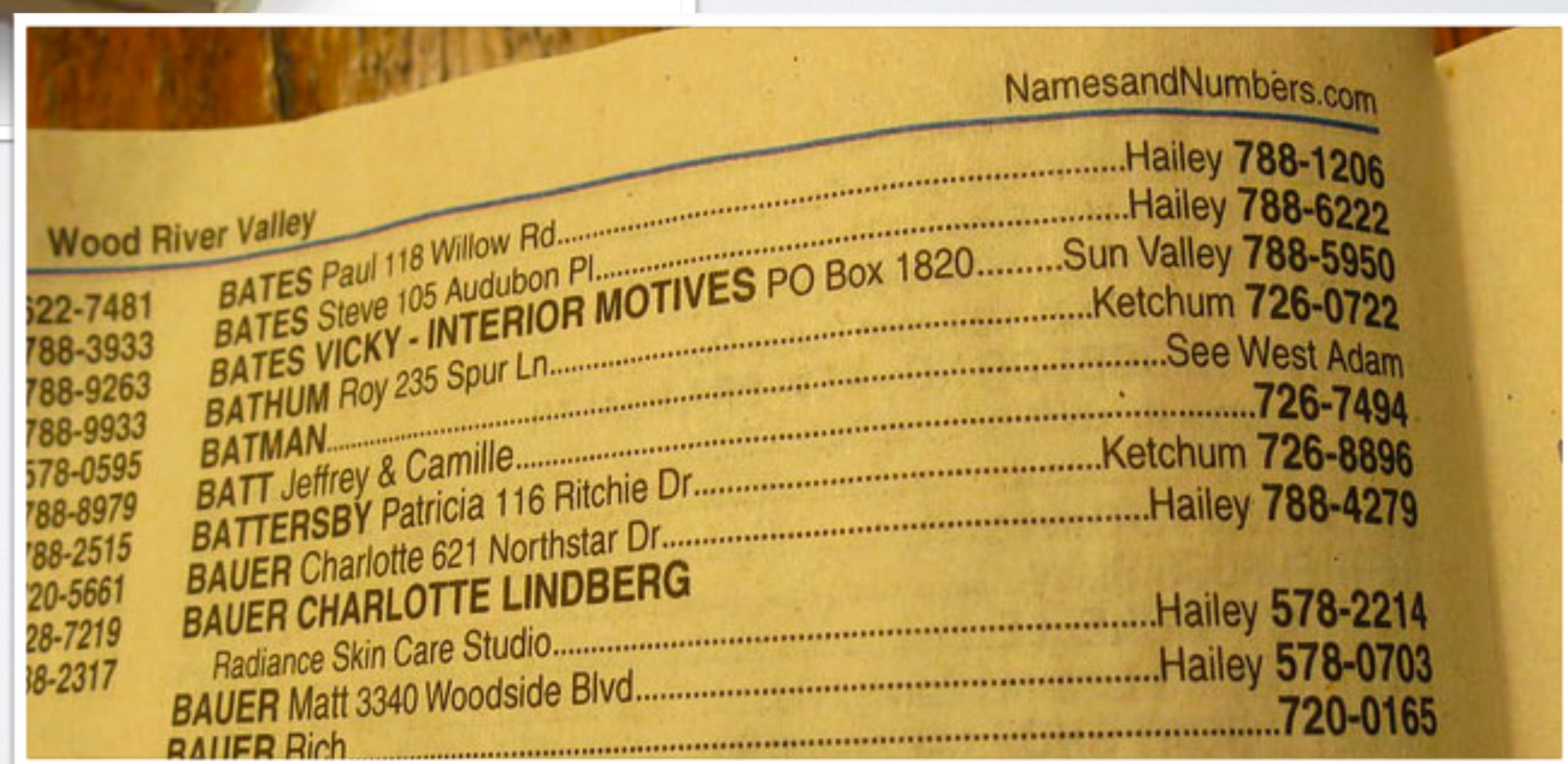
Binary Search

CS16: Introduction to Data Structures & Algorithms

Spring 2020

Outline

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


NamesandNumbers.com

Wood River Valley

522-7481
788-3933
788-9263
788-9933
578-0595
788-8979
788-2515
720-5661
728-7219
88-2317

BATES Paul 118 Willow Rd........................Hailey 788-1206
BATES Steve 105 Audubon Pl........................Hailey 788-6222
BATES VICKY - INTERIOR MOTIVES PO Box 1820.........Sun Valley 788-5950
BATHUM Roy 235 Spur Ln........................Ketchum 726-0722
BATMAN........................See West Adam
BATT Jeffrey & Camille........................726-7494
BATTERSBY Patricia 116 Ritchie Dr........................Ketchum 726-8896
BAUER Charlotte 621 Northstar Dr........................Hailey 788-4279
BAUER CHARLOTTE LINDBERG
Radiance Skin Care Studio........................Hailey 578-2214
BAUER Matt 3340 Woodside Blvd........................Hailey 578-0703
BAUER Rich........................720-0165

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

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

- 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

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

mid

- 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 = mid`**, then we found **`x`**
  - if **`x < mid`**, then **`x`** cannot be in right half of array
  - if **`x > mid`**, then **`x`** cannot be in left half of array

The Problem

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

- 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

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`?
    - 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!

Binary Search 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)
```

Assume `A.size` is power of 2

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)`?

Binary Search Pseudo-Code

```
function binarysearch(A,x):
  if A.size == 0:               O(1)
    return false                O(1)
  if A.size == 1:               O(1)
    return A[0] == x            O(1)

  mid = A.size / 2              O(1)

  if x == A[mid]:               O(1)
    return true                 O(1)
  if x > A[mid]:                O(1)
    return binarysearch(A[mid+1…end], x)
  if x < A[mid]:                O(1)
    return binarysearch(A[0…mid-1], x)
```

copying half the array… is O(n)!!

Binary Search Analysis

- Recurrence relation:

linear

$$T(n) = T(n/2) + \underbrace{c_1 n + c_2,}_{\text{linear}} \quad T(1) = c_0$$

- Plug and chug:

$$T(1) = c_0$$

$$T(2) = T(1) + 2c_1 + c_2 = c_0 + 2c_1 + c_2$$

$$T(4) = T(2) + 4c_1 + c_2 = c_0 + (4 + 2)c_1 + 2c_2$$

$$T(8) = T(4) + 8c_1 + c_2 = c_0 + (8 + 4 + 2)c_1 + 3c_2$$

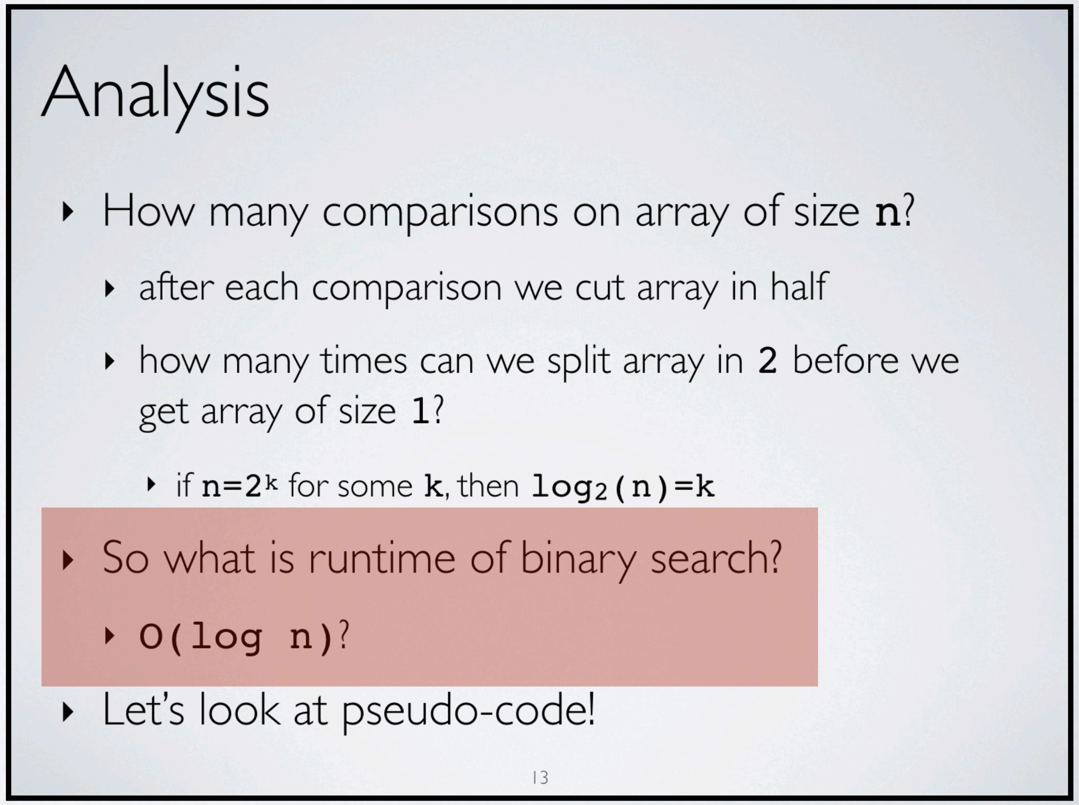
$$T(n) = c_0 + \left(n + \frac{n}{2} + \frac{n}{4} + \cdots + 4 + 2\right) c_1 + (\log n)c_2$$

converges to 2n as n gets large

What is T(n)?

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)`!

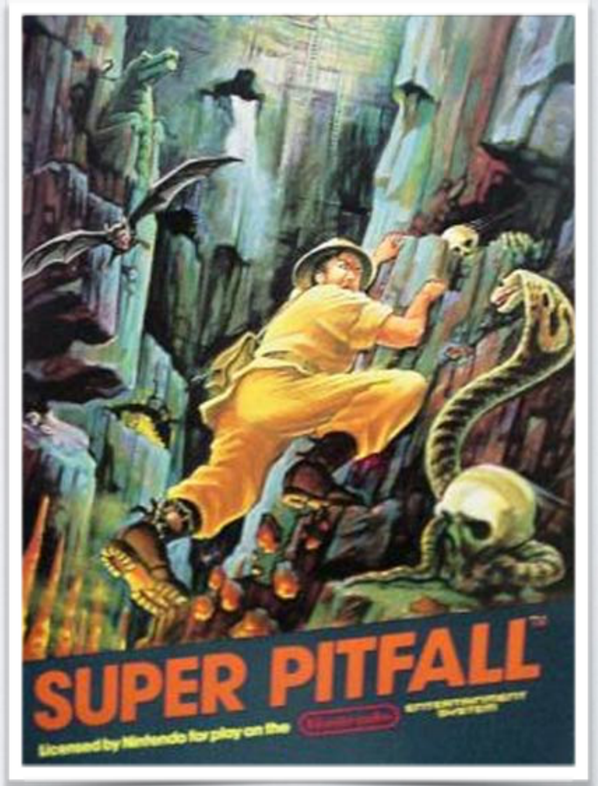


What happened?

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)`**

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"

In-Place Binary Search Pseudo-Code

```
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
```

Activity #2

4 min

In-Place Binary Search

```
A = [0, 3, 8, 10, 10, 15, 18]
x = 7
```

Activity #2

4 min

In-Place Binary Search

```
A = [0, 3, 8, 10, 10, 15, 18]
x = 7
```

Activity #2

3 *min*

In-Place Binary Search

```
A = [0, 3, 8, 10, 10, 15, 18]
x = 7
```

Activity #2

2 min

In-Place Binary Search

```
A = [0, 3, 8, 10, 10, 15, 18]
x = 7
```

Activity #2

1 min

In-Place Binary Search

```
A = [0, 3, 8, 10, 10, 15, 18]
x = 7
```

Activity #2

0 min

In-Place Binary Search Pseudo-Code

```
function binarysearch(A, lo, hi, x):       
   if lo >= hi:                            O(1)
     return A[lo] == x                     O(1)

   mid = (lo + hi) /2                      O(1)

   if x == A[mid]:                         O(1)
     return true                           O(1)
   if x > A[mid]:                          O(1)
     return binarysearch(A, mid+1, hi, x)
   if x < A[mid]:                          O(1)
     return binarysearch(A, lo, mid-1, x)
                                           O(1)
```

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:

$$T(1) = c_0$$
$$T(2) = T(1) + c_1 = c_0 + c_1$$
$$T(4) = T(2) + c_1 = c_0 + 2c_1$$
$$T(8) = T(4) + c_1 = c_0 + 3c_1$$

$$T(n) = c_0 + (\log n) \cdot c_1$$

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