CS2110: SW Development Methods

Analysis of Algorithms

~Searching & Sorting~

- Reading: Chapter 5 of MSD text
  - Except Section 5.5 on recursion (next unit)

Announcements

- **Homework #4 (GUIs)** [*No automatic Web-CAT feedback*]
  - Submit on Collab – **Zip folder with all files & your pictures**
  - Due: 11:30pm, Wednesday, April 1, 2020
  - Feel free to include a `README.txt` file to provide instructions/clarity
  - *Don't forget to cite all sources within your code!*

- **Exam 2** on **Friday, April 3, 2020**
  - **SDAC** students please make sure your SDAC arrangements have been finalized with SDAC so we can provide you with appropriate accommodations.
  - Similar **format** to Exam 1 (Questions on Collab + Coding questions on external website)
  - Details will be given during the Exam review (class before the exam.)

REMINDER: Common Order Classes

- Order classes group "equivalently" efficient algorithms
  - $O(1)$ – constant time! Input size doesn't matter
  - $O(\lg n)$ – logarithmic time. Very efficient. E.g. binary search (after sorting)
  - $O(n)$ – linear time E.g. linear search
  - $O(n \lg n)$ – log-linear time. E.g. best sorting algorithms
  - $O(n^2)$ – quadratic time. E.g. poorer sorting algorithms
  - $O(n^3)$ – cubic time
  - ….
  - $O(2^n)$ – exponential time. Many important problems, often about optimization

Inverse: Another Perspective

Largest size *n* of a problem that can be solved in time *t*

For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds. A microsecond is 1 millionth of a second. (Hint use wolframalpha.com)

$f(n)$	1 sec	1 day	1 year	1 century
$\lg(n)$	2^{10^6}	$2^{8.64\times10^{10}}$	$2^{3.1563\times10^{13}}$	$2^{3.1563\times10^{15}}$
$\sqrt{n}$	10^{12}	7.46×10^{21}	9.95×10^{26}	9.96×10^{30}
n	10^6	8.46×10^{10}	3.15×10^{13}	3.16×10^{15}
$n\lg(n)$	62,746.1	2,755,147,513	797,633,893,349	6.86×10^{13}
n^2	10^3	293,938	5,615,692	56,176,151
n^3	10^2	4,420	31,593	146,679
2^n	$\frac{6}{\log_{10} 2}\approx 19.9$	36	44	51
$n!$	9.45 (between 9 and 10)	13	16	17

Searching and Sorting

Important and Useful Algorithms

Two Important Problems

- **Search**
  - Given a list of items and a target value
  - Find if/where it is in the list
    - Return special value if not there ("**sentinel**" value)
  - Note we've specified this at an *abstract* level
    - We'll see a few examples on how to implement searches today
- **Sorting**
  - Given a list of items
  - Re-arrange them in some non-decreasing order

- With solutions to these two, we can do many useful things!
- **What to count for complexity analysis?** For both, the basic operation is: ***comparing two list-elements***

Searching Algorithms

Professor Snape says…

From: Harry Potter and the Prisoner of Azkaban

- *Ron Weasley's linear search method*
- *Professor Snape's Magical constant time algorithm with a wand!*

Sequential Search Algorithm

- **Sequential Search**
  - AKA linear search
  - Look through the list until we reach the end or find the target
  - Best-case? Worst-case?
  - **Complexity: O(n)**
  - Advantages: simple to code, no assumptions about list
  - Think:
    - Finding a card in a shuffled deck
    - Finding an element in an arrayList
    - …

Sequential (Linear) Search

Case	Best Case	Worst Case	Average Case
Item is present	1	n	$n/2 = n$
Item is not present	n	n	n

Binary Search Algorithm

- **Binary Search**
  - **Precondition: Input list must be sorted**
  - Strategy: Eliminate about half items left with one comparison
    - Look in the middle
    - If target larger than middle element, must be in the 2nd half
    - If target smaller than middle element, must be in the 1st half
- **Complexity: $O(\log_2 n)$**
- Must sort list first, but …
- **Much** more efficient than sequential search
  - Especially if search is done many times (sort once, search many times)

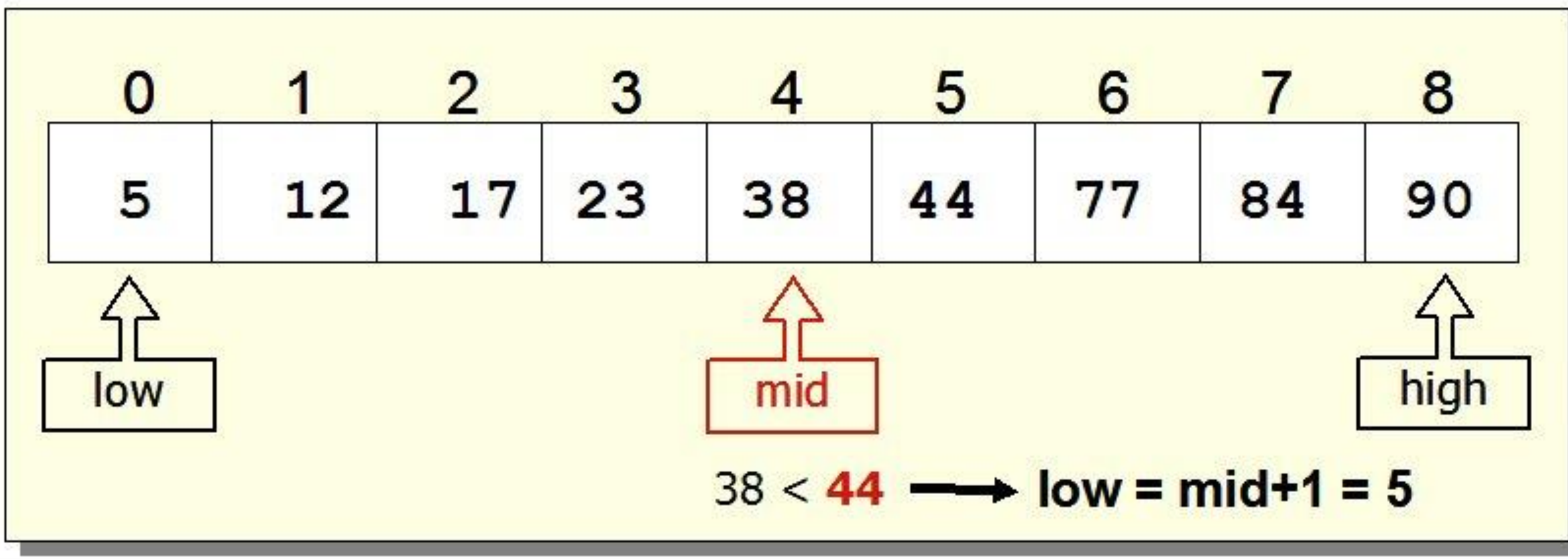
- Note: Java provides static `binarySearch()` method

How to find the mid-point?

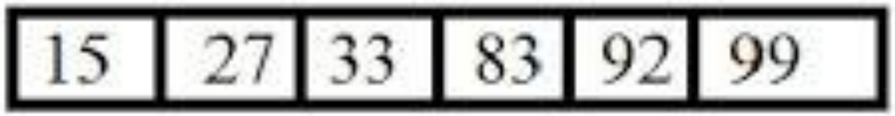
	low	high	mid
#1	0	8	4

search(44)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

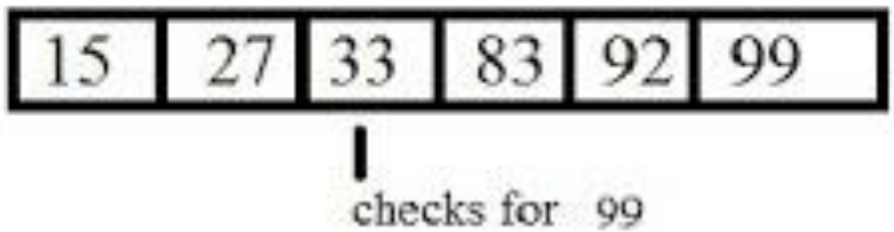


Example



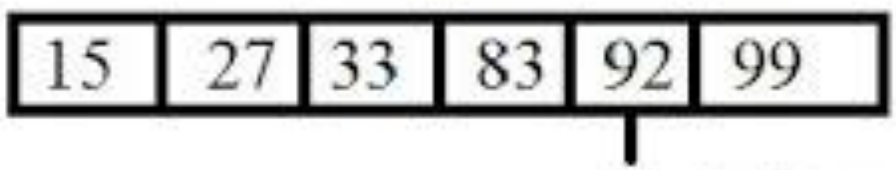
Search=99, low=0, high=5, mid=(0+5)/2 = **2**

Pass 1



low=2**+1**=3, high=5, mid=(3+5)/2 = **4**

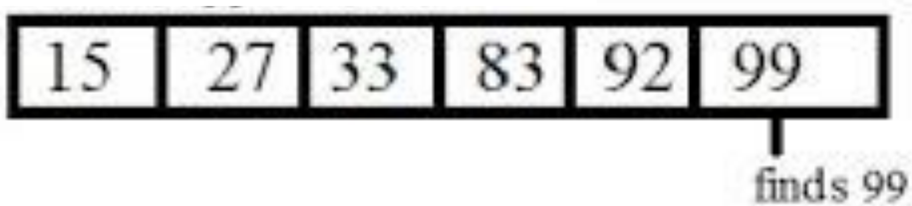
Pass 2



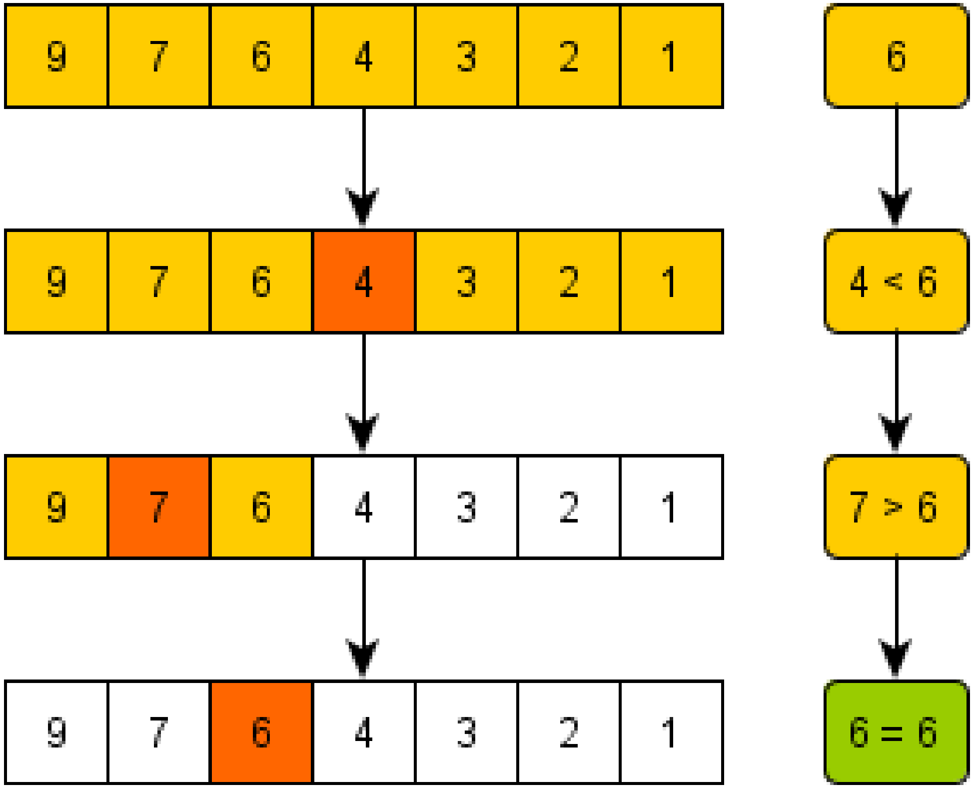
checks for 99

low=5, high=5, mid= **5**

Pass 3



Another Example



In-Class Assignment: Binary Search

- For each of the following arrays of integers, use **binary search** to find the target value in the list. You must turn in:
  - Which index is returned by the algorithm
  - The sequence of indices whose values were *compared to the target*

1.

-1	4	5	11	13

target: 4

2.

-1	4	5	11	13

target: 13

3.

-5	-2	-1	4	5	11	13	14	17	18

target: 3

4.

-5	-2	-1	4	5	11	13	14	17	18

target: 14

Sample Solution to the first Binary Search Question:

1.

-1	4	5	11	13

target: 4

- Ex #1: Index returned: **1**
  Sequence of indices: **2 0 1**

- Please work on Examples 2, 3, and 4.

- Submit solutions to examples **1 through 4**
- Submit individually on Collab

Binary Search

Best Case	Worst Case	Average Case
1	*log n*	*log n*

Discussion Question: Given an unsorted list…

- **Binary search** is faster than sequential search
  - But extra cost! Must <u>sort</u> the list first!
  - It costs O(n log n) to sort --- if we use fastest sorting algorithm
  - Sorting + searching once = O(n log n) + O(log n) = O(n log n)
  - Linear search once = O(n)

So, when do you think it's worth using binary search?

Discussion Question: Given an unsorted list…

- **Binary search** is faster than sequential search
  - But extra cost! Must sort the list first!
  - It costs O(n log n) to sort --- if we use fastest sorting algorithm
  - Sorting + searching once = O(n log n) + O(log n) = O(n log n)
  - Linear search once = O(n)

So, when do you think it's worth using binary search?

- When you are searching many times, it's more efficient to sort once then perform MANY ***binary searches*** (compared to many *linear searches*)
- How many searches? Binary search overtakes the cost of the sort **after "*n*" searches!**

Comparison (Summary: Searching)

- Linear search vs. Binary search

Sequential (Linear) Search

Case	Best Case	Worst Case	Average Case
target present	$O(1)$	$O(n)$	$O(n)$
target not present	$O(n)$	$O(n)$	$O(n)$

Binary Search

Case	Best Case	Worst Case	Average Case
target present	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$
target not present	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$

Sorting Algorithms

Sorting Algorithms

- **The sorting problem:**
  - Given a sequence $\mathbf{a_0 \ldots a_n}$ reorder them into a permutation $\mathbf{a'_0 \ldots a'_n}$ such that $\mathbf{a'_i <= a'_{i+1}}$ for all pairs
    - Specifically, this is sorting in non-descending order …
  - Basic operation: Comparison of keys

- Supplemental material on a couple of these sorting algorithms [resources]
  - Will discuss Mergesort in more details later

- Cool Visualization:
  https://www.toptal.com/developers/sorting-algorithms

How to Sort?

- Many sorting algorithms have been found!
  - Problem is a case-study in algorithm design
  - You'll see more of these in CS 2150 and CS 4102

- Some "straightforward" sorting algorithms
  - Insertion Sort, Selection Sort, Bubble Sort
  - **Each is $O(n^2)$**
- More efficient sorting algorithms

Best Sorts are O(n log n)

  - Quicksort, Mergesort, Heapsort
  - **Each is $O(n \log n)$**

Sorting in CS2110

- **Collections** and **Arrays** classes provide `.sort()` methods!
  - Utilizes `compareTo()` or `Comparator` to ***determine order*** when comparing elements
  - "under the hood", it's a *variant* of something called ***mergesort***
  - Θ(n log n) worst-case -- as good as we can do
  - We'll discuss how Mergesort works soon!

Reminder: What to Count

- Often count some "*basic operation*"
- Or, we count a "critical section"
- Examples:
  - The block of code most deeply nested in a nested set of loops
  - An operation like comparison in **sorting**
  - An expensive operation like multiplication or database query

Summary and Major Points

- When we measure algorithm complexity:
  - Base this on **size of input**
  - Count some *basic operation* or how often a *critical section* is executed
  - Get a formula **f(n)** for this
  - Then we think about it in terms of its “label”, the order class O(f(n))
    - “Big-Oh” means as efficient as class f(n) or better
    - *Upper bound* on how inefficient the algorithm is
  - We usually use order-class to **compare** algorithms
  - We can measure worst-case, average-case
- Data structures are design choices that implement ADTs
  - *How to choose?* Often by the efficiency of their **methods**