Reminders

> HW1 is due next Wednesday
  – fixed a typo & added some clarification

> Lecture videos available on Canvas
  – student questions cannot be heard
  – back of heads can be seen

> Overloading sign up:
  https://goo.gl/forms/clZu6Wpy3xro69s13
Outline for Today

> Binary search on arrays
> Implementing binary search
**Review: Binary Search on Arrays**

**Input:** sorted array $A$ and a value $x$
- array can store any ordered set: ints, floats, strings, etc.

**Output:** index $i$ in $[0, A.length]$ s.t. $A[i-1] \leq x < A[i]$
- (partly vacuously true if $i = 0$ or $i = A.length$)
- returns where $x$ would be inserted to maintain ordering
- if $x$ appears multiple times, this returns the *last* one
  > move “$\leq$” to the right to get the *first* one
Review: Binary Search on Arrays

Maintain a region of the array that is unexplored

> Start with all white region
> Each iteration reduces size of the white region
> Finish with no white region
Review: Binary Search on Arrays

> Can reduce the size by looking at any element i:

> If A[i] <= x, then

> Else x < A[i]
Review: Binary Search on Arrays

- Can reduce the size by looking at any element $i$:

- If $A[i] \leq x$, then

- Else $x < A[i]$

only because $A$ is sorted
Review: Binary Search on Arrays

- **Binary Search**: look at the *middle* element of the white region

  ![Binary Search Diagram]

- This ensures size is *cut in half* each time
- Size of white region approximately $n / 2^k$ after $k$ iterations
- Done after $k = \lg n$ iterations
  - $O(1)$ per iteration, so $O(\log n)$ time
Review: Binary Search on Arrays

Linear search algorithm takes $O(n)$ time
  - start at $i = 0$
  - increase $i$ by 1 each time
  - stop when $A[i] \leq x < A[i+1]$

Binary search is an exponential speedup
  - ... since $\exp(\log n) = n$ ...
  - hard to overstate the importance of this
Simple application

> Suppose we want to find the index $i$ such that $u A[i] + v = x$

> Q: How do we solve this?
  – cannot compute $B[i] = u \times A[i] + v$
  – that would be exponentially slower!

> A: Binary search for $(x-v) / u$ since $u A[i] + v = x \implies A[i] = (x - v) / u$
Suppose we want to find the index $i$ such that $\tanh(A[i]) = x$ 
– (or sigmoid or any other monotonic function)

Q: How do we solve this?
A: Search for $A[i] = \tanh^{-1}(x)$

Q: What if the function is not easily invertible?
– come back to this later...
Problem: find the N-th largest number of the form $2^a3^b5^c$ for any $a, b, c \geq 0$ – i.e., numbers not divisible by anything other than 2, 3, or 5

Idea: generate the numbers in order, stopping at N

Sub-problem: find the $(m+1)$-st number of this form given first m – we can use the first m numbers to find the next one
Interview question 1

> **Sub-problem 1**: find the (m+1)-st number of this form given first m

> If (m+1)-st number is $2^a3^b5^c$, then $2^{a-1}3^b5^c$ is smaller and also of this form (assuming $a > 0$), so $2^{a-1}3^b5^c$ is **one of the first** m numbers
  - likewise for b and c
  - next number is 2, 3, or 5 times an earlier number

> In particular, if (m+1)-st is $2 \times$ an earlier number, then it must be the **smallest** $2 \times$ earlier not in the list
Interview question 1

> Sub-problem 2: given a list A[0..m-1] of the first m numbers, in increasing order, find the smallest i such that 2 A[i] > A[m-1]

> Q: how do we do that?
> A: binary search for (A[m-1]+1) / 2
Interview question 1

> Start with $A = [1]$
> **Algorithm:** for $m = 1$ to $N-1$
>   – binary search to find smallest $i$ s.t. $2 \times A[i] > A[m-1]$
>   – ... likewise for 3 and 5
>   – add the smallest of these three numbers to the list

> Q: total running time?
> A: $O(N \log N)$
Interview question 1

> Can we improve this further?

> Compare smallest $i$ s.t. $2 \times A[i] > A[m-1]$ and smallest $j$ s.t. $2 \times A[j] > A[m]$
  >  - binary search for $(A[m-1]+1) / 2$ and $(A[m]+1) / 2$
  >  - these two indices should be close

> Seems like we’re doing too much work by
  >  - could restrict to $A[i..m-1]$ with $i$ from last search
  >  - but we can do better...
Interview question 1

> Compare smallest i s.t. \(2 \times A[i] > A[m-1]\) and smallest j s.t. \(2 \times A[j] > A[m]\)

> **Observation:** they must be equal or differ by 1

> Since \(A[m]\) is smallest of 2x, 3x, 5x... either \(A[m] = 2 \times A[i]\) or \(A[m] < 2 \times A[i]\)

  – if \(A[m] < 2 \times A[i]\), then \(j = i\) since i is still big enough
  – if \(A[m] = 2 \times A[i]\), then \(A[m] < 2 \times A[i+1]\) and \(j = i+1\)
Interview question 1

> **Algorithm 2:** maintain indexes of smallest $i$, $j$, $k$ s.t.
  
  $2 \ A[i], 3 \ A[j], 5 \ A[k] > A[m-1]$

  for $m = 1$ to $N-1$
  
  – add the smallest of $2 \ A[i], 3 \ A[j], 5 \ A[k]$ to the list
  
  – increment $i, j$, and/or $k$ appropriately

> Q: total running time?
> A: $O(N)$
Interview question 1

> An example of how the fastest algorithm can be produced by starting with a basic technique + a lot of elbow grease

> Binary search disappears in the final answer even though we used to get there
  – we will see other examples of this
  – presentations of the best algorithm will often just describe the optimal solution directly without any binary search
    > makes you sound smarter if you do it that way
Interview question 2

> Find x in a sorted n x n table
  > every row and every column is sorted

> Algorithm 1: binary search every row
  > O(n log n) time
> Find x in a sorted n x n table
  – every row and every column is sorted

> Once again, the binary searches are doing too much work
  – usually the next answer is about the same as the previous one

> Not the case that the indexes only increase by 0 or 1 each time!
> BUT the total increase over n searches is at most n:
  – we only move left each time
  – so we can take at most n steps all together
Interview question 2

> Find x in a sorted n x n table
  > every row and every column is sorted

> **Algorithm 1**: linear every row starting from answer on previous
  > O(n) time

> Again, binary search disappears
Interview question 2

> Find x in a sorted n x n table
  - every row and every column is sorted

> **Algorithm 1**: linear every row starting from answer on previous
  - O(n) time

> Worth noting: O(n log n) is not much slower than O(n)
  - remember that log n is exponentially smaller than n
> Worth pointing out O(n log n) algorithm in an interview
Practical example: web search

> For each word, make record of all the web pages with that word
  – many terabytes of data
  – too much data for one machine...

> Partition web pages randomly across machines

> Each machine records where word appears in its own pages
  – have enough machines that each gets, say, 100 GB of data

> To look up all pages where word occurs:
  – send request to all machines
  – concatenate the lists they return
Practical example: web search

> Q: How does each machine get the pages for a word?
> A: sorting and binary search
  – each machine sorts its records on disk
  – look up a word by using binary search

> Algorithm works fine if A is on disk
  – only need the ability to look up A[i] for any i
  – can do this in Java using FileChannel instead of FileInputStream

> Cost is time for $\lg n$ disk seeks
Practical example: web search

> **Key lesson**: not always necessary to use dynamic data structures
  > – don’t always need hash tables and AVL trees (or B+ trees on disk)
  > – they are more complex, slower, and use more memory (by constant factors)

> Only need them to support intermixed updates and searches
  > – sorting and binary search are fine if all the updates come first

> Sorting also works fine if the data only changes occasionally
  > – at one point, web indexes were only changed *nightly*
    > this is not uncommon in practice
  > – can add new data and re-sort at night when not in use
Outline for Today

- Binary search on arrays
- Implementing binary search
Implementing Algorithms

Key Idea: invariants — facts that are always true
  – critical to correct implementation (and often run time analysis also)
  – often most of the hard work is getting these right

> **Rep invariant**: claim about data structures
  – always true (except briefly when mutating the data structures)
  – ex: in AVL trees, heights of two subtrees at any node differ by at most 1

> **Loop invariant**: claim about method state
  – always true at the *top* of the loop
Implementing Algorithms

> **Loop invariant**: claim about method state
  - always true at the *top* of the loop

> To prove that a loop is correctly implemented, check:
  - invariant is true initially
  - invariant remains true each time the loop body executes

> Then know the invariant is true after the loop
  - choose the invariant so that, when the loop exits, you have enough information to return a correct answer
Implementing Binary Search on Arrays

> Method state: indexes i and j
> Loop invariant: \(A[0], ..., A[i-1] \leq x \) and \(x < A[j], ..., A[n-1]\)

Notes on notation:
- \(A[0], ..., A[i-1] \leq x\) means \(A[0] \leq x\) and ... and \(A[i-1] \leq x\)
- vacuously true if \(i \leq 0\)
  - only making claims about indexes \(\geq 0\) and \(\leq i-1\)
Implementing Binary Search on Arrays

```c
int i = 0, j = n;

// Invariant: A[0], ..., A[i-1] <= x and x < A[j], ..., A[n-1]
while (i < j) {
    int m = (i + j) / 2;
    if (A[m] <= x)
        i = m + 1;
    else
        j = m;
}
return i;
```
Implementing Binary Search on Arrays

```c
int i = 0, j = n;

// Invariant: A[0], ..., A[i-1] <= x and x < A[j], ..., A[n-1]
while (i < j) {
    int m = (i + j) / 2;
    if (A[m] <= x)
        i = m + 1;  // A[i-1] = A[m] <= x
    else
        j = m;
}
return i;
```
Implementing Binary Search on Arrays

```c
int i = 0, j = n;

// Invariant: A[0], ..., A[i-1] <= x and x < A[j], ..., A[n-1]
while (i < j) {
    int m = (i + j) / 2;
    if (A[m] <= x) {
        i = m + 1;    // A[i-1] = A[m] <= x
    } else {
        j = m;
    }
}
return i;
```

Q: What about A[i], ..., A[m-1]?
Implementing Binary Search on Arrays

```c
int i = 0, j = n;
// Invariant: A[0], ..., A[i-1] <= x and x < A[j], ..., A[n-1]
while (i < j) {
    int m = (i + j) / 2;
    if (A[m] <= x) {
        i = m + 1;
    } else {
        j = m;
    }
}
return i;
```

Q: What about A[i], ..., A[m-1]?
A: Also <= x since A is sorted
Implementing Binary Search on Arrays

```c
int i = 0, j = n;

// Invariant: A[0], ..., A[i-1] <= x and x < A[j], ..., A[n-1]
while (i < j) {
    int m = (i + j) / 2;
    if (A[m] <= x)
        i = m + 1;
    else
        j = m;
}

return i;
```
Implementing Binary Search on Arrays

```c
int i = 0, j = n;

// Invariant: A[0], ..., A[i-1] <= x and x < A[j], ..., A[n-1]
while (i < j) {
    int m = (i + j) / 2;
    if (A[m] <= x)
        i = m + 1;
    else
        j = m;
}

return i;
```

x < A[m] = A[j]

x < A[m+1], .., A[j-1] since A is sorted
Implementing Binary Search on Arrays

// Invariant: A[0], ..., A[i-1] <= x and x < A[j], ..., A[n-1]
while (i < j) { ... }

> When we exit the loop, we have
  - i = j and A[0], ..., A[i-1] <= x and x < A[j], ..., A[n-1]
    > (actually, we only obviously have i >= j, but a more careful check shows i = j)
  - in other words, A[0], ..., A[i-1] <= x and x < A[i], ..., A[n-1]
  - thus, the problem specification says i is exactly what we promised to return
    > this is not uncommon...
    > loop invariants are often “weakened” versions of output promise
Implementing Binary Search on Arrays

> When we exit the loop, we return the right answer

> Q: Do we actually exit the loop?
  – usually you get that from the run time analysis,
  – but we did this somewhat sloppily....

> A: Yes, provided that i increases or j decreases every time.
> Q: Do they?
Implementing Binary Search on Arrays

> We set $m = (i + j) / 2$
>   - this means that $i \leq m \leq j$

> If we set $i = m + 1$, then $i_{\text{new}} \geq i_{\text{old}} + 1$
>   - looks good!

> If we set $j = m$, then $j_{\text{new}} \leq j_{\text{old}}$
>   - we are in trouble if $m = j_{\text{old}}$!
Implementing Binary Search on Arrays

- Can we have \( m = j \) when we set \( m = (i + j) / 2 \)?
- Can see that \( m \) will be closer to \( j \) when \( i \) is closer to \( j \)... so consider \( i = j - 1 \) (the worst case)
- Then \( m = (j - 1 + j) / 2 = (2j - 1) / 2 \)... 
- In Java, this integer division will truncate to \( j - 1 \) 
  - we got lucky!
- But Math.round((i + j) / 2.0) could loop forever!
Implementing Binary Search on Arrays

Lessons:

1. invariants are critical to implementing complex algorithms & data structures

2. implementing algorithms correctly requires careful attention to detail
   > easy to make mistakes on this, one of the easiest algorithms we will see!
   > this comes up in interviews too

3. if a library implementation is available, use it!
   > don’t waste your time or risk releasing buggy code
Implementing Binary Search on Arrays

> For the most part, we will stick to pseudocode from here on
  – you’ll still need to write code in the HWs

> From a theory perspective, this wasn’t a real problem
  – we only run into it when j - i is small
  – we could switch to linear search when j - i < 1000
    asymptotic complexity would be the same