

Challenge questions (it will be helpful to think about these before class)

1. How would you express the **time efficiency of simple sort and insertion sort in big O notation**? Why?
   1. Big O notation
   2. O(1) - execute same time regardless of dataset
   3. O(N) – performance linearly proportionate to size of data, eg: linear search (go through every single element and check up till last , return if match )
   4. O(N^2) – performance directly proportional to square of size of dataset, eg: nested loops
   5. O(2^N) – execution time doubles for each addition, eg: recursion/ Backtracking, shitty maze problem tom and jerry
   6. O(logN) – Execution time Decrease if dataset increase, eg: binary search
   7. Simple sort-
   8. Insertion sort –
      1. Best case (everything is already in order, only swap once, O(N)
      2. Worse / average case, O(N^2) need to check each unsorted value with all sorted value at least once
2. How do **simple sort and insertion sort compare in terms of space efficiency**?