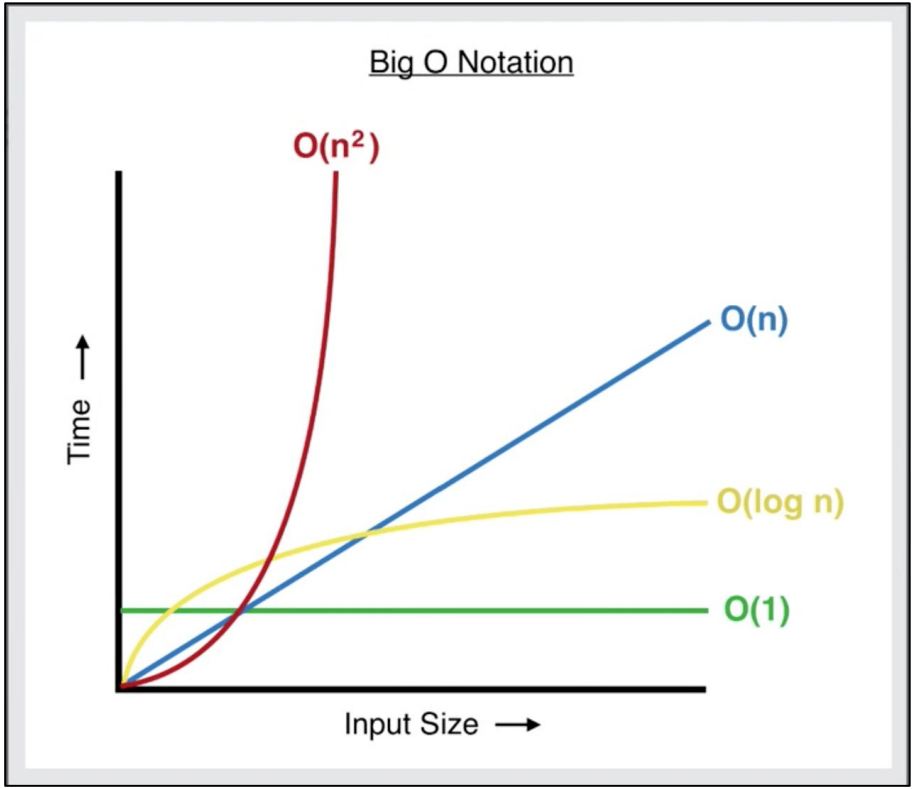
Chapter 6: Algorithms

CSE 2010
Week 9

What is an algorithm?

- Given a problem, an algorithm is a specific procedure for solving that problem.
- The development of efficient algorithms is essential to all areas of computer science (artificial intelligence, security, networking, operating systems, etc), and even other fields such as physics and math (root-finding algorithms, definite integral estimation algorithms, etc).
- We can develop many algorithms to solve one problem, but they will vary in efficiency and complexity.
- In computer science, we care about time complexity (how long it takes to run the algorithm), and we usually use Big-O notation to represent this.



Algorithms

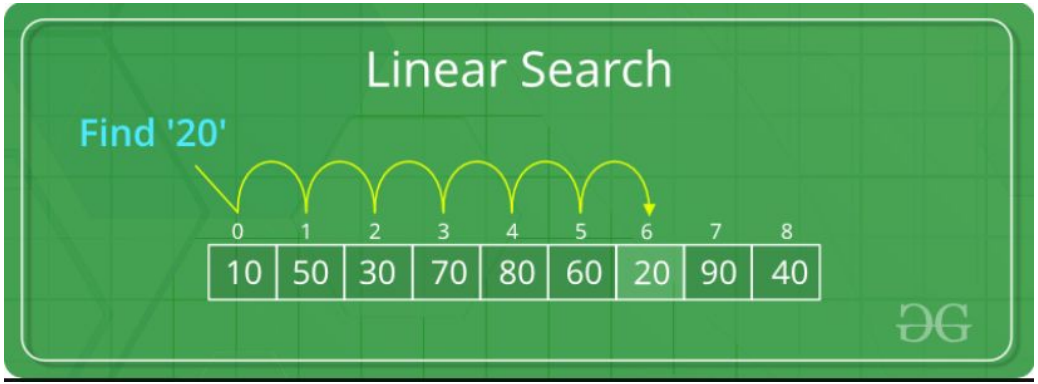
- We use algorithms in everyday life, without even thinking about it.
- For example, let's say you have to do laundry. One load of laundry needs to go in the washer (30 minutes), and dryer (30 minutes).
- Given 5 loads of laundry, we can take two different approaches:
  - Algorithm 1: Completing one load before moving on to the next
    - Put load in the washer. Once complete, put it in the dryer. Once the dryer is complete, load the second load into the washer. Repeat until all laundry is done.
      - Approximately 5 hours.
  - Algorithm 2: Overlapping of loads
    - Put load in the washer. Once complete, put it in the dryer and also put the second load into the washer. Repeat until all laundry is done.
      - Approximately 3 hours.
- Both approaches take care of the laundry problem, but one is obviously faster than the other.

Search Algorithms

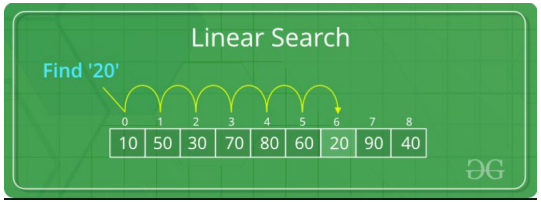
- Search algorithms are used to search for a given value in a container of elements (array, vector, other data structures).
- For instance, we may want to know whether or not a specific value is stored in the container, or how many times a specific value appears in a container.
- For this class, we will examine two basic search algorithms (which you will see later on in your CSE journey):
  - Linear Search
  - Binary Search

Linear Search

- Given a container that stores an indexed sequence of `n` elements, we want to search for a value, `x`, by starting from the first element, and sequentially checking every element until one of the following happens:
  - We find `x`
  - We check every element in the container and we don't find `x`
- What are some real life applications of this....



Linear Search Algorithm Steps

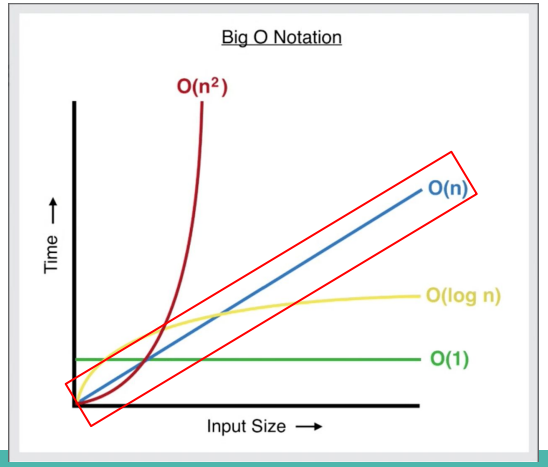


1. Start from the first element (index 0).
2. Compare that element to x
   a. If element == x, then you found it! You can stop here.
   b. If element != x, AND there are still more elements to check go to to Step 3.
   c. If element != x, AND there are no more elements to check, go to Step 4.
3. Move to the next element, and repeat Step 2.
4. If you've reached the end of the container and have not found x, then we know for certain that x is not in the container.

Let's take a closer look at an example....

Linear Search Summary

- Given a container of `n` elements:
  - The least # of checks that we would have to use with linear search is 1, in the case that the first element is the element we are searching for.
  - The most # of checks that we would have to use with linear search is `n`, in the cases where the value we are searching for is not in the container OR it is the last element.
- Time complexity takes into consideration the best, worse, and average case of linear search, so for reasons that some of you will learn in CSE 4310, the time complexity of linear search is O(n).
- Pros:
  - Easy to understand and implement
  - Can be applied on an unsorted container
- Cons:
  - Not very efficient at all.
  - If n is large enough, linear search would take forever.



Binary Search

- Given a SORTED container that stores an indexed sequence of `n` elements, we want to search for a value, `x`, by repeatedly dividing the search interval in half.
  - The first search interval is the entire container (index 0 → n-1).
  - We first check the middle element.
    - If the middle element is equal to `x`, woohoo, we found it!
    - If the middle element is greater than x, then we change the search interval to the first half of the container.
    - If the middle element is less than x, then we change the search interval to the second half of the container.
  - We repeat his until:
    - We find `x`
    - There is no longer anywhere left to search and we don't find `x`
- What are some real life applications of this....

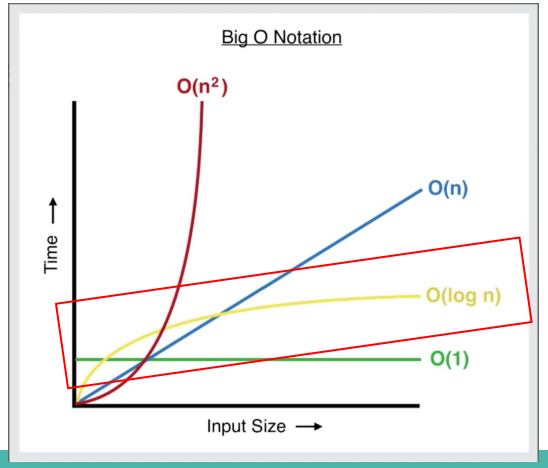
Binary Search Algorithm Steps

1. Set your first search interval: low index = 0, high index = n-1
2. The first element to check is the middle element which is (low+high)/2
3. Compare the mid element to x
    a. If mid element == x, then you found it! You can stop here.
    b. If mid element != x and there is still a search interval
        i. If the mid element is greater than x, then high = mid - 1, and go to step 4.
        ii. If the mid element is less than x, then lo = mid + 1, and go to step 4.
    c. If mid element !=x and there is no more search interval, go to step 5.
4. Set new mid, (low+high)/2, and repeat Step 3.
5. If there is no more search interval and you still have not found x, we know for certain x is not in the container.

Let's take a closer look at an example....

Binary Search Summary

- Given a container of `n` elements:
  - The least # of checks that we would have to use with binary search is 1, in the case that the first middle element is the element we are searching for.
  - The most # of checks that we would have to use with linear search is $\log_2(n)$, in the cases where the value we are searching for is not in the container or in the last search interval.
- The time complexity of binary search is $O(\log_2 n)$.
- Pros:
  - MUCH faster than Linear Search
  - Potential for recursion (we will see in Chapter 10)
- Cons:
  - Has the precondition of needing to be sorted



Linear vs Binary Search Examples

Number of elements (n)	Maximum Number of Comparisons / Checks	
	Linear Search	Binary Search
100,000	100,000	16
200,000	200,000	17
400,000	400,000	18
800,000	800,000	19
1,600,000	1,600,000	20