Today

- Searching (cont)

Assignments

- HW5 out (due Tues)
REVIEW: Finding elements in kv collections

```cpp
template< typename K, typename V >
class Collection
{
public:

    ... // find the value associated with the key
    bool find( const K& key, V& val ) const;

    ...
};
```

Q: Given different implementations what are some ways we can compare them?

lots of possible ways ... for example:

- readability, lines of code, number of if-statements / subcases
- how much memory they use (space)
- how long they take (time)
- average case, best case, worst case scenarios

We’ll mainly focus on worst-case **time complexity**:

- time complexity = how “long” algorithms take
What we really mean by "time"

- we focus on the number of basic operations (steps)
- we assume each step takes roughly the same amount of time
- we want to know how many "steps" needed relative to the input size

For example: "Find" a value at a particular array index

```cpp
typedef pair<string, int> pair_t;

// "magically" know index to look for value
void find_val(const pair_t array[], int n, const string& k, int& v)
{
    int i = magic(k);  // finds i in 1 step
    v = array[i].second;
}
```

Assume:

- size of the input is \( n \) (i.e., \( n \) array elements)
- \( \text{array}[i] \) is one step (array lookup)
- accessing second is one step
- assignment is one step

As a function \( T \) over input size \( n > 0 \), we have that \( T(n) = 5 \)

- this is a "constant time" function (algorithm)

Constant time functions are boring but efficient!
Another example:

```c
void find_val(const pair_t array[], int n, const string& k, int& v) {
    for (int i = 0; i < n; ++i)
        if (array[i].first == k) {
            v = array[i].second;
            return;
        }
}
```

Assume:

- assignment counts as one step
- comparison counts as one step
- array access counts as one step
- accessing first and second each count as one step
- increment counts as one stop
- returning is “free”

**Worst Case** = upper bound on number of steps (i.e., won’t do worse)

Q: What is the “worst case” scenario for this version of `find_val`?

- When `k` is the last element in the vector!
Q: What is the worst case cost of find_val as a function of input size $n$?

- $i=0$, $i<n$, ++$i$, array[$i$], .first, == done every pass ... $6n$
- Last case requires three extra steps (assign, array and member access)
- So: $T(n) = 6n + 3$
- Called a “linear” time function/algorithm

Linear time is a bit less boring than constant time, but also generally efficient
As we go, we'll do more examples

In general, we'll focus on two types of analysis:

- **Detailed**: counting steps (like above)

- **Order-of-Magnitude** (big-O): growth rates (e.g., constant vs linear)
Binary Search

**Goal:** See if we can improve the (time) efficiency of our collection functions

- specifically: insert, find value, find range, remove, and sort
- as we go, we’ll see there are trade-offs (e.g., faster find, slower insert)

Our first try is related to **sorting** ...

- in particular, we can improve search if keys are in sorted order

Q: How does sorted order help to find data faster?

- using binary search!

---

The basic idea of **binary search** (informal)

Iteratively search for a value \( v \) in a non-empty sorted list

1. Pick the **middle item** in a list
2. If \( v < \) **middle item**, then search in *left half* of list
3. If \( v > \) **middle item**, then search in *right half* of list
4. If \( v == \) **middle item**, then found match
Exercise: Trace the steps to find “C” using binary search:

\[ \begin{array}{cccccc}
A & B & C & D & E & F & G \\
A & B & C & D & E & F & G \\
\end{array} \]

\begin{array}{c}
\vdots \text{ Input} \\
\vdots \text{ Pick middle item in list} \\
\end{array}

C < D

\begin{array}{ccc}
A & B & C \\
A & B & C \\
\end{array} \]

\begin{array}{c}
\vdots \text{ New input} \\
\vdots \text{ Pick middle item in list} \\
\end{array}

C > B

\begin{array}{c}
C \\
C \\
\end{array}

\begin{array}{c}
\vdots \text{ New input} \\
\vdots \text{ Pick middle item in list} \\
\end{array}

C = C

... Found match!
Q: Is binary search faster than linear search?
   - yes!

Q: What is the worst case for binary search?
   - Keep searching (halving list) until one element left

Determining the worst-case time complexity (by "cases")

\[
T(1) = c \\
T(n) = c + T(n/2)
\]  ... A "recurrence" relation

- assume some constant \( c \) for each iteration, e.g.,
- 3 steps to pick middle element (length/2, get element, store)
- 2 comparison operations
- for last element, 1 additional comparison
  \( \star \) ... we're being a bit careless with operation costs

Unfortunately, the function \( T \) (the recurrence relation) isn't in a "closed" form
- so it doesn't immediately tell us much about the worst-case cost
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!