Today
- Binary Search (cont)
- Algorithm Analysis (cont)

Assignments
- HW 5 out (due Thurs)
- Quiz tues (merge sort, quick sort, detailed analysis)
**Binary Search**

The basic idea:  
... note: for HW-5, must implement iteratively

Iteratively search for a value $v$ in a non-empty sorted list

1. Pick the **middle item** in a list
2. If $v < \text{middle item}$, then search in *left half* of list
3. If $v > \text{middle item}$, then search in *right half* of list
4. If $v == \text{middle item}$, then found match

Trace the steps to find "C" using binary search:

```
A B C D E F G
```

... Input

```
A B C D E F G
```

... Pick middle item in list

C < D

```
A B C
```

... New input

```
A B C
```

... Pick middle item in list

C > B

```
C
```

... New input

```
C
```

... Pick middle item in list

C = C

... Found match!
(3). BinSearchMap:

- Uses an underlying `ArraySeq<std::pair<K,V>> seq;` member variable
- Goal is to optimize `contains` by using binary search
  - To do this, we need to keep the underlying `seq` sorted
  - `sorted_keys()` also won’t need to call `ArraySeq::sort()`
- Must implement a private helper function:
  
  ```cpp
  bool bin_search(const K& key, int& index) const;
  ```

- The helper iteratively finds index of given key via binary search
  - if the key isn’t in the map, returns “closest” index
  - but, always returns a valid index (for a non-empty sequence)

- Must use `bin_search` helper to implement:
  - `insert` ... find index to insert at (roughly)
  - `erase` ... find index to erase
  - `find_keys` ... find index for first key of range

- Q: What are the expected performance improvements and trade-offs?
  - faster `contains`, `erase`, and `sorted_keys`
  - faster `find_keys` depending on “selectivity” of range
  - slower `insert` compared to just inserting at end of `seq`
seq = [10 30 50 70 90] ... "Keys"

Private helper in BinSearchMap:

    bool bin_search(const K& key, int& index) const;  
    – always “returns” a valid index, unless seq empty

Examples (using above 5-elem seq object):

    int index = -1; // can set to anything ...

    (1). bin_search(30, index); // true, index = 1
    (2). bin_search(90, index); // true, index = 4
    (3). bin_search(5, index); // false, index = 0
        – for insert call: seq.insert(5, index);
    (4). bin_search(95, index); // false, index = 4
        – for insert call: seq.insert(95, index + 1);
        – since 95 > seq[index]
    (5). bin_search(60, index); // true, index = 3
        – for insert call: seq.insert(60, index);
    (6). bin_search(40, index); // true, index = 1
        – for insert call: seq.insert(40, index + 1);
        – since 40 > seq[index]
Asymptotic Notation

Two algorithms with same “growth rate” treated as same “complexity”

• e.g., two linear-time algorithms are of similar complexity
• even if they have different primitive operation-count equations

Basic high-level idea:¹

\[ \text{suppress } \underbrace{\text{constant factors}}_{\text{too system-dependent}} \text{ and } \underbrace{\text{lower order terms}}_{\text{irrelevant for large inputs}} \]

Three main notations for comparing algorithm growth rates:

• “Big-O” \((O\text{-notation})\) ... “not slower than” (upper bound)
• “Big-Ω \((Ω\text{-notation})\) ... “not faster than” (lower bound)
• “Big-Θ \((Θ\text{-notation})\) ... “exactly” (tight bound)

Primarily Big-O notation in this class

• For example, saying an algorithm has a linear-time Big-O growth rate
• ... implies its worst case isn’t worse than linear time

Note the three notations can be used in different contexts

• We could say an algorithm’s average case has a linear-time Big-O growth rate
• ... implying its average case isn’t worse than linear-time

¹From: Algorithms Illuminated