Today

- Quiz 5
- Searching (wrap up)
- Big-O intro

Assignments

- HW5 out (due Tues)
Recall **Linear Search**:  
Iteratively search for a value $v$ in a non-empty sorted list  
1. For each item in the list:  
2. If $v == \text{item}$, then found match  

**Versus Binary Search**:  
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
Determining worst-case time complexity of binary search (by "cases")

\[ T(1) = c \]
\[ T(n) = c + T(n/2) \]

... A "recurrence" relation

Solving the recurrence relation (intuitively)

- How many times do we halve \( n \) to get to 1?
- Similar to asking how many times we multiple by 2 to get \( n \) ...

\[
\begin{align*}
2^0 &= 1 & \text{if } n = 1 \text{ don’t multiply by 2} \\
2^1 &= 2 & \text{if } n = 2 \text{ multiply by 2 one time to get } n \\
2^2 &= 4 & \text{if } n = 4 \text{ multiply by 2 two times to get } n \\
2^3 &= 8 & \text{if } n = 8 \text{ multiply by 2 three times to get } n \\
2^4 &= 16 & \text{if } n = 16 \text{ multiply by 2 four times to get } n
\end{align*}
\]

- Recall that if \( \log_2 n = x \) then \( 2^x = n \) ... and we’re looking for the \( x \)
- This means \( \lfloor \log_2 n \rfloor \) is the number of times we halve \( n \) to get 1
  - e.g., if \( n = 7 \), we halve 2 times, not 2.807... times

The overall cost is then:

\[ T(n) = c\lfloor \log_2 n \rfloor + c \]
- each iteration requires \( c \) steps
- and we iterate \( \lfloor \log_2 n \rfloor \) times (to get to 1 element left)
Proof (using induction)

- Base Case: $T(1) = c \lfloor \log_2 1 \rfloor + c = c \cdot 0 + c = c$
- Assume: $T(n/2) = c \lfloor \log_2 (n/2) \rfloor + c = c \lfloor \log_2 n \rfloor - c + c = c \lfloor \log_2 n \rfloor$
- Inductive step: $T(n) = c + T(n/2) = c + c \lfloor \log_2 n \rfloor$

We call this a “logarithmic” time algorithm ... which is faster than linear time!

Comparing growth rates of $n$ versus $\log_2 n$
Our next implementation of key-value collections (HW-5)

- Use a vector implementation again

- But this time keep it sorted to leverage binary search
  Requires change to insert ...

  Q: How do we do maintain sorted order of list?

  One option: insert at end then sort list (we’ll see this is slow)
  Better: use binary search to find index where new pair should go

- Use binary search for both find and range search

  Q: How would range search work (i.e., keys in low/high range returned)?

  Use binary search to find position of first key, then traverse forward

- Use binary search for remove
  - Find the element to remove, then remove it

- Don’t have to do anything to sort the list!
  - Note that sort is the slowest of the functions in HW-3 and HW-4
  - We still have to return a (sorted) list of the keys (which is $O(n)$)
Complexity Analysis Continued

**Big-O notation focuses on algorithm growth rates**

- an upper bound such that running time will not be worse
- independent of particular implementation or computing device

Given algorithm \( A \), we say \( A \) requires (worst-case) time proportional to \( f(n) \)

- \( n \) is the size of the input
- \( f(n) \) represents \( A \)'s growth rate as a function of \( n \)
- we say that “\( A \) is order \( f(n) \)”
- we write \( O(f(n)) \)

Examples:

- If \( A \) requires time proportional to \( n \), then \( f(n) = n \)
  - we write this as \( O(n) \) ... and say that \( A \) is “order” \( n \)

- If \( A \) requires time proportional to \( n^2 \), then \( f(n) = n^2 \)
  - we write this as \( O(n^2) \) ... and say that \( A \) is “order” \( n^2 \)
Notice that we don’t include constant terms

e.g., instead of writing $3n + 2$, we just write $O(n)$

and instead of writing $6n^2 + n - 8$, we just write $O(n^2)$

• We are interested in algorithm’s general rate of growth (orders of magnitude)
• Which makes it easier to broadly compare them

Order-of-magnitude analysis … Why can we remove the constants?

• Focus on input sizes large enough to make only the growth relevant
  … thus, as the input size increases without bound (in the limit)

• Sometimes called the "asymptotic" efficiency of algorithms