Binary Search

CS16: Introduction to Data Structures & Algorithms
Spring 2020
Outline

- Binary search
- Pseudo-code
- Analysis
- In-place binary search
- Iterative binary search
Phonebook Search

Activity #1

2 min
Phonebook Search

Activity #1

2 min
Phonebook Search

Activity #1

1 min
Phonebook Search

Activity #1
The Problem

- Is an item \(x\) in a sorted array?
  - ex: is 5 in the array above?
- Idea #0
  - scan array to find \(x\)
  - \(O(n)\) running time
- Can we do better?
The Problem

- Observation #1
  - we can stop searching for 11 if we reach 12
  - we can stop searching for x if we reach y > x
- Why?
  - since array is sorted, 11 can’t be after 12
  - since array is sorted, x can’t be after y
- But what if we’re looking for 25?
The Problem

- Observation #1
  - we can stop searching for $x$ if we reach $y > x$

- Observation #2
  - what happens if we compare $x$ to middle element?
    - if $x = \text{mid}$, then we found $x$
    - if $x < \text{mid}$, then $x$ cannot be in right half of array
    - if $x > \text{mid}$, then $x$ cannot be in left half of array
The Problem

- Using observation #2
  - We got rid of half the array!
- What if do it again?
  - same problem…but half the size!
- Does this remind you of something?
The Problem

Find 5

How many comparisons?
Analysis

- How many comparisons on array of size $n$?
  - after each comparison we cut array in half
  - how many times can we split array in 2 before we get array of size 1?
    - if $n=2^k$ for some $k$, then $\log_2(n)=k$
- So what is runtime of binary search?
  - $O(\log n)$?
- Let’s look at pseudo-code!
function `binarysearch(A,x)`:
  
  if A.size == 0:
      return false
  
  if A.size == 1:
      return A[0] == x

  mid = A.size / 2

  if x == A[mid]:
      return true
  if x > A[mid]:
      return binarysearch(A[mid+1...end], x)
  if x < A[mid]:
      return binarysearch(A[0...mid-1], x)
Binary Search Analysis

- Binary search implementation is recursive...
- So how do we analyze it?
  - write down the recurrence relation
  - use plug & chug to make a guess
  - prove our guess is correct with induction
Binary Search Analysis

- What is the recurrence relation of Binary Search?
  \[ T(n) = T(n/2) + f(n), \text{ with } T(1) = c \]
  - where \( f(n) \) is the work done at each level of recursion
- Where does \( T(n/2) \) come from?
  - because we cut problem in half at each level of recursion
- Why is base case \( T(1) = c \)?
- What is \( f(n) \)?
function \texttt{binarysearch}(A,x):
    if A.size == 0:
        return false
    if A.size == 1:
        return A[0] == x
    mid = A.size / 2
    if x == A[mid]:
        return true
    if x > A[mid]:
        return \texttt{binarysearch}(A[mid+1...end], x)
    if x < A[mid]:
        return \texttt{binarysearch}(A[0...mid-1], x)

\textbf{copying half the array... is } O(n)!!
Binary Search Analysis

- Recurrence relation:

\[ T(n) = T(n/2) + c_1 n + c_2, \quad T(1) = c_0 \]

- Plug and chug:

\[
\begin{align*}
T(1) &= c_0 \\
T(2) &= T(1) + 2c_1 + c_2 = c_0 + 2c_1 + c_2 \\
T(4) &= T(2) + 4c_1 + c_2 = c_0 + (4 + 2)c_1 + 2c_2 \\
T(8) &= T(4) + 8c_1 + c_2 = c_0 + (8 + 4 + 2)c_1 + 3c_2 \\
T(n) &= c_0 + \left( n + \frac{n}{2} + \frac{n}{4} + \cdots + 4 + 2 \right) c_1 + (\log n) c_2
\end{align*}
\]

What is \( T(n) \)?

converges to \( 2n \) as \( n \) gets large
Binary Search Analysis

- $T(n)$ is $O(n)$
  - is this a proof?
- As bad as scanning the array…
- But on Slide #13 we said Binary Search was $O(\log n)$!

Analysis

- How many comparisons on array of size $n$?
  - after each comparison we cut array in half
  - how many times can we split array in 2 before we get array of size 1?
    - if $n=2^k$ for some $k$, then $\log_2(n)=k$
- So what is runtime of binary search?
  - $O(\log n)$?
- Let’s look at pseudo-code!
What happened?
Subtlety in Binary Search!

- In our implementation we copied half the array
  - at each level of recursion this cost us $O(n)$
  - so runtime went back up to $O(n)$
Q: What should we do?
In-Place Binary Search

- We should keep reusing the original array
  - no copying of elements!
- We should implement it “in-place”
function binarysearch(A, lo, hi, x):
    if lo >= hi:
        return A[lo] == x

    mid = (lo + hi) /2

    if x == A[mid]:
        return true
    if x > A[mid]:
        return binarysearch(A, mid+1, hi, x)
    if x < A[mid]:
        return binarysearch(A, lo, mid-1, x)
In-Place Binary Search

\[ A = [0, 3, 8, 10, 10, 15, 18] \]
\[ x = 7 \]
In-Place Binary Search

A = [0, 3, 8, 10, 10, 15, 18]

x = 7

Activity #2
In-Place Binary Search

\[ A = [0, 3, 8, 10, 10, 15, 18] \]
\[ x = 7 \]
In-Place Binary Search

\[ A = [0, 3, 8, 10, 10, 15, 18] \]
\[ x = 7 \]
In-Place Binary Search

\[ A = [0, 3, 8, 10, 10, 15, 18] \]
\[ x = 7 \]
In-Place Binary Search

\[ A = [0, 3, 8, 10, 10, 15, 18] \]
\[ x = 7 \]
function binarysearch(A, lo, hi, x):
    if lo >= hi:
        return A[lo] == x
    mid = (lo + hi) / 2
    if x == A[mid]:
        return true
    if x > A[mid]:
        return binarysearch(A, mid+1, hi, x)
    if x < A[mid]:
        return binarysearch(A, lo, mid-1, x)
In-Place Binary Search

- Does $O(1)$ ops at each level of recursion
- Recurrence is now

\[ T(n) = T(n/2) + c_1, \text{ with } T(1) = c_0 \]

- Plug & Chug:

\[
\begin{align*}
T(1) &= c_0 \\
T(2) &= T(1) + c_1 = c_0 + c_1 \\
T(4) &= T(2) + c_1 = c_0 + 2c_1 \\
T(8) &= T(4) + c_1 = c_0 + 3c_1 \\
T(n) &= c_0 + (\log n) \cdot c_1
\end{align*}
\]
In-Place Binary Search

- So in-place binary search is
  - \(O(\log n)\)
- Is this a proof?
Iterative Binary Search

function `binarysearch`(A, x):
    lo = 0
    hi = A.size - 1

    while lo < hi
        mid = (lo + hi) / 2
        if A[mid] == x:
            return true
        if A[mid] < x:
            lo = mid + 1
        if A[mid] > x:
            hi = mid - 1
    return [lo] == x

 Recursive algorithms can be implemented iteratively