Search

- Find an element within a collection.

- Most common:
  - linear search
  - binary search

- Other search:
  - ternary search
  - interpolation search
  - jump search
Linear Search

- Looks at each element of the list sequentially until either find the target value or finish the list without finding it.

Look for 8

- Best case
  - First element $O(1)$
- Average case
  - compare half of the list $O(n)$
- Worst case
  - compare all elements $O(n)$
Binary Search

- Find a value in a sorted array
- It starts at middle, compares middle element
- Base on the result, it eliminates half of the list
- Then search the remaining half

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>50</th>
</tr>
</thead>
</table>

Look for 8

- After each comparison, half the array that’s searching on is eliminated
- Average case & Worst case
  - O(log n)
- Note:
  - For sorted array only, not efficient for linked list.
Ternary Search

- Similar to binary search, but it compares two mid elements at each iteration
- It starts at a third and two thirds
- Base on the result, it eliminates two thirds

Look for 8

- After each comparison, two thirds the array that’s searching on is eliminated
- Looks like its runtime is better than binary, however, the average case & worst case is still the same
  - $O(\log n)$
Interpolation Search

- Similar to binary search but
- Instead of starting at middle, it predicts where the element should be in the remaining of the list

- Similar as looking up a contact in a phone book, you won’t look in the middle for name Brian
- It works best if elements are uniformly distributed
  - Average case
  - $O(\log \log n)$
- Otherwise, bad prediction every time, end up looking at almost the entire list.
  - Worst case
  - $O(n)$
Interpolation Search

- First, predicts where the element should be in the remaining of the list

- Look for 6

- How to predict?
- Assume values are uniformly distributed, where 6 is in the list, is proportional to where 6 is in the element range
- Element range 2 to 16, total 8 elements
- Index range 0 to 7

\[
\text{position from first index} = \frac{(\text{value from first element index range}) - (\text{first element range})}{(\text{element range})} \times (\text{element range}) + \text{first element range}
\]

\[
\text{pos of 6} = \frac{(6 - 2)}{(16 - 2)} \times (7 - 0) + 0 = \frac{4}{14} \times 7 = 2
\]
Jump Search
- Instead of looks at each element of the list sequentially as linear search, it skips a number of elements to check. Once it passes the element it looks for, it’ll go back to check \( \downarrow \) previously skipped element.

| 2 | 4 | 6 | 8 | 10 | 20 | 30 | 50 |

Look for 10, skip two elements each time

- Run-time depends on the number of elements skips each time, we call it a block.
  - A loop to find the block
  - Follow by a loop find the elements in the selected block.
  - \( O(#\text{block}) + O(\text{block size}) \)
  - Optimum: \( #\text{block} == \text{block size} \)
  - Since \( #\text{block} = n / \text{block size} \)
  - block size = \( \sqrt{n} \)