Binary Search

CS16: Introduction to Data Structures & Algorithms
Seny Kamara - Spring 2017
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

0 min
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?

Let’s use the fact that array is sorted...
## The Problem

| 1 | 1 | 3 | 4 | 7 | 8 | 10 | 10 | 12 | 18 | 19 | 21 | 23 | 23 | 24 |

- **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 \( x \) must be in left half of array
  - if \( x > \text{mid} \), then \( x \) must be in right half of array
The Problem

- Observation #2
  - what happens if we compare $x$ to middle element?
    - if $x = \text{mid}$, then we found $x$
    - if $x < \text{mid}$, then $x$ must be in left half of array
    - if $x > \text{mid}$, then $x$ must be in right half of array
  - We got rid of half of the array!
- What if do it again?
  - same problem…but half the size!
The Problem

Find 5

1 1 3 4 7 8 10 | 10 | 12 | 18 | 19 | 21 | 23 | 23 | 24

5 < 10

1 1 3 4 7 8 10

5 > 4

7 8 10

5 < 8

7

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 $1$?
    - $\log_2(n)$ times
- So what is runtime of binary search?
  - $O(\log n)$?
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 the recurrence relation
  - And solve it!
- The recurrence relation of Binary Search is
  - \( T(n) = T(n/2) + f(n) \), with \( T(1) = c \)
  - where \( f(n) \) is work done at each level of recursion
- Why?
  - because we cut the problem in half each time
## Binary Search Pseudo-Code

```plaintext
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)
```

The time complexity of binary search is $O(\log n)$ because at each step, the search space is halved. The diagrams illustrate the decision-making process and the time complexities at each step. The highlighted part indicates where the array is copied, which takes $O(n)$ time. Therefore, the overall time complexity is $O(n)$ if the array needs to be copied.
Binary Search Analysis

- Plug and chug!

\[
T(n) = T(n/2) + c_1 \cdot n + c_2
\]

\[
T(1) = c_0
\]

\[
T(2) = T(1) + 2c_1 + c_2 = c_0 + 2c_1 + c_2
\]

\[
T(4) = T(2) + 4c_1 + c_2 = c_0 + (4 + 2) \cdot c_1 + 2c_2
\]

\[
T(8) = T(4) + 8c_1 + c_2 = c_0 + (8 + 4 + 2) \cdot c_1 + 3c_2
\]

\[
T(n) = c_0 + \left( \frac{n}{2} + \frac{n}{4} + \cdots + 4 + 2 \right) \cdot c_1 + (\log n) \cdot c_2
\]

What is \(T(n)\)?

linear function

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

- \( T(n) \) is \( O(n + \log n) \)
- As bad as scanning array...
  - But in our example it was \( O(\log n)! \)
What happened?
Subtlety in Binary Search!

- In our implementation we copied half the array
  - at each step, this cost us $O(n)$
  - so runtime went back up to $O(n)$

Common pitfall when implementing efficient algorithms
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`\( (A, \text{lo}, \text{hi}, x) \): 
  
  if `lo` >= `hi`:
    return \( A[\text{lo}] == x \)

  mid = \( (\text{lo} + \text{hi}) / 2 \)

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

Activity #2
In-Place Binary Search

Activity #2
In-Place Binary Search

Activity #2
In-Place Binary Search

1 min

Activity #2
In-Place Binary Search
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:
  \[
  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(2^k) = c_0 + kc_1
  \]
In-Place Binary Search

- So if $2^k = n$ we have
  - $T(n) = c_0 + (\log n) \cdot c_1$
- So in-place binary search is $O(\log n)$!
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