CMPT 120

Lecture 35 – Internet and Big Data

Algorithm – Complexity Analysis of
Searching and Sorting Algorithms

Admin Blurb

Announcements

- Office hours for Week 14 - Monday April 15 to Friday April 19 - in the usual locations:
    - Monday April 15 - Anne - 2pm to 3:30pm
    - Tuesday April 16 - Arsh - 11:30am to 1:30pm
    - Tuesday April 16 - Rohan - 1:30am to 3:30pm
    - Wed. April 17 - Arsh - 1:30pm to 2:30pm
    - Wed. April 17 - Rohan - 2pm to 3pm
    - Wed. April 17 - Anne - 2:30pm to 4pm
    - Thursday April 18 - Sitong - 8:30am to 11:30am
    - Friday April 19 - Christian - 10am to 3pm
    - Friday April 19 - Anne - 3pm to 4:30pm

- Make sure you have a fixed copy of maze_1.txt by downloading maze_1.txt again.

- Turtle Maze Calculation set of slides (seen in Monday's lecture) has now been posted!
  These slides provided some help with the function coordinateToMazePosition(x, y, cellSize = 20).

- Deadline is Wed. April 10. If you have questions regarding the marking of your Assignment 1 to Assignment 3 and regarding your midterm, make sure you bring your questions to the instructor and/or the TAs by Wed. April 10. We shall not be attending to such questions beyond this deadline!

Last Lecture

- We had a look at **Merge sort**
- We started introducing **Complexity Analysis** and the **Big O Notation**

Review - Complexity Analysis

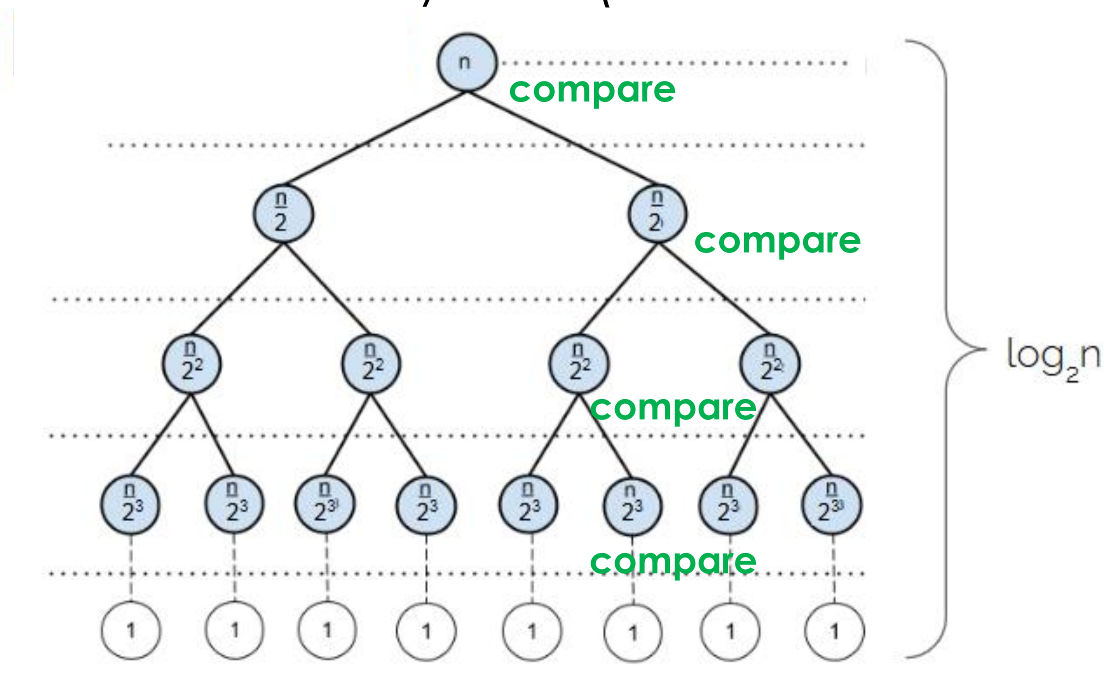
- To compare algorithms, one analyzes the **complexity** of the algorithms, i.e., one considers the amount of resources these algorithms required to perform the same task
- The result of this complexity analysis is called the **Order of an algorithm** (or their time and space complexity/efficiency)
  - To perform **Complexity Analysis**, i.e., to figure out the order of an algorithm, one counts the ***number of operations:*** **comparisons**
  - And since the larger the **list** is, the more operations may be required -> correlation between **# of operations** and **n**
  - **Order of an algorithm** is expressed as a function of **n**
  - We express this function of **n** by using the **Big-O notation**

Today's Menu

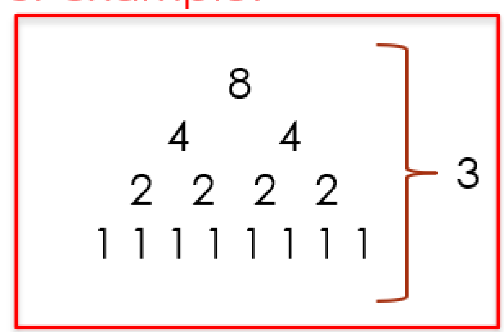
- We shall analyze our **searching** algorithms and figure out which is faster?
  - **Linear search** or **Binary search**
- We shall analyze our **sorting** algorithms and figure out which is faster?
  - **Selection sort** or **Merge sort**

Time Complexity of Binary Search

- How many times do we compare ***target*** with ***element in the middle***?
  - How many times do we divide **n** by 2 until we can't divide anymore (sublists have 1 element)?



For example:



If **n** is 8, we divided 3 times -> $\log_2 8 = 3$

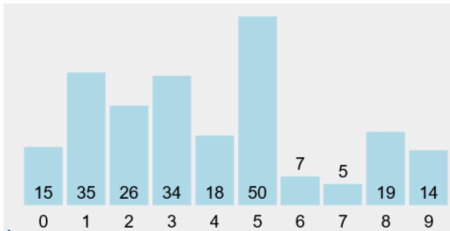
Worst case scenarios:

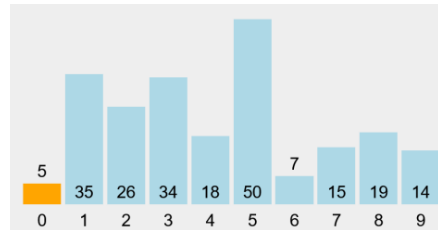
- Target is either found in a sublist of 1 element or not found -> **$O(\log_2 n)$**

Time Complexity of Selection Sort

n = 10

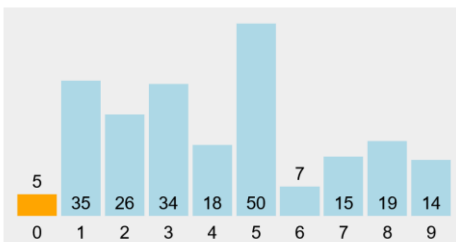
Iteration #1

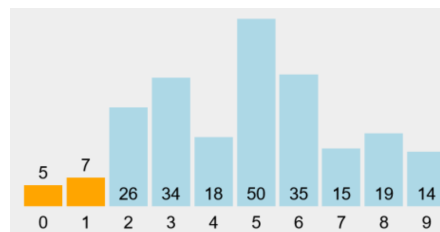




n-1 -> 9 comparisons

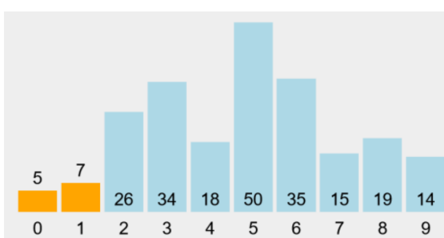
Iteration #2

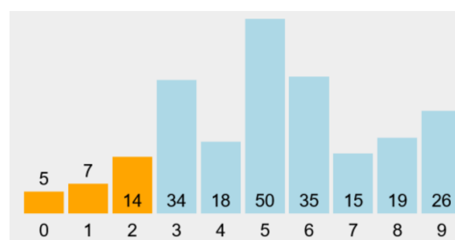




n-2 -> 8 comparisons

Iteration #3

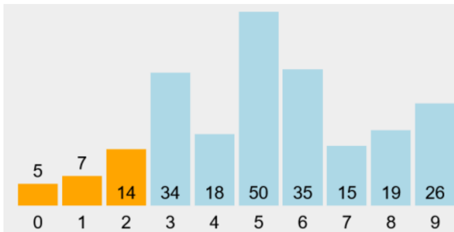


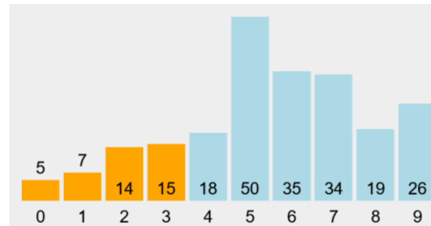


n-3 -> 7 comparisons

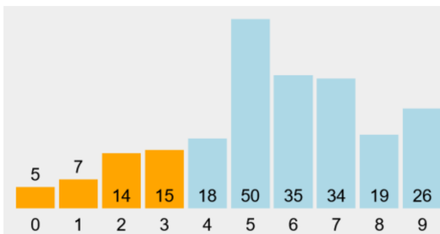
Time Complexity of Selection Sort

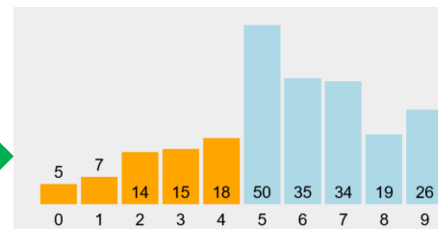
Iteration #4



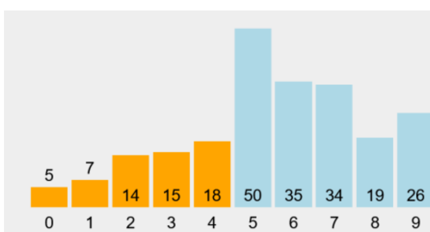


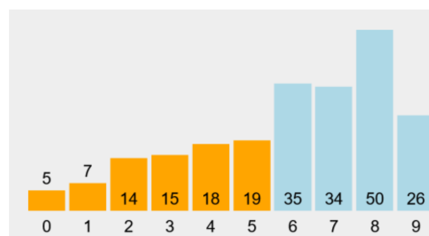
Iteration #5



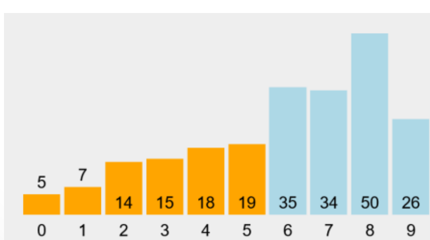


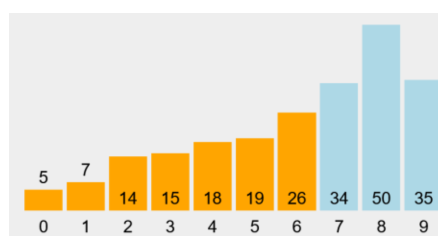
Iteration #6





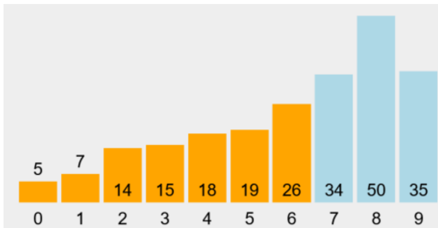
Iteration #7

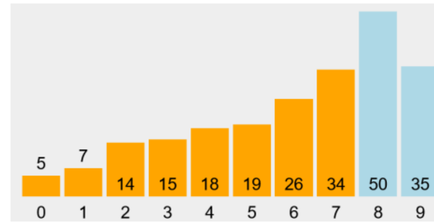




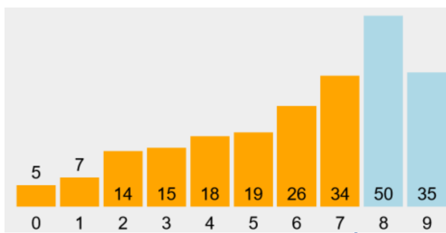
Time Complexity of Selection Sort

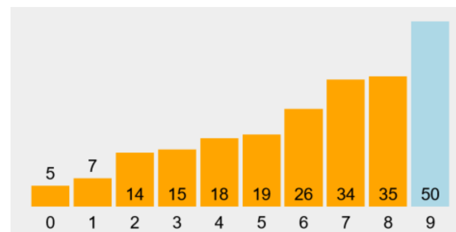
Iteration #8





Iteration #9
Iteration #n-1





1 comparison

All sorted! ☺

- The total number of comparisons is:

$$= \mathbf{n}-1 + \mathbf{n}-2 + \mathbf{n}-3 + \ldots + \mathbf{1}$$

$$= \frac{n(n-1)}{2}$$

$$= \frac{n^2}{2} - \frac{n}{2}$$

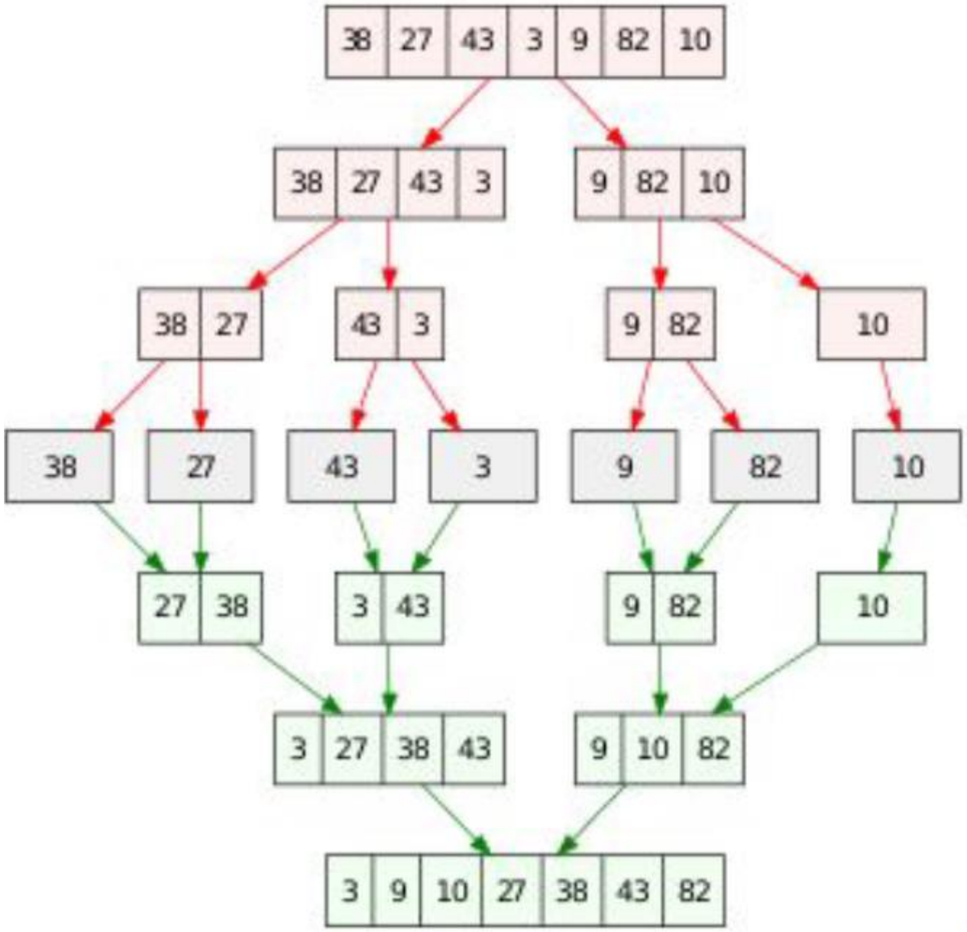
Selection sort is **quadratic, O(n²).** This is true for both the **best** and **worst case (why?)**

Space Complexity of Selection Sort

- Because **Selection sort** is an **in-place** algorithm
  -> -> **O(1)**

Time Complexity of Merge Sort

1. Like Binary search, we divide $\log_2$ **n** times producing $\log_2$ **n** levels
2. In order to perform the Merge operation and sort all elements, we end up comparing all elements, i.e., **n** comparisons, at each of the level.



Worst case scenarios:

- **n** comparisons done at each of the $\log_2$ **n** levels = **n** * $\log_2$ **n** -> **O(n $\log_2$ n)**

Space Complexity of Merge Sort

- Because extra space is needed to create the merged lists at each level, **Merge sort** has a space efficiency of **O(n)**

- This may be problematic for large **n**

Conclusion

When we analyze the complexity of algorithms, we consider the algorithm's **worst case scenario**

Time Efficiency

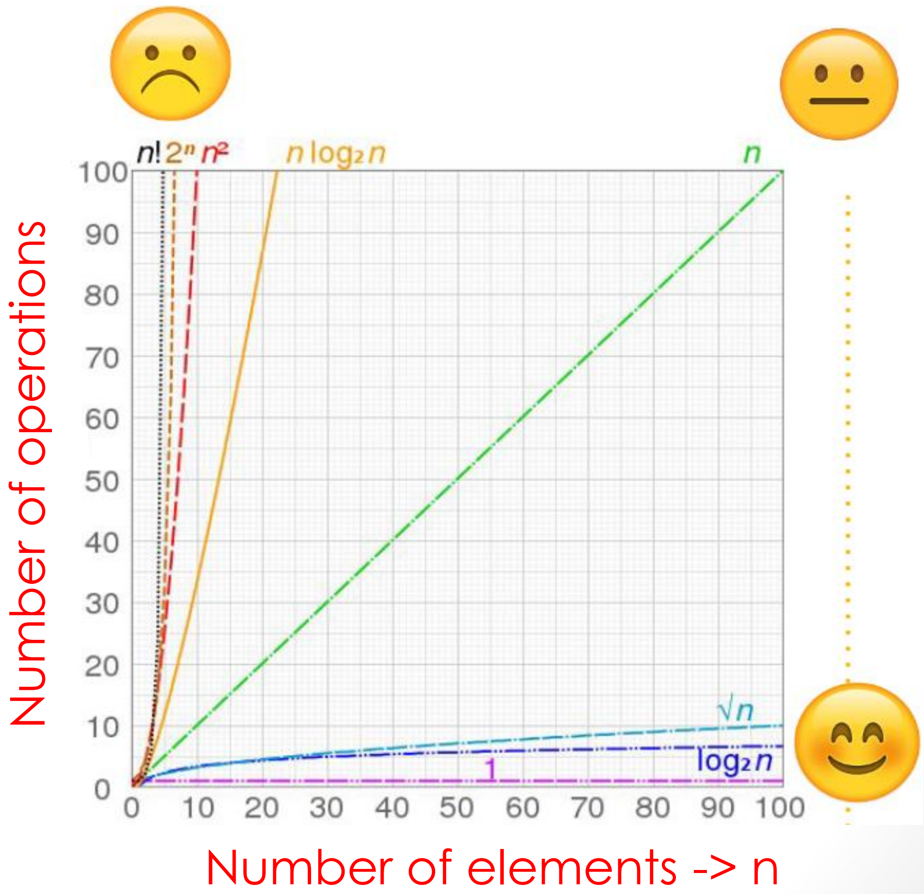
Linear search -> **O(n)**

Binary search ->**O(log$_2$ n)**

Selection sort -> **O(n^2)**

Merge sort ->**O(n log$_2$ n)**

So, which **search** algorithm is faster? And which **sort** algorithm is faster?



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

What happens when n gets larger?

- When we analyse the complexity of an algorithm, we look at the time/space it requires to execute "as **n** goes to infinity"

n	Ex: Get first element in list **O(1)** Always constant	Ex: Linear search **O(n)**	Ex: Selection sort **O(n^2)**	Ex: Binary search **O(log n)**
10	1	10	100	3.32
100	1	100	10,000	6.64
1000	1	1000	1,000,000	9.96
10000	1	10000	100,000,000	13.28

Last Lecture – Wow!

- Practice Exam 10 + in-class activity #10